# Applying Inspection to Object-Oriented Software

by

F Macdonald, J Miller, A Brooks, M Roper and M Wood

Empirical Foundations of Computer Science (EFoCS)

RR-95-188 [EFoCS-14-95]		June 1995

# Applying Inspection to Object-Oriented Software

F Macdonald, J Miller, A Brooks, M Roper and M Wood

Empirical Foundations of Computer Science (EFoCS) [*]

June 1995

**Abstract**

*The benefits of the object-oriented paradigm are widely cited. At the same time, inspection is deemed to be the most cost-effective means of detecting defects in software products. Why then, is there no published experience, let alone quantitative data, on the application of inspection to object-oriented systems? We describe the facilities of the object-oriented paradigm and the issues that these raise when inspecting object-oriented code. Several problems are caused by the disparity between the static code structure and its dynamic runtime behaviour. The large number of small methods in object-oriented systems can also cause problems. We then go on to describe three areas which may help mitigate problems found. Firstly, the use of various programming methods may assist in making object-oriented code easier to inspect. Secondly, improved program documentation can help the inspector understand the code which is under inspection. Finally, tool support can help the inspector to analyse the dynamic behaviour of the code. We conclude that while both the object-oriented paradigm and inspection provide excellent benefits on their own, combining the two may be a difficult exercise, requiring extensive support if it is to be successful.*

[*]Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, U.K., +44 (0)141 552 4400, fraser@cs.strath.ac.uk

## 1 Introduction

The inspection process was first described by Michael Fagan in 1976 [8]. It is a structured method for statically validating documents. A team consisting of the author of the document, a moderator, a recorder, and a number of inspectors proceed to inspect the document using a multi-stage process. The inspection starts with a period of planning, where the participants are selected and materials prepared. The next stage is the overview, where the group receive a briefing on the document under inspection. During preparation, each member of the team individually becomes familiar with the material and starts to gather defects. The preparation stage is then followed by the actual inspection meeting, which involves the entire team. At this point the team categorise each defect for type, class and severity and record it for the author to fix. This meeting is followed by a period of rework, where the author addresses each defect. Finally, a follow-up is carried out to ensure each defect has been addressed.

The benefits of inspection are widely known. In addition to Fagan's papers describing his experiences [8, 9], there are many other favourable reports. For example, Doolan [6] reports a 30 times return on investment for every hour devoted to inspections. Russell [22] reports a similar return of 33 hours of maintenance saved for every hour of inspection invested. Davis [5] indicates that inspection can cut development costs by 25 to 30 percent.

The last decade or so has seen an explosion in the use of object-oriented techniques, with several new programming languages being developed to provide the facilities required to implement object-oriented systems. One of the most popular is C++, a hybrid object-oriented language developed from C by Bjarne Stroustrup [24]. Another is Eiffel, developed by Bertrand Meyer [19]. The object-oriented paradigm provides several new concepts. The most fundamental is the *class*, which encapsulates data and the methods used to access that data in one package. Similar to the use of ADTs in traditional programming, classes provide a means to enforce data privacy. Classes cannot be used directly, instead an instance of the class (an *object*) has to first be created. This is similar to declaration of a variable in a procedural programming language. *Inheritance* allows the features of one class to be used and extended by a derived class. It allows an existing class to be reused, instead of coding the new class from scratch, but also provides the means to add new facilities. *Polymorphism* is the ability of a declaration to refer to objects of more than one class, provided they are related by a common base class. For example, if class B inherits from class A, wherever a reference is made to an instance of class A, an instance of class B can be substituted. This is allowed because an instance of class B will include all the properties of class A, although it may also define further properties. Finally, the concept of *genericity* is the ability to design classes which can be parameterised with respect to type and operations, so that, for example, we may have a generic stack class which can be instantiated to perform stack operations with any type of object.

The object-oriented paradigm is generally perceived to provide several benefits. Firstly, it provides the means for producing a good design. Classes provide a natural way to encapsulate data, while also making the system very modular. Since a class tends to be designed around a set of data and the functions associated with that data, classes are usually highly cohesive, yet the coupling between them is weak. These are usually deemed to be required properties of any good design. Another potentially major benefit is code reuse [15], which occurs in two ways. Reuse within a system is achieved through inheritance. Also, a good object-oriented system should produce classes which can be reused in other systems. In fact, there is a major interest in producing reusable libraries of classes. Generic classes are always good candidates for reuse. Yet another quoted benefit is reduced maintenance effort. As the details of the implementation of the class are kept hidden from the client, the maintainer is free to make changes to the class internals, provided the interface is not changed. Similarly, inheritance, along with polymorphism, reduces the maintenance need when extending the system. Class functionality can easily be extended by subclassing, and these subclasses can then be used wherever their parent class is already used. A final claim of object-oriented programming is that it is more natural [2]. This is due to its data-centred nature. The system is built by modelling an abstraction of the real world, with everything based on real objects.

Given that inspection is supposedly the most cost-effective means of finding defects, and the popularity of object-oriented programming languages, it is surprising that there is little, if any, work on inspecting object-oriented code, as indicated by Jones [12] and supported by our own search of the literature. In this paper we will describe some of the issues in applying inspection to object-oriented software, and some of the techniques that may be applicable in improving the effectiveness of such inspection.

The rest of this paper is organised as follows. Section 2 describes existing work which has relevance to inspection of object-oriented software: this includes testing and maintenance of object-oriented software. Section 3 is devoted to providing examples of the difficulties that can be found in inspecting object-oriented code. We provide examples in both a pure object-oriented language (Eiffel) and a hybrid language (C++). Between them, these two languages demonstrate almost all the features available in any object-oriented programming language. Section 4 describes some of the techniques which can be applied to help reduce these difficulties. Section 5 contains our conclusions.

## 2 Related Work

As indicated in the introduction, there is no published experience on applying inspection to object-oriented software. However, there has been other work on object-oriented systems which has relevance to inspection. Here we describe testing and maintenance of object-oriented software, as well as program understanding. We also describe some qualities of object-oriented code that may be subject to inspection.

### 2.1 Testing Object-Oriented Software

One area that has relevance to the inspection of object-oriented software is that of testing. For example, Jüttner *et al*. [13] describe some of the problems that object-oriented systems can present for integration testing. Since object-oriented software is characterised by objects and classes which interact using message passing, inheritance and using relationships, integration testing requires a different approach to that used for procedural software, where the system consists of modules linked by function calls and common data. In fact, according to Jüttner *et al*. [13], the relationships within object-oriented code are much more intense than those in procedural software, which blurs the distinction between class and integration testing, as compared with the equivalent module and integration testing of procedural systems. Furthermore, there is no simple hierarchical system structure to which a systematic testing strategy can be applied.

Some of the difficulties found when testing object-oriented software are as follows [13]. Inheritance poses a difficulty for class testing. Although it is reasonable to expect that a class may be tested in isolation, the use of inheritance may mean there are hidden interactions between inherited code and the new class. Therefore, both the old and new methods, along with their interactions, must be tested. There is also a problem in trying to trace control flow. The code for each class in no way describes the order in which the methods may be called. This is compounded by polymorphism and dynamic binding, which precludes the static prediction of which methods will actually be called at runtime. Stubbing is also made more difficult, due to a combination of the different system structure, increased encapsulation, and mutual dependencies between methods.

Finally, the use of short methods moves the complexity from method bodies to the interactions between them. Again, when combined with polymorphism and dynamic binding, this results in complex interactions which hinder testing.

Although inspection is affected by the same features that make testing of object-oriented code more difficult, inspection is fundamentally different from testing in that the code is never actually run while under scrutiny. Instead, inspection is a form of static analysis where the key task is understanding the code to discover defects in the program logic. This static nature of inspection has implications for object-oriented code, especially where such features as polymorphism and genericity, which depend on the dynamic behaviour of the code, are concerned. When these features are used, the dynamic structure of the system is very different to the static structure defined by the source code. In fact, the problems that this creates for inspection are not dissimilar to those found when trying to understand object-oriented code to perform maintenance.

## 2.2 Maintaining Object-Oriented Software

Wilde *et al*. [27] describe two prerequisites for successful maintenance. First of all the system has to be easy to change. This is a prime goal of the object-oriented paradigm, and is achieved, to some degree, by the encapsulation mechanism of the class. The second requirement is an in-depth understanding of the software which is being maintained. In both this paper and an earlier paper by Wilde and Huitt [26], the authors describe the problems that object-oriented software pose for maintenance.

The first problem described by Wilde *et al*. [27] is the tracing of dependencies in a system which makes use of inheritance. The method being examined may not only be used with the class it is declared in, but also with any subclasses which may inherit from this class. Similarly, any declarations referred to may exist in this class or in some superclass. The severity of this problem depends on the depth of the inheritance hierarchy. There is also a problem where the system contains a large number of small methods. For example, in one Smalltalk system described, of the 450 methods present, over 80% consisted of two lines or less [27]. The large number of methods increases the number of relationships that exist in the system, many of which must be understood by the maintainer to ensure the correctness of any changes made. It may also be the case that what is considered to be a single unit of functionality in a procedural program is spread over several classes in an object-oriented system. To understand one piece of functionality then requires the study of several classes and their interactions.

Wilde and Huitt's earlier paper describes some additional issues [26]. The use of polymorphism and dynamic binding prevent the exact method invocation from being predicted statically. Instead, the maintainer has a set of methods, any one of which may be invoked. It is therefore more difficult to identify dependencies within the system. There is also a problem when the maintainer becomes familiar with one or two versions of a method. This may lead to the assumption that all versions of that method have the same behaviour, when in fact each method has a slightly different behaviour. Furthermore, there may be far more possible dependencies within an object-oriented system than within a procedural system. All of these problems hinder the program understanding task which the maintainer must perform to undertake a correct modification to the system.

## 2.3 Delocalised Plans and Program Understanding

In addition to work on object-oriented systems, some work on understanding procedural systems has relevance. For example, Soloway *et al.* [23] describe a problem that is found in trying to understand procedural programs which is equally applicable to object-oriented systems, referred to as the presence of "delocalised plans". A delocalised plan is defined as being when "pieces of code that are conceptually related are physically located in non-contiguous parts of the program". The large number of small methods found in object-oriented systems may mean they contain many of these delocalised plans. Soloway and his colleagues describe two studies they performed. The first used professional programmers from the Jet Propulsion Laboratory to study the design of software documentation for maintenance (this is also described in an earlier work by two of the authors [17]). The second study involved protocol analysis of a code inspection performed at IBM.

The subjects in the maintenance study appeared to employ two strategies for program understanding: micro-strategies, which were used when trying to understand lines of code, and macro-strategies, which were used at a higher level. A typical micro-strategy used was an “Inquiry Episode”. Here, the subjects followed a pattern consisting of several steps: read some code, ask a question about it, form a conjecture, perform a search for confirmation, then come to some answer. These inquiry episodes are triggered when a subject comes across a delocalised plan. However, in object-oriented code the vital step of searching for confirmation may be hard, because methods tend to be more distributed. This is complicated by the use of inheritance and polymorphism, as will be described later.

Soloway *et al.* also identified two types of macro-strategies. A systematic strategy involved the programmer trying to understand the entire program, while an as-needed strategy was used by programmers wishing only to understand the portion necessary to implement a change. For inspection, the systematic strategy may be the most appropriate, since the inspector must understand all the code to evaluate it for defects. However, as the authors point out, a systematic strategy can only be applied to small programs. The complexity and size of larger systems prevent anyone from understanding the entire system (in reasonable time), and so there must be some means of splitting the system into logically cohesive, but independent units.

Finally, the authors also demonstrate that program understanding is crucial to inspection. In their study of code inspection meetings, they found that 34% of the time was spent on achieving clarity of the code. The more complex the code is, the longer it takes to understand. Since object-oriented systems tend to consist of large numbers of interactions such as message passing and inheritance, this may make them harder to understand and therefore more time-consuming to inspect. Object-oriented systems shift the complexity from method bodies to the interactions between them [13].

## 2.4 Inspecting for Quality

In addition to inspecting for defects in the code, where ‘defect’ roughly equates to ‘bug’, there are other qualities which code can be inspected for. Fagan [9] indicates some of these, including portability, installability, and usability. While it is intuitively obvious that we require code of good quality, and while many people feel that they can differentiate between low quality and high quality code, it is difficult to define such an attribute. One attempt to define quality for object-oriented code is the “Law of Demeter” [16], a style rule for the construction of methods which minimises the number of objects which a method can send messages to. The authors start by defining an acquaintance class as a class which is not an argument or instance variable of a method, but which supplies a method used in this method. A preferred acquaintance class is a class of global variables used in the method or a class of objects created within the method. A method’s preferred supplier classes are either preferred acquaintance classes or an instance variable or argument class of this method. The Law of Demeter then has two forms. The first is the class form, which has two versions. The minimisation version states “Minimise the number of acquaintance classes over all methods.” The strict version states “All methods may have only preferred supplier classes.” The object form of the law states “All methods may have only preferred-supplier objects,” where preferred-supplier objects are argument variables, the current object and any subparts of the current object, any directly created objects or any global objects. These laws restrict object communication and are intended to reduce dependencies between classes, promoting maintainability and understandability. While it is the programmer’s responsibility to apply the Law of Demeter, a compiler can be used to enforce the class form’s strict version. The object version is more difficult, however, and cannot be enforced at compile time. Currently, the only technique which may be used to check for its use is inspection, but it may be difficult to inspect large amounts of code for compliance with the law. Every method must be checked for its use by classifying all objects used within that method. With large systems, this is time-consuming and error-prone. Other indicators of code quality have similar problems.

Another quality which object-oriented code can be inspected for is reusability. Over the last ten years, there has been increasing interest in the area of object-oriented domain analysis [1]. Domain analysis is the study of a specific application area to identify potential reuse of analysis, design and code. Where identification of reusable code is concerned, inspection would seem to be the ideal time to conduct such activities. This would be achieved by the inclusion of a domain analyst in the inspection. The analyst would then be able to give input on the suitability of code for reuse. If necessary, the analyst can give guidance on making appropriate changes to “almost reusable” code to allow full reuse. Since reusability is deemed to be a major benefit of the object-oriented paradigm, inspecting for reusability should be an important part of object-oriented code inspection.

Code quality and reusability are two qualities of object-oriented code that are judged using inspection. It is therefore important that object-oriented code is amenable to such inspection, not just to remove defects but to ensure that the code itself is of a high standard and is capable of being maintained and reused in the manner which the object-oriented paradigm is reputed to support.

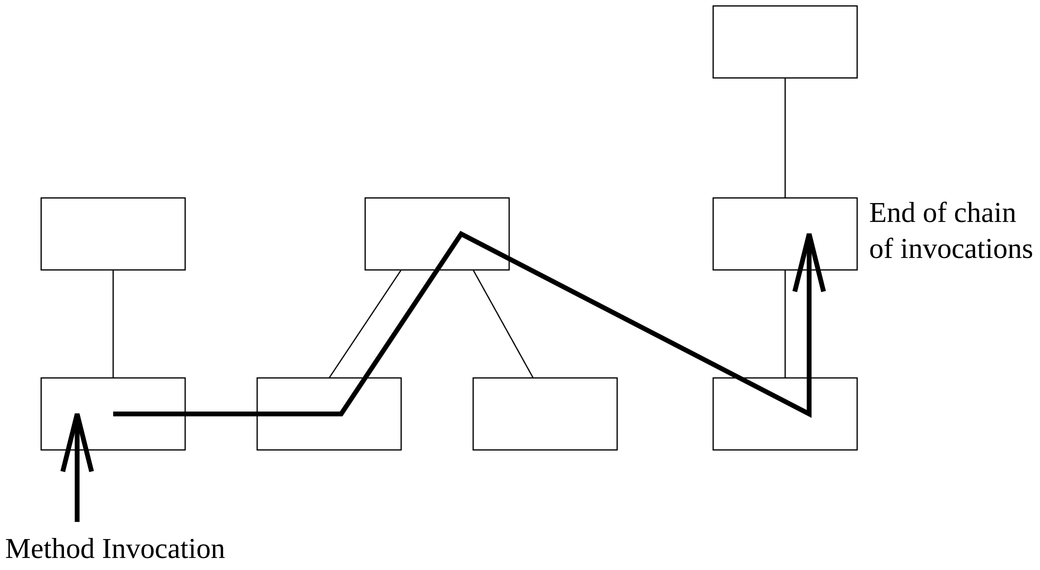


Figure 1: A sequence of method invocations in an object hierarchy.

## 3 Issues in Applying Inspection to Object-Oriented Software

### 3.1 Method Size and Distribution

A typical object-oriented system consists of many small methods, each of which provides only a little functionality (for examples of this, see [27]). Therefore to understand more than just trivial parts of the system, large numbers of these methods must be cognitively grouped together. It may be difficult to reconstruct the meaning of this code, however, and we have described Soloway *et al.*'s [23] work on "delocalised plans," which occur when conceptually related code is spread over spatially distributed parts of the program.

The method distribution problem can be easily illustrated. Consider Figure 1 which depicts an object hierarchy. The arrow on the lower left object indicates an invocation of one method. However, it may be found that this method in turn invokes other methods, which in turn invoke yet other methods, and so on. It quickly becomes apparent that there may be a large of chain of method invocations which must be followed. This is illustrated by the arrow moving through several objects. It must also be considered that each method may invoke multiple methods, which themselves invoke multiple methods. This greatly increases the paths that must be followed. It follows that the complexity of the system is transferred from method bodies to the interactions between them. Inspection is then made harder by having to understand all these interactions to predict the effect of a single method call.

## 3.2 Inheritance

Inheritance is perhaps the most powerful feature of object-oriented programming, and is one of the major differences between object-based and object-oriented code. Inheritance allows the behaviour of one class to be reused and extended by another class. The derived class (subclass or child class) has all the features of the base class (superclass or parent class), but adds further behaviour which generally indicates some type of specialisation.

For example, in categorising a population of students, we may have some class `student` which has properties common to all students, such as name, matriculation number and so forth. However, we may require a class called `post_graduate_student`, which has all the attributes of `student`, but also requires additional details such as supervisor and topic. We could create a completely separate class, and paste in the code from `student` but this is wasteful and could make maintenance difficult, as any changes in the common code must be made to both classes. Instead, we define `post_graduate_student` to inherit from `student`, then add only the new information. Any change made to `student` is then automatically reflected in `post_graduate_student`.

Despite the advantages of inheritance, including code reuse and reduced maintenance effort, there is difficulty in understanding such code due to the distribution of behaviour over several classes. These problems are detailed below.

### 3.2.1 Single Inheritance

Given the code in Figure 2, and an instance `new_manager` of class `MANAGER`, when we inspect the line

```
class EMPLOYEE

feature
   calculate_tax is
     ...
   end

end -- EMPLOYEE

class SALESPERSON
inherit
   EMPLOYEE
feature
  ...

end -- SALESPERSON

class MANAGER
inherit
   EMPLOYEE
feature
   award_bonus is
      do
        bonus := sales_increase * bonus_rate
      end

end -- MANAGER
```

Figure 2: Example Eiffel code for single inheritance

```
new_manager.calculate_tax
```

we immediately wish to find the definition of `calculate_tax` to ensure that this is also correct. The logical starting point is to inspect the `MANAGER` class. But as the feature `calculate_tax` is common to both `SALESPERSON` and `MANAGER` it is actually defined in `EMPLOYEE` and inherited. Finding the definition then involves traversing the hierarchy examining each inherited class. In deeper hierarchies with many inherited classes the definition may take some time to locate.

A similar problem is encountered with the definition of `award_bonus` in class `MANAGER`. Within this definition there are three references which may need to be followed. These can be defined anywhere within the `MANAGER` class or its inherited class(es). As these paths are followed, we move further and further away from the

```
class parent1 {
public:
    parent1() {...};
    void function1() {...};
};

class parent2 {
public:
    parent2() {...};
    int function2(int z) {...};
};

class child : parent1, parent2 {
public:
    child() {x = 0};
    void call_me(int y) {
        ...
        x = function2(y);
        ...
    }
private:
    int x;
};
```

Figure 3: Example C++ code for multiple inheritance

original code. As we get further away from the original code, it becomes more difficult to remember the context in which we were searching in this direction. The effectiveness of the inspection may then start to decrease.

### 3.2.2 Multiple Inheritance

Given the C++ code in Figure 3, which implements a simple example of multiple inheritance, and assuming a declared instance `MyChild` of class `child`, then on finding the call

```
MyChild.function1;
```

we must find the definition of `function1`. We start by examining the `child` class, but there is no reference to `function1`. We then assume that it is declared in a parent class. But which one? And how far up the hierarchy must we look? Four or five levels of inheritance is not uncommon. This can mean the exploration of up to ten classes, assuming two parent classes. If there are more than two parent classes, then the number is proportionately bigger. If the *parent* classes use multiple inheritance, then the number of possibilities is further increased. A similar situation occurs with

```
MyChild.call_me(2);
```

Although we initially know that `call_me` is declared somewhere in `child` or its parents, there is now another level of indirection: where is `function2` declared? The possibilities include `child`, `parent1` and `parent2`, along with any of their ancestors. Note that we assume C++ is being used in the strict object-oriented sense. Instead, it may be that there are functions declared which are not part of any class, with `function2` being one of them. In this case, the declaration could exist practically anywhere.

### 3.2.3 Further Issues Concerning Inheritance

In addition to single and multiple inheritance, as described above, there are concepts which can complicate the situation. Also, there are concepts which can ease the task of inspection. We start by describing some Eiffel characteristics allowing the features inherited from the base class to be adapted in several ways. We also describe several idiosyncrasies from C++.

Repeated inheritance is an Eiffel concept which can cause confusion. It is generally used when we wish to reimplement an inherited feature by adding some behaviour in addition to that which it already has. We could simply inherit it, then define another feature which calls the first and then implements the required behaviour, but this means the new feature must have a different name. If we inherit it, then redefine it to keep the same name, we no longer have access to the original feature, and hence must repeat the code in the new feature. Repeated inheritance allows us to inherit the feature once to redefine it (allowing us to use the name) and then inherit it a second time to rename it (which gives us access to the feature, albeit under a different name). We then simply define our new feature using the old name, and use the new name to access the old code. As can be seen from this description, repeated inheritance is by no means a trivial concept. When inspecting code which uses repeated inheritance, it can be very easy to lose track of which code is actually being used. An inspector may miss the repeated inheritance and assume the code being called is simply that in the base class, missing out the additional code in the derived class. If the inspector does spot the repeated inheritance, he has to find the original feature in the inherited class, inspect it, then inspect the new code in the derived class.

Also in Eiffel, the use of `rename` allows a feature in the base class to be renamed when it is inherited in the derived class. Finding the feature involves searching the derived class for the feature definition, or for a `rename` clause. If the feature has been renamed then the clause gives an idea of which route through the parent classes should be followed, but gives no indication of how far up the class hierarchy one must travel.

The Eiffel `redefine` clause can remove some problems. If a feature is redefined for this class, then the definition must occur in this class, and can be found easily. By its nature, however, redefinition is almost always bound to use features declared towards the top of the class hierarchy. To inspect properly, these definitions must also be found. In a similar vein, the `undefine` keyword removes the definition of the corresponding feature for this class, leaving it in a deferred state. The definition of the feature must now occur either in this class, or any class which may inherit from it, reducing the amount of which must be inspected for the definition. Finally, the join mechanism that exists in Eiffel consolidates several inherited methods, possibly via multiple inheritance, from two different classes. This consolidation provides an anchor point from which the definition of the method can be searched for. Again, this can reduce the number of classes which may have to be inspected.

The `friend` keyword is a controversial C++ feature. A function declared to be a `friend` of a class will have unrestricted access to all private data of that class, thus completely circumventing the usual encapsulation mechanism. While there is an argument in using `friend` for efficiency reasons, or to allow the usual implicit parameter of the function to be made explicit, thus allowing coercion, `friend` can hinder inspection by removing much of the logical structure of the system.

A redeeming aspect of C++ is the control of feature visibility provided by the

```
class A {
private:
   int a;
};

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

Figure 4: An example of visibility restriction in C++.

keywords `public`, `protected` and `private`. These keywords limit the access of both non-member functions and derived classes to class features. For example, given the code in Figure 4, the method `b` declared in class `B` makes an illegal reference to the variable `a` declared in class `A`. This reference is illegal because `a` has been declared as `private` in `A` and is not available to any other classes. Similar access rules can apply when inheriting base classes. By restricting access to features, we improve the encapsulation of the class and reduce the possible interactions between that class and any other classes. Intelligent interpretation of such access restrictions can therefore be used to limit the amount of code which has to be inspected. However, as class features may be declared as `public`, `protected` or `private` and classes may then be inherited as `public`, `protected` or `private`, the visibility rules involved are fairly complex. It may be difficult for inspectors to decipher exactly what interface is presented by each class when they inspect code to ensure that the interfaces presented are the minimum required and nothing more. Feature visibility can also be restricted in Eiffel, where a feature can be made visible only to a certain class clientele, defined by the parameters given after the `feature` keyword. Similar issues arise here.

## 3.3 Polymorphism and Dynamic Binding

Polymorphism is the ability to take more than one form. In object-oriented programming, it generally denotes the ability of a reference to refer to more than one class

```
class X
feature
   process is
      do
         put_string("In class X.")
      end
end -- X
```

```
class Y
inherit
     X redefine process
end
feature
   process is
     do
        put_string("In class Y.")
     end
end -- Y
```

```
class Test
feature
   make is
      local
         local_x : X
         local_y : Y
      do
         !!local_x.create   -- local_x refers to an object of class X
         !!local_y.create   -- local_y refers to an object of class Y
         local_x.process    -- X.process is called
         local_x:=local_y -- local_x now refers to an object of class Y
         local_x.process    -- Y.process is called
      end
 end -- Test
```

Figure 5: An example of polymorphism and dynamic binding.

of object. Polymorphism goes hand in hand with dynamic binding, which allows the function associated with such a reference to be inferred at run time. This contrasts with static binding, where the exact function call is known at compile time and can never be changed while the program is executing.

The concepts of polymorphism and dynamic binding can best be described by example. Consider the code in Figure 5 (based on Figure 5 from [15]). `local_x` is initially created to be of class `X`, whilst `local_y` is of class `Y`. On the first call of `local_x.process`, In class X. is printed. Although, on the surface, `local_x` and `local_y` appear to be instances of completely different classes, the assignment statement `local_x:=local_y` is allowed because class `Y` is derived from class `X`, therefore an instance of class `Y` already contains all the details of an instance of class `X`, and may also contain much more. `Y` is said to *conform* to `X`. This type of behaviour is the essence of polymorphism. Note that `local_y:=local_x` is not legal, since an object of class `X` may not have enough information to completely fill an instance of `Y`. Since `local_x` now refers to an object of class `Y`, the next time `local_x.process` is called, IN CLASS Y. is printed. This is dynamic binding at work: the actual `process` feature called depends on the dynamic type of `local_x`, not its static type.

Further polymorphism can occur with parameter passing. If we define a feature to take a parameter of class `X`, then in addition to an instance of class `X`, we can also pass an instance of class `Y`, or indeed any derived class of `X`. This can cause difficulties in C++ ,where multiple methods can be declared which differ only in their parameter lists, both in number of parameters and parameter types. It then becomes more difficult to predict which method is called at run-time, taking into account any coercion of parameters which may take place.

The concept of polymorphism is very powerful, but this power comes with a price. Like recursion, polymorphism is simple in theory, but hard in practice. Ponder and Bush have written about the problems that polymorphism causes for program understanding due to dependence on the dynamic data state of the program [21]. The problem is especially acute when combined with inheritance. Ponder and Bush show data from a Smalltalk-80 system, where, for example one method is defined 90 times, and where there are 89 procedures which are received by 486 different types. This leads to “considerable ambiguity,” an understatement of immense proportions. Consider the second call of `local_x.process` in Figure 5. When inspectors reach this call they must know the current type of `local_x`. It is simple to find in this example, but in larger systems, the type assignment may be much further away. It is also possible for confusion to occur when multiple type changes occur.

A specific case occurs with a popular demonstration of polymorphism: iterating over an array of heterogeneous types. Consider the following Eiffel declaration:

```
my_array : ARRAY[X]
```

In this case, elements in the array `my_array` may be of class `X`, `Y` or any other class derived from `X`. To call the same feature `process` for every element of the array, we simply write:

```
my_array.item(i).process
```

for each value of `i`. However, when we inspect this code, we must check every possible definition of `process` that may be used. It can be difficult to find all implementations of the feature, due mainly to the distribution definitions over several classes. More importantly, we must ensure that every implementation has consistent behaviour, producing the same 'result', however 'result' is defined. Things may be further complicated if different implementations deliberately have slightly different behaviours. If inspectors believe they understand one implementation, they may incorrectly assume all implementations produce the same behaviour. This may occur when naming conventions are not adhered to.

Polymorphism used in feature parameters causes similar problems. Any features of the parameter that are used must be checked for consistent behaviour, since the implementation which is called at run time may not be known. If one implementation has slightly different behaviour from the others, it may introduce subtle bugs. We must find exactly which implementation is being used.

## 3.4 Genericity

Genericity is the ability to define classes that are parameterised with respect to type. They are usually used to define container classes which can then be used to hold any type of data. Some common examples include lists, hash tables and trees. Without genericity, a new class would have to be written every time we wished to store a new type of data. We would then have multiple classes defining exactly the same behaviour, which would cause maintenance and administrative difficulties. Genericity also allows the use of static type-checking.

Genericity in C++ is implemented using templates. Class templates are used to define a related family of classes. The class is defined with one or more type parameters which can then be used as normal types within the class definition. When the template is instantiated with the appropriate type, all instances of the argument are replaced by the new type to produce a new class. Similarly, function templates can be used to define a related family of functions, defining similar operations on multiple types. An instantiated version of the function is created for each type which uses it. Both class and function templates can have multiple type arguments.

In Eiffel, genericity is achieved with generic classes which take a list of type parameters. Eiffel defines two types of genericity. Unconstrained genericity is simply when the type which can be supplied is not limited in any way. In this case, there is little common behaviour which we can rely on having access to. Therefore, it is used only for the simplest generic classes. Much more useful is constrained genericity, where the parameters are constrained to be derived from a certain class. This constraining class specifies the minimum behaviour which we can expect from any class which may instantiate the generic class. In fact, unconstrained genericity is simply constrained genericity where the constraining class is class `ANY` (every non-basic class which has no explicit inheritance clause inherits from class `ANY`, hence every class conforms to class `ANY`). In this situation, it is usual to leave out the constraining class altogether.

An example of constrained genericity is given in Figure 6. Here, we define class `SEARCHTREE`, which can be instantiated by any class, as long as it conforms to class `COMPARABLE`. This constraint is necessary because we require use of a comparison operator to insert new elements into the tree. However, there is a problem for inspection in relying on the implementation of such behaviour. In this case, `COMPARABLE` is a well-defined class and the meaning of the less-than operator is well-known. However, it may be the case that the constraining class is less well-defined, with far more complex operations. Derived classes may have redefined key behaviour in an inconsistent way. Therefore, what works for one instantiation may produce slightly different behaviour for another instantiation. This implies that a generic class must be inspected with respect to each instantiating class. This can be problematic with respect to the number of possible classes, and the possibility of new classes being added as time goes on.

```
class NODE[G]

creation make_leaf

feature{SEARCHTREE}
   value : G
   left, right : NODE[G]

   make_leaf(avalue : G; leftchild : NODE[G]; rightchild : NODE[G]) is
        do
             value:=avalue
             left:=leftchild
             right:=rightchild
        end
end -- class NODE

class SEARCHTREE[G -> COMPARABLE]

feature {SEARCHTREE}

   root : NODE[G]

   insert_into(element : G; current : NODE[G]) is
        do
             if current = void then
                !!current.make_leaf(element)
             else
                if element < curr.value then
                   insert_into(element, current.left)
                else
                   insert_into(element, current.right)
                end
             end
          end

   feature
      insert(element : G) is
         do
              insert_into(element, root)
         end
   end -- class SEARCHTREE
```

Figure 6: An example of genericity in Eiffel.

## 3.5 Partitioning Object-Oriented Code for Inspection

The previous sections have assumed the entire system is capable of being inspected at once. The inspector browses the code until he becomes familiar with it and understands its structure, and can then start to examine the code for defects, perhaps aided by a checklist. In reality, most systems will be far too complex to be inspected in a single step, and will be split into chunks. The amount of code inspected is also limited by the two-hour rule: an inspector should not spend more than two hours at a time on individual preparation, and an inspection meeting should not last more than two hours. There is a general belief that the effectiveness of any inspection is greatly reduced when such limits are exceeded. Finally, there may be guidelines in place on the rate at which code should be inspected, which may be as low as one or two pages an hour. Again, exceeding these limits may decrease the effectiveness of the inspection. These three factors produce the problem of deciding how to split the code. For a simple object-based system, this problem is no worse than for modular procedural code.

For a system with a large inheritance hierarchy, the problem is much more difficult. As demonstrated in the previous sections, there are many dependencies which must be resolved when inspecting object-oriented code. If the system is arbitrarily split, then inspectors may be left with references to code which they have no access to, preventing them from properly completing the inspection. When inheritance is involved there is a problem similar to that found in testing, where 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. The same is true of inspection. Each class must be inspected in the context of any parent classes. There may be many such parent classes, which combine to produce a very large body of code to be inspected. On the other hand, a single method may be too small a unit to inspect, even before considering the number of references that may be left unresolved by inspecting the method on its own. A one- or two-line method has very little semantic information to allow an accurate characterisation of the behaviour of the system.

# 4 Assisting Inspection of Object-Oriented Software

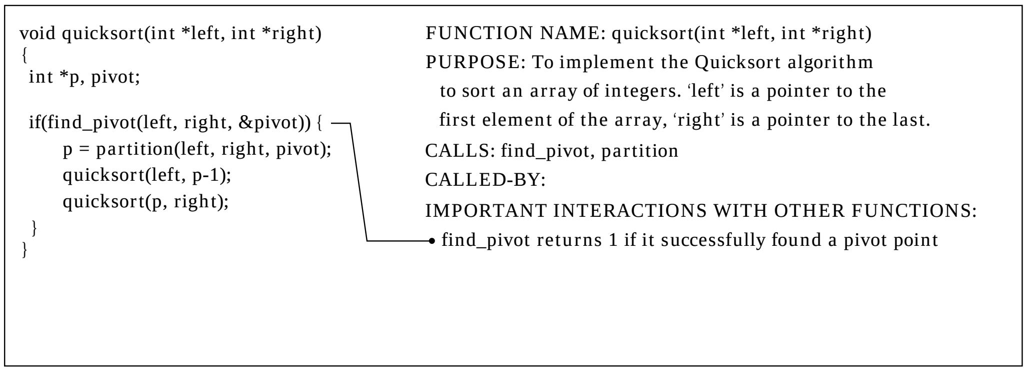
## 4.1 Programming Techniques

From the previously described problems caused by inheritance, it is apparent that there is a need to control the use of inheritance. Korson and McGregor point this out, and suggest a "very disciplined" use of inheritance [15]. Gamma *et al.*, in their book on patterns for object-oriented design [10], suggest one principle of object-oriented design is:

*Favour object composition over class inheritance*.

They argue that the use of object composition allows you to keep each class encapsulated and dedicated to one task. It also inhibits the growth of class hierarchies into "unmanageable monsters." Such hierarchies will be very difficult to inspect, as demonstrated in Section 3.2. A guideline for programming is the Law of Demeter, described above, which limits message passing to a reduced set of related objects. By rationalising communication patterns between objects, the system may be easier to understand and inspect [16].

Van Emden [25] describes a method of inspection called structured inspection, which make use of what he refers to as the *inspection protocol*. The inspection protocol defines what is to be carried out at the inspection. In a traditional inspection, as described by Fagan [8], the protocol may only consist of a checklist to assist in defect finding. For a structured inspection, the code to be inspected is written with the goal of inspectability in mind. Comments are placed in the code to guide the inspection. The comments are assertions about the state of the program at that point, with each being a well-defined inspection item. The inspection then consists of examining each of these items in turn, checking that the code between two assertions ensures the truth of the latter assertion. This idea may greatly increase the focus of the inspection, as well as breaking the code into more manageable chunks for inspection, thus alleviating one problem identified. The assertion checking system provided in Eiffel would seem to be an ideal vehicle to make use of such an idea.



```
void quicksort(int *left, int *right)
{
 int *p, pivot;

 if(find_pivot(left, right, &pivot)) {
      p = partition(left, right, pivot);
      quicksort(left, p-1);
      quicksort(p, right);
 }
}
```

```
FUNCTION NAME: quicksort(int *left, int *right)
PURPOSE: To implement the Quicksort algorithm
  to sort an array of integers. 'left' is a pointer to the
  first element of the array, 'right' is a pointer to the last.
CALLS: find_pivot, partition
CALLED-BY:
IMPORTANT INTERACTIONS WITH OTHER FUNCTIONS:
• find_pivot returns 1 if it successfully found a pivot point
```

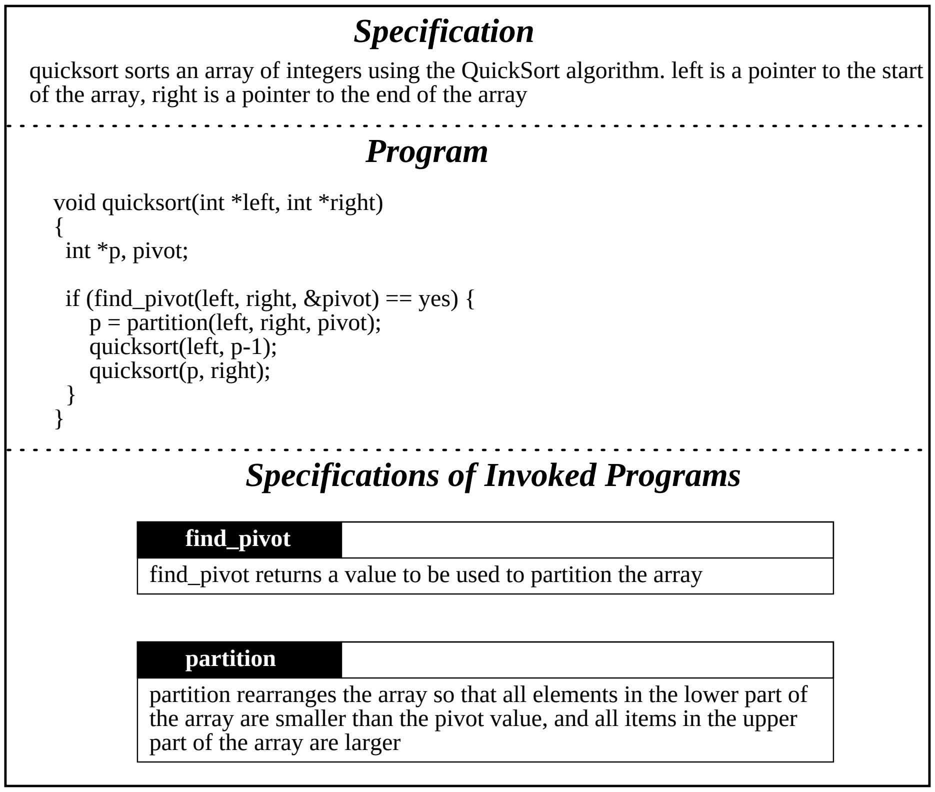
Figure 7: An example of Soloway *et al.*'s program documentation method.

Eiffel assertions also provide an opportunity to ensure that all implementations of a polymorphic method have the same behaviour. This can reduce confusion where more than one implementation exists, but where each implementation has a slightly different behaviour.

## 4.2 Supporting Documentation

With specific reference to delocalised plans, Soloway *et al*. [23] present a type of program documentation which explicitly identifies relationships which form delocalised plans. The documentation consists of the program text on the left side of a page. The right hand side contains explicit descriptions of interactions which form delocalised plans. For example, any subroutine called in the program fragment would be documented in the accompanying text, indicating its purpose, and any routines which it may call. There may also be comments which describe the interactions occurring in the code. These are linked by arrows to the appropriate line. An example of the documentation is shown in Figure 7, using code based on that given for an implementation of Quicksort in [14]. While this idea may be useful to make explicit the dependencies within object-oriented code (for example "This method relies on method X in the parent class"), there is no obvious support for describing dynamic behaviour. The technique fails due to its static nature.

A similar, but far more refined method is proposed by Parnas *et al*. [20]. They



***Specification***

quicksort sorts an array of integers using the QuickSort algorithm. left is a pointer to the start of the array, right is a pointer to the end of the array

***Program***

```
void quicksort(int *left, int *right)
{
  int *p, pivot;

  if (find_pivot(left, right, &pivot) == yes) {
      p = partition(left, right, pivot);
      quicksort(left, p-1);
      quicksort(p, right);
  }
}
```

***Specifications of Invoked Programs***

**find_pivot**

find_pivot returns a value to be used to partition the array

**partition**

partition rearranges the array so that all elements in the lower part of the array are smaller than the pivot value, and all items in the upper part of the array are larger

Figure 8: An example of Parnas *et al.*'s Display Method.

assert that successful software systems will have more people reading the code than writing it. Furthermore, while it is easy to understand isolated details of the program, it is far harder to make sense of the overall structure. It is therefore important that a system is well documented, with the structure being made apparent. Parnas *et al.* propose a solution with their “Display Method”. In this method, each fragment of the program is represented by a display. The set of displays associated with a program is supplemented by a lexicon, containing definitions of all the terms in the program, and an index of all the variables and program fragments along with where they occur. An example of a display is given in Figure 8. A display consists of three parts. The first parts presents a specification of the program fragment this display represents. The second part contains the source code of the fragment under consideration. The final part presents specifications for any subroutines invoked by this fragment. It is therefore possible to study the correctness of this program fragment without requiring access to any other code which it may use.

While the display method is intended for use with procedural code, where Parnas *et al.* [20] have reported great success in inspecting safety critical software, the principles can be applied to object-oriented code. The proliferation of small methods provides the ideal subject for this technique, although it may be that the average one- or two-line method may be too small a unit to use. Such a method usually only consists of a single method invocation, and has little semantic information, whichever way it is described. Being designed for use with procedural code, there is also no policy to deal with inheritance. Furthermore, like Soloway's method, the technique is static, and is therefore of limited use in describing the dynamic behaviour of object-oriented code which is necessary to understand the system.

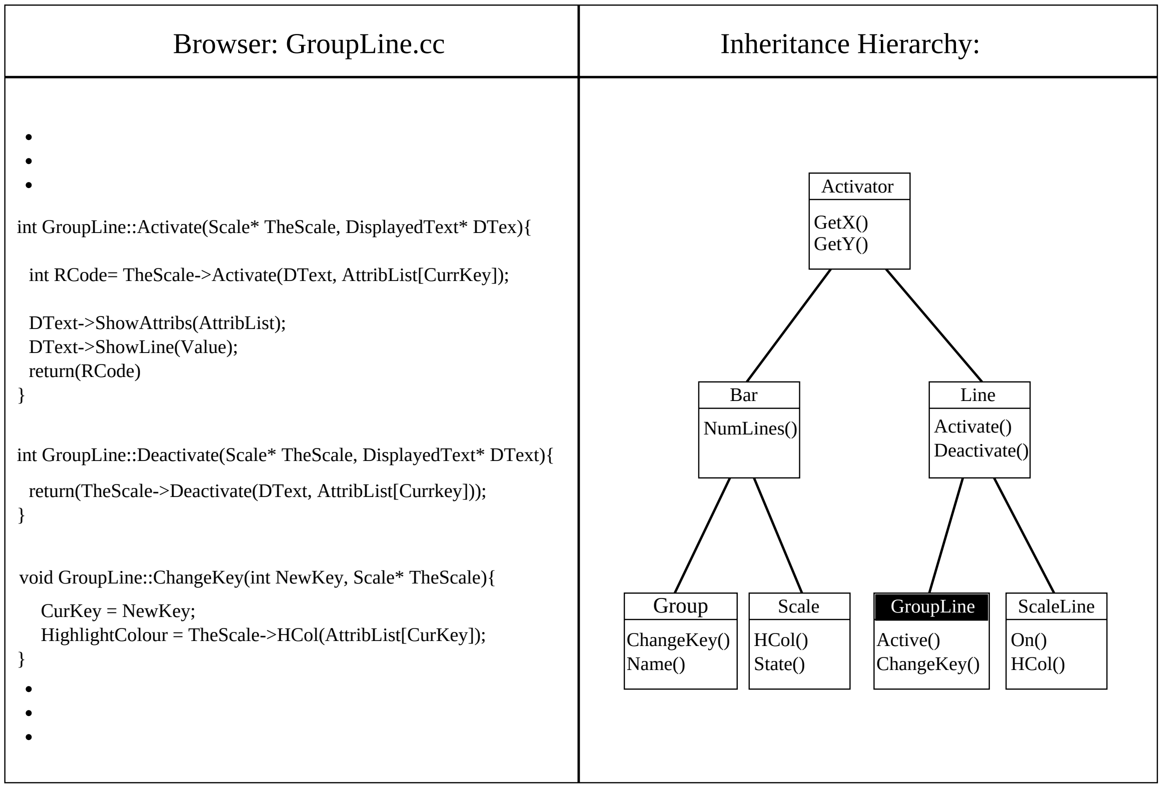
Although enhanced documentation can help an inspector, it is clear that the static nature precludes the description of the dynamic behaviour of the system, which, as we have seen, is far more important in an object-oriented system. Therefore it may be that some form of tool support, possibly working in conjunction with the running system, may be more appropriate.

## 4.3 Tool Support

Although there are already a number of tools to support the inspection process (see [18] for a recent review of those available), none explicitly support the dynamic nature of object-oriented code. Instead, the code is treated as a static document. From Section 3 it is clear that this is not sufficient.

An automated tool for viewing code statically, based on the work of Parnas *et al.* [20] described in the previous section, is one possibility. Each display could be presented on-screen, with hypertext-style links between the displays, allowing the inspector to browse round the system. However, this still only represents the static structure of the code. We still require some form of dynamic inspection tool to help inspect code which makes use of dynamic object-oriented features.

A possibility lies in tools designed to support maintenance. Section 2 introduced



```
•
•
•

int GroupLine::Activate(Scale* TheScale, DisplayedText* DTex){

  int RCode= TheScale->Activate(DText, AttribList[CurrKey]);

  DText->ShowAttribs(AttribList);
  DText->ShowLine(Value);
  return(RCode)
}

int GroupLine::Deactivate(Scale* TheScale, DisplayedText* DText){

  return(TheScale->Deactivate(DText, AttribList[Currkey]));
}

void GroupLine::ChangeKey(int NewKey, Scale* TheScale){

    CurKey = NewKey;
    HighlightColour = TheScale->HCol(AttribList[CurKey]);
}
•
•
•
```

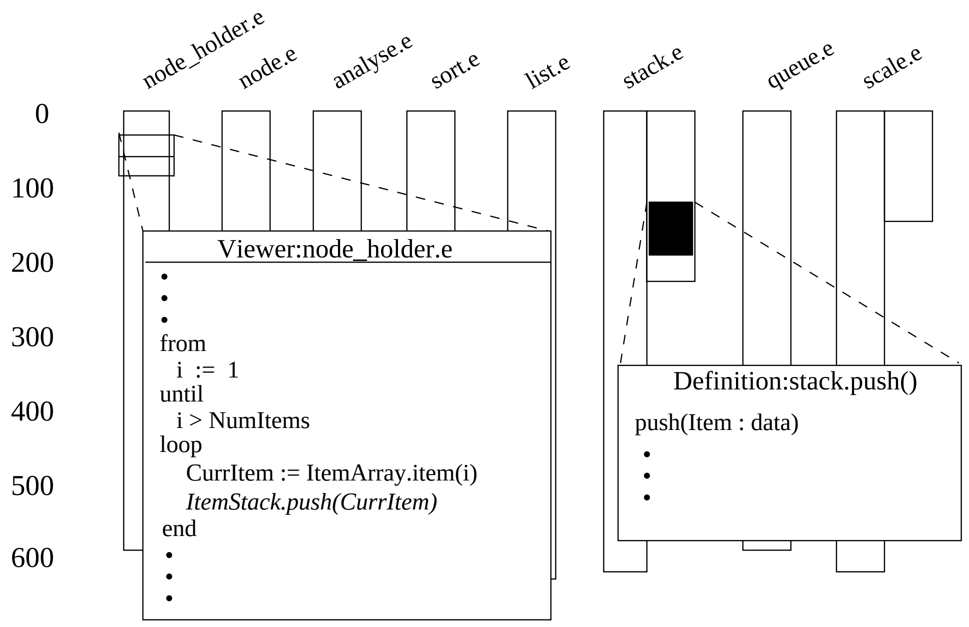
Figure 9: An example maintenance tool set.

the idea that the requirements of inspection are similar to those of maintenance. It is therefore obvious that tools which help understanding for maintenance could also be used to support inspection. One example is Valhalla, a prototype object-oriented development environment described by Wilde *et al*. [27]. This system provides object animation capabilities for both the development and maintenance phases. The animation allows viewing of messages passed between objects. This can aid understanding of the dynamic properties of the system. Instead of analysing static pages of text, the inspection would then consist of analysing these animations.

Another example of such support for maintenance is described by Crocker and Mayrhauser [3]. They describe a suite containing four types of tool. Framework tools are used to provide an infrastructure, and include a database which allows the sharing of information among tools. Mundane tools are those used for information gathering, including a cross-reference generator, control flow graph generator and test driver generator. Change tools are used to predict the effects of program modification. They include a consistency checker and ripple effect analyser. From our point of view, the most useful tools are those described as knowledge tools, which can assist in program understanding. The *inheritance hierarchy generator* generates a graph of the inheritance relationship in the system, which can then be studied to enhance understanding of the system. The *abstraction generator* is used to help build new abstractions, and add and delete methods. The abstraction generator takes a set of input functions and generates a subset of functions which are related by common data. A subset of these can then be used as the interface for a new class. A similar analysis can be used to decide which data items are candidates for membership of the class. Finally, the remaining code is analysed to indicate changes required to use the new class in place of the existing code, for example removing references to encapsulated data. The analysis for the addition and removal of methods is similar. The final tool described is a *code browser* which displays the output produced by the other tools. This is used in conjunction with a *code slicer*, which allows the view of the program to be limited by certain criteria, such as occurrences of a certain variable or method call. An example of how such a toolset may appear is given in Figure 9. The class hierarchy is displayed on the right-hand side, with the text of the current class displayed on the left-hand side. The displayed code may be a program slice on a variable usage or function call. It is then easy for the code inspector to traverse the class hierarchy and inspect the required code.

These maintenance systems would suffer from one drawback if they were to be applied to software inspection: they require the software system to be complete and running. They are not designed to work with fragments of code, such as individual classes. Browsing using the class hierarchy may also be limiting because it does not reflect the dynamic nature of the code.

Given that a major problem in inspecting object-oriented code is tracing method calls and references over several classes, it may be useful to have some form of reduced representation which provides an overall view of the code being inspected. Such a representation would be similar to that used be Seesoft, as described by Eick *et al.* [7]. Seesoft is a tool designed for visualising line-oriented software statistics. The main window consists of a number of columns, each of which represents a source code file.



```
•
•
•
from
  i := 1
until
  i > NumItems
loop
   CurrItem := ItemArray.item(i)
   ItemStack.push(CurrItem)
end
•
•
•
```

```
push(Item : data)
•
•
•
```

Figure 10: How a reduced representation could be used to assist inspection of object-oriented code

Within these columns, a horizontal line is used to represent a line of code within the file. These lines are coloured according to the value of some attribute, e.g. age. A separate scale is used to display the entire value range for this attribute. The user can click on values in this scale, or the columns and lines themselves to toggle each value on and off. This allows the display of code with just a certain value or a range of values. Such manipulations allow the user to find useful patterns in the code, in effect allowing interactive querying of a database of statistics. Seesoft also provides code reading windows which allow the user to access the source code under consideration.

This type of tool could be extended to assist inspection of object-oriented code as follows. While inspecting code in a reading window the reduced representation would highlight the current line of code. If this line was a method invocation, the definition of that method would also be highlighted. The inspector could then immediately move to that definition, and so on. The history of such a progression may be stored and when the inspector comes to a suitable understanding of some method, it would be possible to quickly backtrack to the previous method, where this understanding could be applied.

This process would continue until the original starting point was reached. By speeding the traversal between methods, it is easier for an inspector to gain an understanding of the code. At the same time, the inspector is forming a mental picture of the system with the reduced representation. A further use of such a system would be used to help decide which code should be included in an inspection. By tracing method invocations, the classes required to perform the inspection could quickly be found. An example of how this may look is given in Figure 10. The window contains reduced representations of all files under inspection. The file currently being accessed (`node_holder.e`) has a cursor over it, indicating the current line being inspected, as determined by the browser window. The current line is shown in italics, and in this case is a call of method `push` in class `stack`. The definition of this method has been found by the tool in file `stack.e`, as indicated by the inverse patch on this file, and the definition is displayed in the method window.

Other visualisation systems designed for object-oriented code may provide help with inspection. For example, De Pauw *et al*. [4] describe a language independent visualisation system. The system uses a preprocessor to instrument the subject programs with code which generates events. These events can be received by a visualisation application which can use the data to update one or more views of the program. The authors describe several visualisations they have constructed. They include an allocation matrix, showing the number of classes instantiated by each class, and an inter-class call matrix, showing patterns of communication between each class. Although these visualisations are intended for use in debugging and code tuning, and rely on having the entire system available and running, the principles could be applied to smaller chunks of code. Visualisation could allow the inspection team to provide summary information on which methods and classes are used by each class. They can then use this information to partition code for inspection.

## 5 Conclusions

Inspection is widely believed to be the most effective means of finding defects in software. At the same time, the object-oriented paradigm is cited as providing many benefits in developing software. However, there is little published material on applying inspection to object-oriented software.

We have described how some of the facilities of an object-oriented language can inhibit inspection of code written in that language. Inheritance can impede the search for the definition of class features. Polymorphism and dynamic binding combine to hinder the static prediction of which methods will be invoked at runtime. Genericity can prove problematic with respect to the number of classes that may have to be inspected in conjunction with a single class, due to the dependence on the behaviour of these instantiating classes. The last two problems stem from the disparity between the static code structure and the dynamic, runtime system structure. Furthermore, object-oriented systems tend to consist of a large number of small methods, which distributes functionally related code over a wider area than procedural systems, making inspection more difficult. This also increases the number of relationships which exist within the system which have to be understood. Finally, while inspection is an ideal time to enforce code quality, the notions of quality of object-oriented code are less well-defined than those of procedural code, and may be difficult to enforce during inspection.

We have also described some techniques which may be applied to inspection of object-oriented software to help obviate these problems. There are several programming techniques which make code easier to inspect. Restricting the use of inheritance reduces the complexity of the class hierarchy [10], whilst the Law of Demeter can reduce the amount of unstructured communication between objects [16]. Code can also be written with inspectibility in mind, by placing assertions about the state of the program as comments in the code [25]. Eiffel assertions may be used for this purpose. In addition they may be used to ensure all implementations of a polymorphic method have the same behaviour. There are also several methods of documenting code to assist in inspection. Soloway *et al*. [23] describe a documentation method for making delocalised plans in the source code explicit. Parnas *et al*. [20] present a documentation system called the Display Method which allows portions of source code to be inspected in isolation using the specifications of any other program fragments that may be called. Finally, we described how tool support could assist with object-oriented code, especially with the more dynamic properties. An automated version of Parnas *et al*.'s Display Method may be one possibility, whilst another may be in the use of tools used to support maintenance, such as the Valhalla system [27] or the maintenance suite described by Crocker and Mayrhauser [3]. Finally, visualisation tools may be of some help, such as the static Seesoft system [7] or the dynamic visualisation system described by De Pauw *et al*. [4]. We believe that this type of support is essential if inspection is to be successfully used with the object-oriented paradigm. Lack of such support will reduce the benefits of our most effective defect finding process.

## References

[1] E. V. Berard. *Essays on Object-Oriented Software Engineering Volume 1*. Prentice-Hall, 1993.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications* (2ed.). Benjamin/Cummings, 1994.

[3] R. T. Crocker and A. Mayrhauser. “Maintenance Support Needs for Object-Oriented Software,” *Proceedings of COMPSAC ’93*, pp. 63–69.

[4] W. De Pauw, R. Helm, D. Kimelman and J. Vlissides. “Visualizing the Behaviour of Object-Oriented Systems,” *Proceedings of OOPSLA ’93*, pp. 326-337.

[5] A. M. Davis. “Fifteen Principles of Software Engineering,” *IEEE Software*, Vol. 11, No. 6, November 1994, pp. 94–101.

[6] E. P. Doolan. “Experience with Fagan’s Inspection Method,” *Software - Practice and Experience*, Vol. 22, No. 2, February 1992, pp. 173–182.

[7] S. G. Eick, J. L. Steffen, E. E. Sumner Jr. “Seesoft - A Tool For Visualizing Line Oriented Software Statistics,” *IEEE Transactions on Software Engineering*, Vol. SE-18, No. 11, November 1992, pp. 957–968.

[8] M. E. Fagan. “Design and Code Inspections to Reduce Errors in Program Development,” *IBM System Journal*, Vol. 15, No. 3, 1976, pp. 182–211.

[9] M. E. Fagan. “Advances in Software Inspection,” *IEEE Transactions on Software Engineering*, Vol. SE-12, No. 7, July 1986, pp. 744–751.

[10] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[11] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley, 1993.

[12] C. Jones. “Gaps in the Object-Oriented Paradigm,” *IEEE Computer*, Vol. 27, No. 6, June 1994, pp. 90–91.

[13] P. Jüttner, S. Kolb and P. Zimmerer. "Integration and Testing of Object-Oriented Software," Proceedings of EuroSTAR '94.

[14] A. Kelley and Ira Pohl. *A Book on C*. Benjamin/Cummings, 1990.

[15] 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.

[16] K. J. Lieberherr and I. M. Holland. "Assuring Good Style for Object-Oriented Programs," *IEEE Software*, Vol. 6, No. 5, September 1989, pp. 38–48.

[17] S. Letovsky and E. Soloway. "Delocalized Plans and Program Comprehension," *IEEE Software*, Vol. 3, No. 3, May 1986, pp. 41–49.

[18] F. Macdonald, J. Miller, A. Brooks, M. Roper, M. Wood. "A Review of Tool Support for Software Inspection," In *Proceedings of CASE '95 International Workshop on Tools and Technologies* (to be published).

[19] B. Meyer. *Eiffel: The Language*. Prentice Hall International, 1992.

[20] D. L. Parnas, J. Madey and M. Iglewski. "Precise Documentation of Well-Structured Programs," *IEEE Transactions on Software Engineering*, Vol. SE-20, No. 12, December 1994, pp. 948–976.

[21] C. Ponder and W. Bush. "Polymorphism Considered Harmful," *ACM SIGSOFT Software Engineering Notes*, Vol. 19, No. 2, April 1994, pp. 35–37.

[22] G. W. Russell. "Experience with Inspections in Ultralarge-Scale Development," *IEEE Software*, Vol. 8, No. 1, January 1991, pp. 25–31.

[23] E. Soloway, J. Pinto, S. Letovsky, D. Littman and R. Lampert. "Designing Documentation to Compensate for Delocalized Plans," *Communications of the ACM*, Vol. 31, No. 11, November 1988, pp. 1259–1267.

[24] B. Stroustrup. *The C++ Programming Language* (2ed.). Addison-Wesley, 1991.

[25] M. H. Van Emden. “Structured Inspections of Code,” *Software Testing, Verification and Reliability*, Vol. 2, 1992, pp. 133–153.

[26] N. Wilde and R. Huitt. “Maintenance Support for Object-Oriented Programs,” *IEEE Transactions on Software Engineering*, Vol. SE-18, No. 12, December 1992, pp. 1038–1044.

[27] N. Wilde, P. Matthews and R. Huitt. “Maintaining Object-Oriented Software,” *IEEE Software*, Vol. 10, No. 1, January 1993, pp. 75–80.