Abstract

C++ Lens is an innovative visualisation technique and associated tool, that assists maintainers comprehend C++ source code. The primary role of the C++ Lens technique and tool is to visualise the interface relationships existing between C++ classes and to signify the relative importance of these relationships. The technique innovatively combines, via the definition of lenses, elements of reverse engineering, code browsing, visualisation, and metrics gathering into a coherent framework for code comprehension. This paper describes the basis of the technique and presents some results of applying the C++ Lens technique and tool to large-scale commercial software.

Keywords: Large-Scale Commercial Software, Software Tools, Code Visualisation, Software Maintenance, Reverse Engineering.

1. Introduction

Program comprehension is one of the most frequent and essential, yet difficult and costly, software engineering activities to undertake [7, 15, 17]. It is required each time a program is corrected, reused, migrated, reengineered, inspected or enhanced and involves three key activities [19]:

- Understanding what the system does and how it relates it its environment.
- Identifying where the system change should be effected.
- Deducing how the components identified for correction, modification or reuse work.

A person is said to understand a program when they are able to explain the program, its structure, its behaviour, its effects on its operational context and its relationships to its environment in human-oriented terms, rather than in the terminology and syntax of its program representation. That is, a natural language understanding is reached regarding the association between the unfamiliar part of the program and the formal representation of the code. This is known as the concept assignment problem [3].

The complexities of comprehension may be attributable to a number of factors such as the complexity and size of the code base, tightly coupled interfaces, poor documentation and little knowledge of design intent. These factors combine together, to make estimations of project duration and cost very difficult for software developers and maintainers. They also heighten the risk of new errors being introduced and propagated through the system as a result of changes being made.

In addition, program comprehension to date has been a very people-oriented activity, requiring ever increasing numbers of professional software practitioners. This growth in systems development, allied to increased software maintenance needs has led to an estimated shortfall of some 300,000 software professionals [22]. The problem is also exacerbated by staff turnover as evidenced by a survey of UK IT professionals, which found that the average annual employee resignation-rate was over ten percent [16]. It is therefore likely that the programmer making the change is not the person that designed or initially implemented the system. This again increases the risk of errors being introduced into the program during the change process [10] since the maintainer must ‘learn’ the code in order to understand, identify and deduce its components and characteristics.

With such a large number of practitioners and the need to continually recruit new staff, it not surprising to find that there is a substantial difference in both effectiveness and quality of output. This claim is supported by results of two studies, which show that the variance in the ability of professional programmers to deliver software solutions may be in the order of ten [13] to fifty-fold [8]. Companies with large-scale software systems therefore not only have to accept the large variance in programmer productivity, but must consider the training needs of maintenance programmers, such that they can safely make changes to the ever-increasing legacy code base. In reality, the only way to achieve this is to train the staff on-the-job, usually working directly with the source code. This is because, even if documentation exists, the logic of the software system is usually so complex that the source code is the only exact record of what the software does, how it operates [17] and how it is related to its environment.
2. Comprehending Large Scale Object-Oriented Programs

Recently much emphasis has been placed on the development and subsequent maintenance of systems using modern programming languages such as C++, object-oriented techniques and associated design approaches such as patterns [6, 10, 11] all of which are based heavily on the concept of abstraction. From a program comprehension perspective such advances may be considered to be a double-edged sword. On the one hand they promote abstraction of system concepts and as such can be used to separate the major sub-systems of a product and help bridge the conceptual gap between the program and real-world ideas. However, on the other hand they introduce a number of potential problems:

- **Lost abstractions**: abstractions that are meaningful at the design level may become 'lost' when implemented at the code level.

- **Complex structure**: if abstractions are overused, the code may become structured like a giant web [10], often to the extent that a programmer who is unfamiliar with the code may only be able to see a confusing network of interacting classes [17].

- **Restrictive requirements**: whilst design templates or patterns offer a degree of defined structure to the development or maintenance approach, often they require that systems are developed from the outset in this manner. Since, templates and patterns cannot easily be retrospectively applied during the maintenance of legacy system code; this restricts their use in many situations.

- **Tight coupling**: as mentioned above, object-oriented techniques commonly use abstraction to separate the major sub systems of a product. However, the resulting abstractions may be tightly coupled via their interfaces such that if one abstraction is used in many places and that abstraction’s interface is incorrect, then repairing it forces repair of all situations in which it is used, with the risk of causing further undesirable events elsewhere in the code.

Additionally, programming of large software systems raises many complex issues not apparent in smaller code bases [12, 17]. For example, C++ is a mixture of at least four programming paradigms: procedural programming, abstract data types, object-orientation and template programming [18]. Central to the power of the language is the concept of classes which may be implemented as template, polymorphic, abstract or base classes [20]. As such, C++ appears to fit into the design remit for a language that can handle programming-in-the-large (PITL) [4], defined as the process of controlling the interaction between modules or components of a system.

This implies that C++ is a suitable language for large-scale projects. However in practice, excessive compile-time coupling, which is virtually irrelevant for small-scale projects, can dominate the development and maintenance time for larger projects [12]. As a project grows, changes in a header file can demand a complete system rebuild with the effect that changes can become too expensive to implement [5]. Experience has shown a one-line modification of central header files, for a 500 KLOC project, can result in a two hour rebuild time, and for programs with many million-lines of code the rebuild time may extended to days [5]. The result of this is that effort is often concentrated on implementing the code in a manner that minimises rebuild time, rather than in a manner that reflects design intent. Implementation details thus start to obscure design intent and make program comprehension more difficult.

Additionally, the object-oriented principles of encapsulation and abstraction, which enable and encourage classes to be viewed as black-boxes, mean that the interfaces between the black-boxes become more important than the box contents themselves. These interfaces act as key abstractions between and throughout the different translation units. Thus, instead of a programmer reading a single module and understanding a suite of functions, a programmer has to find the key classes within the code and understand the role and effect of making changes to their interfaces across the entire system.

A number of the approaches commonly used for comprehension of C++ legacy source code are listed below, together with identified deficiencies for large scale code bases:

- **Running the software**: can help the programmer understand the user interface. When coupled with white box testing it can also flex the key micro-architectures. However it does not highlight key class interfaces nor does it reveal dormant interfaces that the designer has included for future flexibility.

- **Reading the source code**: can be one of the most difficult ways to comprehend large-scale legacy C++ source code. This is because the programmer has to search across translation units in order to piece together a complex interaction web, often requiring the extensive use of basic string searching tools. Source code comments, if present, can help the comprehension process greatly, however, they can easily become out of date and unreliable.

- **Debugging the source code**: by reading and running the code in combination can greatly enhance the comprehension speed of the programmer. However, source code debuggers do not enter header files (unless the header has in-
Generating graphical class hierarchies from the

Source code browsers: can report on how inheritance is used. However, inheritance is a small (and often over emphasised) part of the interface relationships in use in commercial code C++. Usefully, a browser can also report on where a class is defined and navigate to where an interface is used, thereby saving time over using string-searching tools. Finding where an interface is referenced is very useful for understanding large programs and finding the key abstractions. However, there may be many hundreds or thousands of classes within a large industrial project and as such, it may be difficult to differentiate the classes since browsers are text oriented in their presentation of references and give no indication of scale.

Metrics: gathering tools such as McCabes’s Visual Testing ToolSet™ [24] can give the maintainer an idea of complexity or coverage. However, in the main these metrics are geared towards procedural programming and do not analyse interface interactions. Research is proposing object-oriented metrics such as coupling, cohesion, similarity and volatility [21]. However, tool support to help a programmer gather these type of metrics and to usefully analyse them for maintenance purposes is currently very limited [1, 2].

Generating graphical class hierarchies from the source: by tools such as Rational Rose [25] and SNIFF+ [14, 26] enables partial Universal Modelling Language (UML) diagrams to be reverse engineered from C++ source code. This enables the inheritance hierarchy of a set of classes to be seen. It can also report on simple associations between classes (‘has-a’ relationships rather than ‘is-a’ inheritance). However, the diagrams produced by these tools do not convey the physical dependencies between classes and hence do not communicate the impact of changes made to the key abstractions.

Whilst the above approaches offer some assistance to the comprehension process of legacy C++ code, in isolation, the methods tend to be laborious and error prone particularly for large-scale code bases. There is also an absence of methods or tools that focus comprehension through identification of the key abstractions within the code. Additionally, any in-built flexibility in the design, implemented as dormant classes, remains hidden with the above methods.

Techniques and tools are thus required which are independent of development approach and coding style and which can assist the maintainer find and analyse the key abstractions in a legacy code base such that they can understand the impact of changing them. The remainder of this paper describes the C++ Lens technique and tool and its effectiveness for comprehending and maintaining commercial large-scale source code.

3. C++ Lens Technique and Tool

The primary role of the C++ Lens technique and tool is to visualise the interface relationships existing between C++ classes and to signify the relative importance of these relationships. The technique innovatively combines, via the definition of lenses, the reverse engineering, code browsing, visualisation, and metrics gathering approaches described in the above section into a coherent framework for code comprehension. This is achieved through incorporation of the following elements:

- Reverse engineering: graphical representation of inheritance hierarchies.
- Source code browsers: textual representation of the importance of an abstraction.
- Visualisation: instantaneous visual analysis of data via the concept of fish-eye views. That is, the focus of attention is drawn to key interfaces by sizing them in proportion to their importance. The closer an item is to the focus, the larger it is drawn and the more relevance it has to the underlying issue [9].
- Metrics: a suite of tools under development within the project that reverse engineers C++ code. These tools gather metrics and combine the available
information from the browser database to present the programmer with a list of classes scaled by their importance.

- **Lenses**: a number of lenses have been defined that change the font size of a class to be proportional to its importance in the system. The importance can be based upon any metric such as the number of references to the class or the number of other classes that the class itself uses. This allows a sense of scale, in terms of importance or weighting, to be superimposed upon the graphical representation of the classes produced by the reverse engineering tools.

The C++ Lens tool uses the Microsoft Visual C++ browser database as the information source. Using this browser guarantees that the tool recognises the same language as the compiler. Essentially, the browser database is queried for a list of all translation units in the system and for each translation unit it reports all the classes, functions and global data. For each class it reports all member functions and variables and for each class member the attributes describing its protection, storage and scope.

The C++ Lens tool can depict program data due to the unique handle that represents every item in the browser database. The C++ Lens tool has a class hierarchy that has a database handle abstraction as a base class. Derived from this base class are classes that represented modules, classes, function and data members. In addition modules can contain classes; classes can contain function and data members; classes can also contain all their derived classes (the tool currently assumes single inheritance). The modules, classes, functions and data members are all held as linked lists, so that they can be formed into tree structures, the nodes of which can be visited by enquirers, due to the database handle class exporting a visitor pattern interface.

4. Applying the C++ Lens Tool to Large-scale Code Bases

The C++ Lens tool incorporates a number of defined lenses that enable the font size of a class to be printed proportional to its importance. The importance can be based upon a number of defined factors such as the number of references to the class, the number of other classes that use the class, or even the number of other classes that the class itself uses. There are currently two basic types of lenses. These are explained with reference to their use with the Microsoft Foundation Class (MFC) library.

4.1. Reference Lens:

This lens is based upon a simple metric; the number of references made anywhere in a program to a particular class. The largest most visible classes in the resultant hierarchical diagrams are the most important interfaces in the application. This enables a maintainer to immediately see if the inheritance model has errors in it. For example, if a base class is drawn small, with large derived classes, then it is not acting as an effective interface – the dependencies are on the wrong objects. A maintainer can also immediately see the likely build times that a change to the header file of a class will cause. This is unimportant for small projects, but for industrial projects with many hundreds of KLOC, the impact of such a change can hugely impede development progress.

Figure 1 shows part of the MFC (Microsoft Foundation Class) hierarchy shown with a reference lens applied. It can be seen that the root base class (to over 150 classes), CObject, has been modelled well, signified by the fact that it is drawn relatively large. More controversial is the large number of references to CWnd. CWnd has many private and protected members. These members are likely to have been changed during development more rapidly than the CWnd interface. Therefore there would have been many 'rebuild the world' header file changes that could have been avoided if the designers had separated the CWnd interface from the CWnd implementation.

![Figure 1. The reference lens applied to MFC](image)

4.2. Uses Lens

This lens can answer the question ‘how easy will this class be to reuse in another setting?’ If a class is printed large with this lens then it is tightly coupled with other parts of the application, making it hard to extract this class for reuse.

Figure 2 shows the Uses Lens applied to the same part of the MFC as Figure 1. CObject is remarkable for it’s lack of size. This means that it is loosely coupled to the rest of MFC and could be fairly easily extracted (if you wanted to!). More importantly CWinApp is huge, this means that to use CWinApp away from the MFC context would be almost impossible. As all MFC applications have an instance of CWinApp that is globally available,
any application written using MFC will be highly dependent upon CWinApp and therefore the application will always be tied to MFC.

4.3. Lenses applied to Swift

The success of the tool however, has been most apparent in its application to the development and maintenance of source code from commercial software on a daily basis. Figure 3 shows a part of the class hierarchy from Swift, a product developed by the sponsor of this research. Swift contains well in excess of 100 KLOC of bespoke C++ code. It is based upon the Microsoft Foundation Classes and has many years of development effort invested in it. It can be seen from Figure 3, produced without any lenses applied, that the six classes represented are related by inheritance. What cannot be seen from this diagram, however, is the impact of making a change to any of these classes. That is, there is no information about what roles the classes play in the larger context of the whole program.

In contrast, Figure 4 shows the same classes with a reference lens applied. It is immediately obvious that CBlock is a base class acting as an interface to the rest of the program. It can be easily deduced that there would be a large impact if the class were to be modified. This impact could be in terms of recompilation time and more seriously if a function needed to be retired from this class then the code ripple effect would be very great.

Figure 5 shows the same classes again but with the uses lens applied. Here both CBlock and CTransmit can be seen to make most use of other classes in the code. This implies that the CBlock interface is heavily coupled with other classes, and that the pair of classes in combination are the major force behind the abstraction. The other classes in the hierarchy are only minor specialisations of the key abstraction.

In contrast, Figure 4 shows the same classes with a reference lens applied. It is immediately obvious that CBlock is a base class acting as an interface to the rest of the program. It can be easily deduced that there would be a large impact if the class were to be modified. This impact could be in terms of recompilation time and more seriously if a function needed to be retired from this class then the code ripple effect would be very great.

It can also be seen that CTransmit is slightly larger than CBlock. This shows that CTransmit must contain the majority of the implementation detail as it makes heavy use of other classes in the code base. It can thus be inferred that CTransmit is highly likely to have to be recompiled when other abstractions change.

Figure 6 shows a different part of the Swift class hierarchy. This is the type of diagram produced by traditional design tools that support reverse engineering. It conveys no more information than the inheritance relationships of the classes.
Figure 7 shows the same classes under a uses lens. The diagram indicates that CWidget is an abstract interface that the other classes implement and that there appears to be no problems associated with this.

Figure 7. The Widget hierarchy with a uses lens applied

Figure 8 however reveals a problem. With the reference lens applied to the same piece of code, it can be seen that CWidget is not being referenced throughout the code. That is, the design intent for this small hierarchy has been lost in the implementation. All the derived classes are referenced highly by comparison to the base class. Patently the base class is not acting as an abstract interface - even though Figure 7 indicated that this was the likely intent. As a result of this the Swift development team were able to act upon this and re-factor the code.

Figure 8. The Widget hierarchy with a reference lens applied

5. Current Developments

This technique of giving a class visual characteristics dependant upon individual metrics is being extended in several ways. Beyond size, position is being investigated as a means of conveying importance. Colour is also being considered as a means of indicating a categorisation, but unless a key is also included, there is no immediate sense of hierarchy. Finally, symbols and different font faces are being attached to the class name to represent different associations of the class. These techniques should help the technique scale to even larger projects.

Currently, as shown in figures 7 and 8, the lenses have to be applied separately to the code. However, increasing the number of visual characteristics described above, enables the use of multiple axes to represent different overlapping metrics. For example, the use of two axes would enable correlation of two metrics. Thus, if a reference lens affected representation by size and a uses lens affected representation by distance from the left-hand margin, any large classes close to the left should be split into separate interface and implementation classes.

Tools have already been developed that can gather simple metrics [1, 2]. Metrics such as these are being used to develop more lenses and increase the number of properties about a system that can be displayed. The use of visualisation tools such SemNet [9] are also being investigated in order to enable three-dimensional representation of code from different angles. Indeed, once multiple inheritance is supported within the tool, then a third dimension will be necessary to avoid the layout problems of two-dimensional cyclic graphs.

The C++ Lens technique can be used to visualise any metric and any class relationship. The authors are currently constructing a tool to reverse engineer the class associations from C++ source code. It is intended that the resultant network of relationships will be viewed through another set of lenses to further aid the maintenance process. The use of lenses for class associations would require further research with regard to scalability.

6. Conclusion

The importance and difficulties associated with program comprehension are not in doubt if effective and safe changes are to be made to large-scale software systems. There is also a pressing requirement for tools to aid the comprehension process, particularly with regard to comprehending the importance of interfaces and their dependencies within the code. To date however, the methods available to acquire the knowledge essential to understanding the code base tend to be primitive and may be design or implementation style dependent.

Object-orientation emphasises the importance of class interfaces over implementation details and helps to manage complexity by compartmentalising problems. This is why it is good for dealing with large projects. The construction of good abstractions and the interdependence of interfaces are therefore where the majority of effort is spent when designing an object-oriented program. The C++ Lens technique and tool can help a designer or maintainer visualise C++ code and ensure that the classes as implemented, are weighted in proportion to the original design intent. The C++ Lens tool is currently in daily use on commercial software code of one million lines plus, can be used for both bespoke code and 3rd party source libraries and like all good reverse engineering tools has been used to load its own source code and analyse itself.

7. Acknowledgement

The authors would like to thank Sofel Limited for the practical and financial support they are giving to this project.
8. References


[16] *PC Week*, May 1998


9. Tools

[23] *LOOK!*, Objective Software Ltd, 1 Michaelson Square, Kirkton Campus, Livingston, EH54 7DP, UK.


[26] *SNiFF+*, TakeFive Software Ltd, Surrey Technology Centre, 40 Occam Road, Surrey Research Park, Guildford, Surrey, GU2 5YG, UK.