Making Movies: Watching Software Evolve through Visualisation

James Westland Cain¹ and Rachel Jane McCrindle²

¹Quantel Limited, Turnpike Road, Newbury, Berkshire, RG14 2NE, UK
james.cain@quantel.com

²Applied Software Engineering Research Group, Department of Computer Science, The University of Reading, Whiteknights, PO Box 225, Reading, Berkshire, RG6 6AY, UK
r.j.mccrindle@reading.ac.uk

Abstract. This paper introduces an innovative visualisation technique for exposing the software defects that develop as a software project evolves. The application of this technique to a large-scale industrial software project is described together with the way in which the technique was modified to enable integration with the software configuration management process. The paper shows how a number of forces acting on the project can be equated to changes in the visualisations and how this can be used as a measure of the quality of the software.

Keywords. Software Visualisation, Software Configuration Management, Large Scale Software Engineering, Software Evolution.

1 Introduction

Even the simplest of designs in evolving software systems can over many years lead to unnecessarily complex and inflexible implementations [11], which in turn can lead to vast amounts of effort expenditure during their enhancement and maintenance. Large-scale commercial developments rarely have simple design issues to solve and resultantly are particularly prone to maintenance problems. Such problems are further exacerbated by a decaying knowledge of the code as the software ages and original programmers move onto new assignments [1], and by the huge variance in effectiveness and quality of output of replacement programmers joining the team [12]. These factors make the maintenance and in particular the comprehension of a system difficult and the associated impact analysis of a proposed change a time-consuming and error prone process.

Maintenance and evolution of large-scale software systems can be aided through good initial design [14, 15, 16, 17] coupled with definition of [18, 21] and compliance to [4, 6, 19] programming heuristics. Software visualisation is another technique that has been shown to have an important role to play in program comprehension and which can be readily applied to legacy systems as well as newly developed code [3]. In order to expose the causes of software degradation and thereby assist in the comprehension of large-scale software throughout its evolution we have developed the Visual Class Multiple Lens (VCML) technique and toolset. VCML uses a combination of spatial and temporal visualisations to expose and track the increasing

complexity, brittleness and ultimate degradation of a system over time [5]. This paper gives a brief overview of the key concepts of the VCML approach and discusses how we have extended the work by combining it with a configuration management tool in order that the evolution of an entire project can be visualised. The resultant software evolution ‘movies’ enable trends and deficiencies within a system to be rapidly identified and readily tracked. Examples of movies produced from the visualisation of a large-scale commercial system are also given and explained.

2 Visual Class Multiple Lens (VCML) Technique

As projects grow the maintenance overhead starts to cause inertia. It becomes harder for the developers to add new features without breaking the existing source code. More time has to be devoted to defect tracking and bug fixing and regression testing becomes paramount. It therefore becomes especially important to understand and maintain as far as is practicable the structure of the code. VCML uses a combination of visualisation techniques with a variety of lenses to show the structure, quality and compliance to good design principles and programming heuristics of the code.

2.1 Visualisations

Spatial visualisations [1, 2, 4, 10, 20] represent graphically the structure of source code whilst temporal visualisations can be used to visualise software release histories [8]. As both spatial and temporal visualisations have proved their worth to the program comprehension process in isolation, it is reasonable to infer that further advances could be made if both approaches were used in combination. That is, there is a need for a tool to visualise source code changes over time in order to check that the design intent is still realised by the source code. Gall et al. [8] for example, have visualised software release histories through the use of colour and the third dimension. They chose to represent the system at module level rather than the source code level and at this level of granularity found that the system could scale well using time as the z-axis. Koike et al. [10] also use the Z-axis in their visualisation to represent time. However they chose to represent a system at source level and found that this strategy led to high complexity for even fairly small amounts of source code and that the resultant visualisations did not scale well. Ball et al. [1] also represent changes at source code level over time with their visualisations, and by using colour to infer activity areas, so that reengineering hot spots become apparent. Our approach takes this a stage further by combining heuristic codification into spatial and temporal visualisation of software source code through the use of the VCML technique.

2.2 Lenses

A number of *lenses* have been defined that change the visual representation of a defined characteristic 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. [3]. This technique allows class size to be determined through metrics gathered from reverse

engineering source code. The lenses presented in this paper are drawn using a radial view with information culling in order to project metrics into the visualisation.

Within our current visualisations we use *size* to represent the number of references to a class and *colour* to represent the number of classes that each class uses. The lenses are orthogonal in nature and may therefore be used in combination as a Visual Class Multiple Lens (VCML) view to expose how the different underlying metrics interact and affect compile time coupling [5]. For example, in order to assess some of the finer details of the code an overall visualisation may be refined such that the low *uses* count (blue) classes can be filtered out as shown by viewing Figure 2 (lens applied) rather than Figure 1.

2.3 A Visualisation of Piecemeal Growth

Version control systems such as MKS Source Integrity, Rational Clear Case and Microsoft Source Safe are used in industry for team development of software. They all have command line interfaces that allow for automation of source code check-out. Batch files can be made that check out images of software as it exists at a specified date and time. Therefore, using such systems, it is possible to check out source code trees at time lapsed intervals and to evaluate how the source code has changed. In our approach we are combining this output with the lenses in order to assess the evolving characteristics of a software system. Currently, the VCML are based on exhibiting characteristics of the code based on:

Increased size: the size in a VCML view indicates the number of *references* to the class throughout the source code. As the project develops the system often becomes highly coupled to a number of key classes. Thus if there is a need to change a large class there is a significant overhead in terms of recompilation and regression testing. The rest of the project is tightly coupled to the class definition. As time elapses in a project there is a tendency for classes to grow in size. The more large classes there are, the more likely a programmer will have to make a change that affects large parts of the source code. At the very least recompilation will be required and at a higher cost regression testing for all affected modules may be required. Even more costly however, is a failure to realise that regression testing is required on such a scale.

Red shift: the colour in a VCML view indicates the number of classes that the class *uses*. As the project develops class implementations get fleshed out and resultantly classes tend to get less cohesive and generally causing a ‘red shift’ in the VCML view.

3 Concatenating VCML Views into Movies

Previously, VCML views have only ever been presented in sequences of three or four images [5] and with a time span of some six months between each view. Whilst this gives a useful indication of how a software system has evolved over time the level of granularity is insufficient to enable an analysis of software evolution to be correlated with events occurring within the project team during development.

For example, factors such as staffing, skill level of developers and project management strategies have been shown to have a huge impact on software evolution

and its resultant quality [7, 12]. By taking for example, two-weekly snap shots of the software source code and making a VCML view for each, it is possible to generate ‘movies’ that illustrate the gradual change in software attributes over time. Through this approach it is also possible to identify increases in the rate of change of attributes in the VCML views and to attempt to correlate these with changes in the dynamics of the project team.

3.1 Layout Algorithm

Repeatability [9] is key to being able to make movies out of the individual images. If the layout algorithm is not deterministic and repeatable then the classes will be drawn in different locations for each source code snap shot, and the movies would show the classes ‘jumping about’ between each frame thereby preventing comparisons from being made and trends from being identified. To solve this VCML views use a radial tree view so that classes are generally always drawn in the same position.

VCML views also use a non Euclidean geometry, somewhat similar to the Focus + Context approach [13], in that the size of each element in the tree varies according to the interest that the viewer places on the element. The difference between VCML views and Focus + Context techniques is that in VCML views an attribute of each element is determining its own importance whilst Focus + Context techniques demand that the user determines where to investigate further in order to obtain more detail. VCML views thus allow the user to discover which elements are important without demanding viewer interaction.

3.2 Generating the Pictures in Theory

In order to generate the movie pictures, it was postulated that image collection could be automated through the aforementioned use of scripting (batch file) techniques. Command line driven source code check-out facilities from a configuration management database enables complete image trees to be checked-out, with each image tree containing the code in its entirety at a particular date. Therefore by using simple script files it should be possible to check out images of the code at regular (e.g. monthly or bi-weekly) intervals across the entire duration of the project, compile each check-out and save the browser database generated by the compiler. Once this operation has taken place the intermediate files can be deleted to recover disk space.

This approach would result in a directory of browser database files spanning the history of the project at regular intervals each of which can then be individually rendered by the Visual Class tool into a set of VCML views. Thus there is a fairly automated process that can generate a stack of images representing the evolution of the software, which can then be concatenated together to make a small animation or movie.

3.3 Generating the Pictures in Practice

In practice however, the above method had several problems which needed to be overcome. For example, one of the source code history files was found to have been corrupted about one year prior to archiving for the purposes of this research. This meant that about six months of source code images would not compile. Actually, the

corruption was relatively easy to fix by hand but as it has to be done for each image, was very labour intensive.

A second and more serious problem was that compilers, system header files and operating systems themselves evolve over time and resultantly most of the image trees would not link. This meant that it was not possible to build an executable from the source code. This did not matter for the purposes of the research, as only a compile is required to generate the information necessary to create a browser database. However, it did mean that the script files had to invoke the utility to generate the browser databases separately from the invocation of the compiler.

Finally Visual Class does not inherently have the ability to save bitmaps, even though this can easily be added at a future date. In the meantime however we are using screen dumps as an alternative approach. Another problem was that Visual Class tends to leak memory on repeated loading of browser databases; so each script invocation only generated four bitmaps, which although not prohibitive is inconvenient and time consuming (this too will be fixed in future versions of the tool).

4 The Movies

Swift is a large-scale commercial piece of software in daily use and which is continually evolving to meet the demands of its Industry. It has well over 100,000 lines of original in-house code as well as incorporating third party COTS (Components-off-the-Shelf) such as MFC (Microsoft Foundation Classes). Development of *Swift* began in the Spring of 1997 utilising a single developer and growing to eight fulltime personnel by the Summer of 1999. Resultantly, as well as the added complexity introduced through the required updates, the code became a mix of the coding styles and ideologies of the individual programmers.

Figure 1 shows twelve VCML views of the *Swift* project at two-month intervals. Each diagram has been generated by taking a snapshot of the source code, compiling it to generate a browser database, and loading the database in Visual Class.

Immediately it can be seen that the internal complexity of the *Swift* project is increasing over time. The first frame show 549 classes, the last shows 947. That is, the number of classes has almost doubled, and more importantly the proportion of large classes has grown even more rapidly. This can be seen as a visualisation of the compile time coupling within *Swift*.

Figure 2 also shows twelve VCML views of the *Swift* project at two-month intervals. It is different from Figure 1 because the lens threshold has been set to twenty in each image. This means that any class that uses less than twenty other classes in its implementation are not drawn. The first image has 6 classes, of which only one is sizable. Two years later, the final image has eleven classes, most of which are large. These views also show that there is a 'red shift' over time. This tends to indicate that the major classes in the project are becoming less cohesive as the system evolves.

4.1 Colour Change Analysis

From the outset *CWnd* is a prominent class in Figure 2's images. Over the two years it gradually grows in size but it does not change colour. This is because *CWnd* is at the

heart of the Microsoft Foundation Classes, a COTS component that was used in the implementation of Swift. As the MFC version was not changed during the project development then the implementation of `CWnd` did not change. Thus the colour of `CWnd` is constant over time.

Although it is initially hardly visible, `CMainFrame` is present on every image in Figure 2. It only however becomes prominent towards the end of the first year's development. Over the duration of the project it has changed colour from yellow to red. This class is derived from an MFC class, but implemented by the Swift team. It is central to the application's implementation, as it represents the main window of the application, handling most of the menu invocations. Therefore it is both highly dependent on the MFC classes and acts as a conduit for message passing around the application. The change from yellow to red indicates that the developers are adding increasing functionality to `CMainFrame`, gradually turning it into a 'god' class [21].

4.2 Size Change Analysis

`CWnd` grows steadily in each image in Figure 2. This makes sense, as there were increasingly more windows and dialogs implemented in Swift as time went by, and each window and dialog implementation would inherit directly or indirectly from `CWnd`. As the MFC implementation was never changed this is not a problem because `CWnd` was never the cause of a recompilation.

`CMainFrame` grows markedly from the beginning to the end of the visualisation. It is being used by an increasing number of classes. This indicates that as time progresses an increasing amount of code will need to be recompiled if it ever changes, and as it grows the compile time coupling with the rest of Swift is increasing.

4.3 Size Increase and Red Shift

`CMainFrame` is both growing in size and moving across the colour spectrum towards red. The combination of these attributes means that `CMainFrame` is likely to change frequently and each time it does it demands the recompilation of more and more dependent classes.

Given this issue also applies to a host of other classes such as `CBlock`, `CTransmit`, `CBound`, `CSwit32Doc` and `CSwift32App`, one can see that a good proportion of the development work was being focused on these key classes. Each time they change most of the application needs recompilation. Towards the end of the two-year period these classes were being changed daily. Each time this occurred the developer would have to set the compiler running for about an hour.

Worse still, each time they were checked into the revision control system, all the other developers would have to recompile their code bases. Each change could cost the team as a whole six hours compilation time. The effect of this was that people were reluctant to check in their changes. They would keep code in private branches for many days or weeks. When they came to checking in their changes the head revision of the code base was often quite different to the version that they had checked out, so large merges of code were common.

Merging source code files is an arduous and labour intensive process, where conflicting code changes have to be resolved by comparing three files (the old checked out version, the modified local version and the head revision). Most

commercial software configuration management tools supply tools to aid with these types of merging. Even so the merging process can easily reintroduce bugs that have already been fixed. This happens because the programmer may choose an old piece of code over a head revision to enable their code changes to still make sense and compile.

5 Program Comprehension

Movies can show a great amount of detail regarding system behaviour over time, for example, it is possible to equate the speed up in growth of the in big red classes to the numbers of developers joining the project and their experience. The Swift project team changed markedly over the two-year time frame. The first six months of the project had only one developer, the designer of the product. The second six months saw the addition of another senior engineer. During the third six-month period two junior developers were recruited. During the final six months the team had six developers, with one of the senior developers becoming the project manager and the original designer joining a different department (though he was available for consultation).

During the last six months many bugs that had been fixed seemed to reappear in new versions (patches were being issued to customers approximately every three weeks) and productivity seemed to slow even though there were more developers working on the system.

The last three images in Figure 1 show a marked increase in the rate of growth of the classes and the red shift. This shows that the novice programmers were placing more and more code in 'god' like classes. This breaks one of the heuristics advocated by [21] primarily because programmers new to C++ tend to write in a 'C' like manner that does not scale well. This then leads to large recompile times and the software becomes brittle and prone to bugs reoccurring.

Pressure on the project team to deliver software releases every three weeks meant that 'fire-fighting' fixes were common. In the short term these worked and got the product 'out of the door', but in the long term one can see from the VCML views during the last six months, that software quality was being eroded by these practices.

6 Conclusion

The VCML view can show how some classes can dominate a project when they both use many classes in their implementations and have many references to them in the rest of the source code. The concatenation of the views into movies allows the visualisation of the effects of code changes over time. The novel technique of making VCML movies illuminates the forces that cause the need for refactoring as projects grow. The visualisation presented here shows how the code from a commercial project became more brittle as junior programmers took over the software maintenance. The views show how problems of program comprehension can cause very hard project management problems.

VCML views made into movies allow for the visualisation of the project evolution on a macro scale. They show that software quality can be visualised to allow for

analysis of hotspots, areas of code that are impeding software development and those which need to be refactored.

We are currently working on visualising another large industrial project and the effects on the technique before and after major code refactoring takes place. We are also working on correlating the VCML movies with ‘bug’ reports from the two products.

Acknowledgements

We would like to acknowledge the financial and project support provided by Quantel Limited and Softel Limited, without which this work would not have been possible.

References

- [1] Ball T., Eick S., *Software Visualisation in the Large*, IEEE Computer, April 1996.
- [2] Burkwald S., Eick S., Rivard K., Pryce J., *Visualising Year 2000 Program Changes*, 6th IWPC June 1998.
- [3] Cain J., McCrindle R., *Program Visualisation using C++ Lenses*, Proceedings 7th International Workshop on Program Comprehension, 20-26, Pennsylvania, May 1999.
- [4] Cain J., McCrindle R., *Visual Class Tool for Assessing and Improving C++ Product Quality*, 3rd Annual IASTED International Conference Software Engineering and Applications, Arizona, October 1999.
- [5] Cain J., McCrindle R., *The Spatial and Temporal Visualisation of Large Software Systems*, submitted for peer review, January 2001.
- [6] Ciupke O., *Automatic Detection of Design Problems in Object-Oriented Reengineering*, Technology of Object-Oriented Languages and Systems, August 1999.
- [7] Coplien, J., A Generative Development-Process Pattern Language in Coplien J., & Schmidt D., (eds.), *Patterns Languages of Program Design*, Addison Wesley, 1995.
- [8] Gall H., Jazayeri M., Riva C., *Visualizing Software Release Histories: The Use of Colour and The Third Dimension*, IEEE Int. Conf. on Software Maintenance, September 1999.
- [9] Herman, I., Melançon, G., Marshall, M. S., *Graph Visualization and Navigation in Information Visualization: A Survey*, 24 IEEE Transactions on Visualization and Computer Graphics, vol. 6, no. 1, January-March 2000.
- [10] Koike H., Chu H., *How Does 3-D Visualisation Work in Software Engineering? Empirical Study of a 3-D Version/Module Visualisation System*, The 20th International Conference on Software Engineering, 19 - 25 April 1998.
- [11] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison Wesley, 1999.
- [12] Landauer, T.K., *The Trouble with Computers: Usefulness, Usability, & Productivity*, London: MIT Press, 1995
- [13] Lamping, J., Rao, R., Pirolli P., *A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies*, ACM Computer-Human Interaction, 1995
- [14] Martin, R., C., *The Open-Closed Principle*, *Engineering Notebook*, January 1996.
- [15] Martin, R., C., *The Liskov Substitution Principle*, *Engineering Notebook*, March 1996.
- [16] Martin, R., C., *The Dependency Inversion Principle*, *Engineering Notebook*, May 1996.
- [17] Martin, R., C., *The Interface Segregation Principle*, *Engineering Notebook*, August 1996.
- [18] Meyers, S., *Effective C++*, 2nd Edition, Addison Wesley, 1997
- [19] Meyers, S., Klaus, M., *A First Look at C++ Program Analysers*, *Dr. Dobbs Journal*, Feb. 1997.
- [20] Rao, R., Card, S., *Exploring Large Tables with the Table Lens*, ACM Computer Human Interaction, 1995.
- [21] Riel, A. J., *Object-Oriented Design Heuristics*, Addison Wesley, 1996.

Fig. 1. Twelve images with a threshold of zero, at approximately two-month intervals (animated at <http://www.reading.ac.uk/~sssmccri>)

Fig. 2. Twelve images with a threshold of twenty, at approximately two-month intervals (animated at <http://www.reading.ac.uk/~sssmccri>)