A Longitudinal study of Depth of Inheritance and its effects on programmer maintenance effort

Adrian Creegan
Department of Computer Science and Information Systems,
University of Limerick,
Castletroy, Limerick,
Ireland
adrian.creegan@ul.ie

Chris Exton
Department of Computer Science and Information Systems,
University of Limerick,
Castletroy, Limerick,
Ireland.
chris.exton@ul.ie

Abstract

Inheritance is a defining feature of the object oriented paradigm. Amongst other things, it enables the programmer to establish conceptual relations between domain objects, and promotes the partitioning of systems to increase reuse and dynamic extension. That notwithstanding, it has received some criticism especially in relation to the burden of comprehension it imposes on maintainers. As a result, measurements of this code feature have been incorporated into several established metrics as contributing to the level of predicted effort.

The question of its negative impact on maintenance effort is not a new one, indeed it has been revisited several times over the past 15 years. However, there is little hard empirical evidence quantifying its effects. Much of the previous research arrives at inconclusive and in some cases contradictory findings. While some of these previous studies have a strong empirical foundation, the maintenance tasks investigated have by and large been confined to small and somewhat idealised source code bases. Furthermore, many of these studies were performed in laboratory settings employing students as experimental subjects. They may not therefore be representative of typical maintenance scenarios.

This paper attempts to re-examine this subject. It is hoped, that the methodology employed will enable its findings to be more extensible to production situations. To this end, a large open source project was chosen as a subject for study. Data was extracted from this project using repository mining techniques by means of several custom tools which are discussed. These tools established the inheritance hierarchies of the source code and captured their evolution over a time frame of six years. The resulting data was then statistically analysed. Special attention was paid to establishing the presence of large scale restructuring operations. Such operations have been predicted as a strategy to cope with accumulating conceptual discrepancies in maintained software.

The study found that while such a signature was present, it was most likely due to seasonal effects. The depth of inheritance of associated code could not be correlated with this signature to any significant degree. A functional relationship between the number of file revisions and their distribution within the repository was also observed. This along with the previous results is presented and discussed.

Keywords: POP-II.B. maintenance; POP-II.B. programming comprehension; POP-III.C. procedural / object-oriented; POP-V.B. longitudinal studies.

1. Introduction

THE importance of the maintenance phase in the software development life cycle model is well recognised. Related activities have been estimated to account for as much as 90 percent of the total

lifecycle cost (Erlikh 2000). As Lehman (1980) observed when formulating his laws of software evolution, maintenance is both necessary and unavoidable in order for systems to remain useful. Understanding factors that impact this phase of the software life-cycle is of critical importance.

Fundamental to the Object Oriented development paradigm is the notion of object inheritance and hierarchical decomposition and partitioning of features. According to Meyer (1988), inheritance provides the support necessary to exploit the conceptual relations between classes of domain objects (c.f. page 217). It affords the reuse, encapsulation, and dynamic binding of source code. However, this construct is not without its drawbacks and concern has been voiced on occasion over the burden it imposes upon comprehension tasks. Particular attention has been paid to the coupling it introduces between bodies of source code that are conceptually related but may be physically dispersed throughout the code base. In some circumstances, a full understanding of such code may only be arrived at by means of navigating the whole inheritance hierarchy. When this is not done, as previously research would suggest is often the case, the maintainer may be oblivious to behaviour defined elsewhere in the class hierarchy, and its consequences to the code under their immediate consideration.

Soloway et al. (1988) considered similar problems in procedural code bases and identified that they can be source of error in maintenance operations. He referred to them as *Delocalized Plans* and argued that in the presence of delocalization, maintainers may be tempted to draw incorrect inferences on merely what is locally apparent. These conclusions may differ significantly from those drawn if the maintainer elected to navigate the various conceptual dependencies. As Von Mayrhauser (1995) observed, navigation of such plans is clearly not consistent with the purely opportunistic behaviour observed in the majority of programmers. While Soloway's research was based on procedural code, the implications for the object oriented domain have been examined by Wilde and Huite (1992), and studied in terms of inspection impact by Dunsmore et al. (2000).

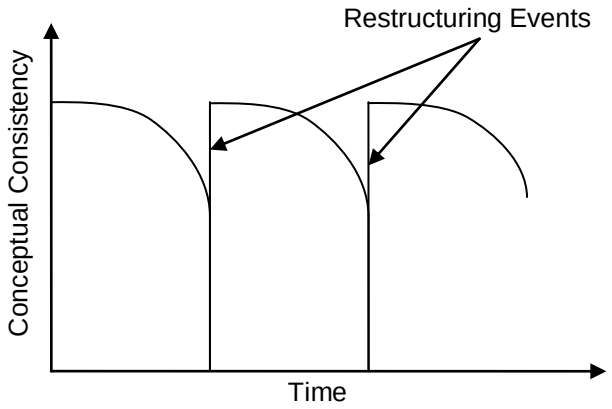

Figure 1Hierarchy evolution with re-structuring (after Dvorak)

Dvorak (1994) proposed that class hierarchy depth factors in the comprehension of object oriented code. He suggested that deep inheritance features impact negatively upon the efficiency of the software maintenance phase. Furthermore, he predicted that this effect might be observable in the change history of the application's code base. In his opinion, sections of the source code would evolve in a manner whereby its conceptual consistency would erode over time. These discrepancies will accumulate until a certain critical mass is achieved. At this juncture, maintenance staff will be compelled to undertake large scale restructuring in order to restore the conceptual integrity of the application's design. According to Dvorak, these operations would manifest themselves as a cyclical component signature to the source code modification history. Moreover, he suggested that this effect should be more pronounced in classes at deeper levels in the inheritance hierarchy due to

the higher likelihood of delocalisation. At least two other studies (Daly et al. 1995)(Prechelet et al. 1998) have examined the impact of class inheritance depth on maintenance effort. Both studies were laboratory based and both reached contradicting conclusions.

2. Research Goal

There is to some degree an accepted wisdom that depth of inheritance impacts negatively on code maintenance. Its inclusion as a component of several popular object oriented code metrics would seem to support this notion (Chidamber et al. 1994)(Fiorvanti et al. 2001)(Li et al. 1993). Contemporary theories of human cognition, as substantiated by observed programmer behaviour, lend further support to this belief. That notwithstanding, hard empirical evidence quantifying its impact on code maintenance with a degree of confidence is difficult to come by.

A few of the studies conducted to date have arrived at contradictory findings with the work of Daly et al. (1995) and Prechelet et al. (1998) being conspicuous in this regard. In these and other cases conclusions have been drawn from observations of students in laboratory conditions. In the author's opinion has diminished their external validity.

There are some notable exceptions. Fioravanti and Nesi (2001), performed a study that correlated maintenance effort to object oriented statistics, however the application they studied was quite small. There is also the study by Li and Henry (1993) which established a positive correlation of maintenance effort to inheritance but its individual contribution was not quantified; it was reported as an aggregate figure comprising inheritance depth and several other metrics.

It is the intention of this and a number of follow up papers to re-examine the issue of depth of inheritance and attempt to quantify its impact on the maintenance phase in a manner that is realistically extensible to production scenarios. This study will concentrate on quantifying overall repository change rates at the file level and establishing a firm foundation for further work. It will also examine the code committal signatures for evidence of cognitive dissonance as predicted by Dvorak.

3. Research Method

The methodology employed by the authors may be summarised as follows. A large, enduring, open source software application was identified to in order provide data. The Software Configuration Management (SCM) system for this project was then mined for check-in histories of its constituent files using a custom tool *CVS-Analyzer*. A second custom tool *Code-Analyzer* then operated over all files indexed by the first tool re-constructing and parsing each check-in variant. Information from each parse including class extension and implementation details, methods, fields and statement counts was then extracted and indexed against the revision history. A final custom tool *Code-Linker* then operated over the data set generated by the first two tools providing a longitudinally aware linkage of the various extension class references. All three custom tools recorded their outputs in a relational database. Finally, the results from the three custom tools were analysed for statistical correlations.

3.1 Source Data

All source code data for the analysis was extracted from the Eclipse Programmer's Workbench. Eclipse is a popular open source application with large community of users worldwide. It boasts more than 1000 optional add-ons, supporting multiple languages, tools, and software development methodologies. The project was initially founded by IBM in November 2001 in association with a consortium of software vendors.

In February 2004, the Eclipse foundation was founded. Chartered as a not-for-profit organisation it was established as an independent body from both IBM, and the consortium which at this stage had

grown to over 80 members. The foundation is charged with driving the platform's development, and maintaining an open, vendor independent community around the project.

The rational for choosing this project for study may be summarised as:

1. It is a popular and widely used application. The code is fit for purpose and functional. The application has undergone continual refinement in order to enjoy this popularity over a 6 year period.
2. The application has been in the public domain for 6 years at the time of writing and as a result there is a considerable body of data available for analysis.
3. As the application prior to donation was proprietary to IBM and its associates, it constitutes a plausible candidate of a body of work developed to commercial production standards.

Since its release, Eclipse has been maintained by developers from both IBM, and the open source community. Many of the application's contributors were not involved in its initial development; and the project constitutes a useful instrument for investigating programmer comprehension of unfamiliar code. The Eclipse platform is written in the Java programming language.

The complete version history of the source code was downloaded from source code repository onto a local hard disk. The Eclipse web site conveniently provides the full repository as a single file which is updated regularly to reflect the current code base. The file, in zip format, inflates as a file system on a local disk; all major sub-systems and add-ons are represented as separate sub-directories. The repository snapshot as employed by the author consumes some 8 Giga Bytes of disk storage inflated.

3.2 Development of Analytical tools

CVS-Analyzer is a custom tool written to operate over the internal repository files. The tool was configured to iterate over all Java files in the inflated Eclipse repository and extracted metadata on the revision history for each file under SCM. Eclipse uses the open source CVS Concurrent Versioning System. While CVS has been largely superseded by newer open source SCM systems (such as Subversion), its simplified storage model is amenable to large scale data mining operations. CVS uses the native file system as its backing storage and its topology is hierarchical. A project comprises of a root entity off which modules, each represented by a corresponding sub-directory, extend. Modules in turn can exhibit finer grain structure which is captured by sub-directories and the de-composition repeats ad-infinitum.

Each project file within an application is represented by a corresponding storage file. All versions of a given project file are encoded within its associated storage file using the GNU RCS Revision Control System format. This scheme stores the latest version of the file as a current image; previous versions are recovered by successively applying changes as specified in the file's log and generated when the code was previously checked in. This operation logically 'undoes' the previous check-ins until the desired revision is recovered. Care must be exercised when designing code to perform this operation as errors accumulate.

Code-Analyzer is a custom tool designed to extract this information for each revision within a storage file. The extracted data from the parsing operation is logged against the source code's corresponding file and revision records in the analysis database. The Eclipse Platform is implemented exclusively in Java. In order to extract the type structure from the source files some consideration was given to the approach of building multiple versions of the application and employing the Java reflection API. This method was rejected due to concerns over the reliability of the build process and the time it would take.

Instead the authors elected to construct a Java lexical analyser and Parser to extract the required information. Though not a trivial task itself, this undertaking is greatly simplified by the use of open source lexical analyser toolsets. One such toolset, ANTLR 3, was utilised in the project. Development was further simplified by the availability of a wide selection of contributed grammars;

one of which was used as a starting point when constructing the tool's parsing logic. The ANTLR toolset can be configured to generate classes in both Java and C# programming languages from the supplied grammar definition. These classes are then linked to the application specific source code to form the data input component of the resulting application. Integration and adaptation of these generated classes with application specific code is simplified by the library's extensive use of factory patterns and program to interface methodology.

The application was constructed to identify the following features in a supplied source file:

- All *import* clauses thereby allowing it to capture the context in which type references should be interpreted.
- All *package* clauses in order to capture the context in which any class definitions should be interpreted.
- All *class* definition clauses, including those for inner classes and interfaces are captured; this information is considered of primary importance to the analysis. Care is taken to ensure that *Generic* class definitions are handled appropriately. In the tool, such class definitions are flagged as Generic but no attempt is made to establish type reference linkages on the generic's *type arguments*. Type annotations are not captured.
- All *interface* definition clauses, including those for inner interfaces are captured; this information is considered of secondary importance to the analysis. The tool currently logs the interface's identifier; it does not de-compose the interface into its members, capture any generic attributes or capture any inner class or inner interface definitions. It was felt that this would not significantly influence the analysis though if needs be the tool can be configured to capture such data with minimal modification.
- All *field* definition clauses, including the field name, and type classifier; generic classifiers are captured but their associated *type arguments* were not.
- All *method* definition clauses, including those for constructors are captured. The message signature is captured including any generic classifiers which are flagged; generic *type arguments* for such classifiers are not. Anonymous class definitions within methods are not tracked; neither is any information regarding exception declarations.
- All *extends* clauses are captured and references to the generalized type recorded. This provides the basic information used to determine the depth of inheritance in the class hierarchy.
- All *implements* clauses are captured. The associated *type list* is captured but no attempt is made to establish type linkages.
- All comments within the file are stored to database as a single text field at the file revision level. Capturing such text per method is problematic and was not pursued for two reasons:
    1. ANTLR is usually configured to code comment data onto a separate token stream. This improves the efficiency of the main parse.
    2. Comments associated with a method are not always guaranteed to be within the method body (e.g. they may be in the method header).
- The number of *class-statements* for each class was measured as distinct from the class' total lines of code (in new line characters) or semi-colon counts. Class statement counts were aggregated onto those of contained inner classes.
- The number of *method-statements* for each method was measured as distinct from the method's total lines of code (in new line characters) or semi-colon counts.

All results were logged to relational database under the previously discussed database schema.

The last tool applied to the data extraction was the *Code-Linker* tool. The tools discussed previously capture the static structure of the source code but are confined to file boundaries and record the state at an arbitrary point in time (i.e. the time the author decided to commit the code). The *Code-Linker* tool scans the static analysis stored in the relational database and associates linkages between the various type definitions and, if necessary, across the file boundaries. The tool was written in C#.

The *Code-Linker* was designed to preserve correct inter-type relationships in the time dimension; a specific version of a specialising class must be analysed in terms of the correct version of its super class. This was considered desirable since it is a prerequisite to cross-sectional (between classifiers) and longitudinal (within classifier) statistical analysis. It is, however, unreliable to assume that associated classes are committed together. It is also unsafe to suppose that super classes and sub-classes are committed consistently in a certain relative order. It was therefore considered more accurate to establish the inter-class inheritance relationships at an application label vis-à-vis file revision level; the rational being that the code base has a higher likelihood of self consistency. This was considered reasonable since conceptually a labelling operation collapses the state of the code base as a whole to a single identity; which is then used as a specifier when this state is to be recovered in toto at some future point. It is worth noting that the specific time a label is applied is not captured within the CVS backing data and needs to be inferred from the commit times of that label's associated file revisions.

The tool is designed to scan the repository on a label by label basis. All file revisions active for a given label and their contained type definitions are processed together. Cross linkages are determined between the various types with particular attention paid to any inheritance relationships. The results are stored in the database against the type definitions captured previously. The depth of inheritance of the type is also computed at this point. Any failed linkages are flagged for manual analysis and intervention. At this point, the relational database has a statically analysed type structure that is fully type referenced and available for analysis.

3.3 Data Analysis Method

A preliminary analysis of the data was confined for the purposes of this paper to examining the overall evolution of the repository and gross inheritance code characteristics. The reader is referred to a number of follow up papers for a more detailed examination of depth of inheritance effects.

The analysis performed here examined the check-in committal at the file level. The statistical distribution of file revisions was determined and analysed in order to establish whether the repository's overall evolutionary characteristics were well behaved over time and predictable. This analysis also provided a simple mechanism for identifying possible candidate files displaying cognitive dissonance. It was expected that such files would exhibit supra variability above that of the general body of source code.

A simple linear regression model was fitted to this distribution and repeated for several points in time in the project's history. Covariance and significance tests were then applied so as to 1) establish correlation with the model, 2) determine the descriptive capability of the model, and 3) establish the statistical likelihood that any candidate dissonant files are not due to random effects. A cursory examination of the distribution of classes according to depth of inheritance along with their change rates was also conducted.

4. Results

Table 1 summarises the gross statistics captured from the data extraction phase. The 'Total Eclipse Derived Type Revs' is the gross figure of type revisions that were specialised from a class within the Eclipse code base. Only those classes derived from eclipse project classes were considered as candidates for the analysis. To include external libraries would immediately pose the question as to what inheritance depth should be assigned to the external references. It was also felt that enforcing closure on the input data would improve the likelihood of exposing potential dissonance effects.

The ‘Total Revisions Net of Test Classes’ is the number of type revisions that have a namespace starting with ‘org.eclipse’. This eliminates a large number of automated unit tests that are likely to be written by code generators and therefore contribute nothing to the analysis. The last three figures summarise the number of individual classifier instances within the code base vis-à-vis the number of revisions of the classifiers which are conveyed in the preceding totals. An individual classifier is identified by the combination of its fully qualified name and the path to the source file relative to the project root.

Gross Statistic	Value
Total Files	73562
Total Check-ins (File Revisions)	581930
First Recorded Check-in	2001-04-28 15:41:01
Last Recorded Check-in	2007-10-12 23:00:49
Total Distinct Labels Applied	9098
Total Labelled File Revisions	12900172
Number of Distinct Classes	87698
Total Type Revisions	717590
Total Eclipse Derived Type Revs	422194
Total Revisions Net of Test Classes	421822
Total Root Type Revisions	199813
Total Specialised Type Revisions	222009
Nbr Root Types	30239
Nbr Specialised Types	26428
Nbr Specialisations Considered	24973

Table 1 - Gross Repository Statistics

The final figure ‘Nbr Specialisations Considered’ is a subset of all Eclipse specialisations that are unambiguously linked to their super classes. The code linkage operation assigns a degree of certainty to the super class linkage based upon the number of candidate types available matching the classes extends specification. This disambiguation is arrived at according to a variety of criteria based on file’s location, package, fully qualified domain name and location within the source code file system. In a minority of cases, the number of available candidates exceeds 1. This can happen for a variety of reasons that include partitioning and promotion of incubation source code, and the relocation of class definitions within the file system. To improve the quality of the analysis, the input data set was confined to those classes that had only one eligible candidate identified in the type linking operation.

The version of the repository operated upon was that archived by Eclipse staff on 2007-14-10. The content of the repository was obtained as a distribution file from the Eclipse web site. It comprised a zip compressed archive of all revisions of all Eclipse related projects up to that date. The file inflated on the local hard-drive consumed some 8 GBytes of disk space.

Analysis took approximately 8 days on two servers the largest of which was a Dual Core 64bit

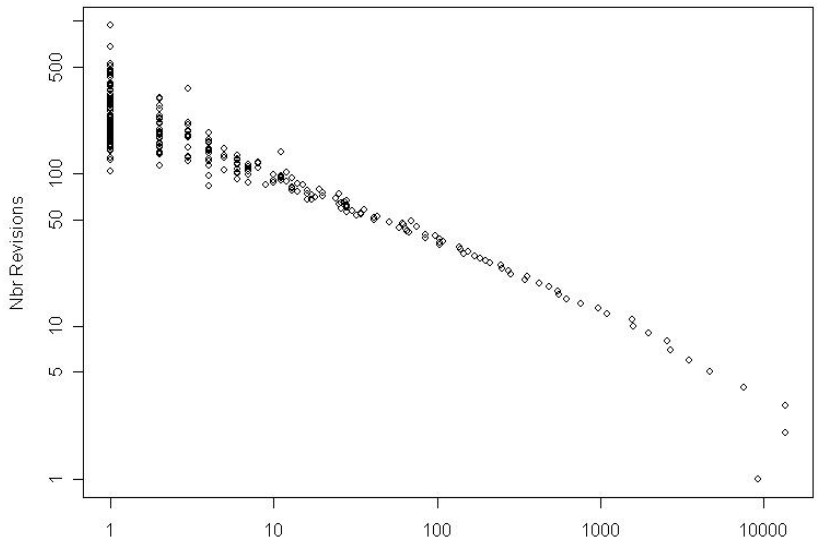

Figure 2 - File Revision Distribution

processor running at 3.2 GHz. This machine was configured with a RAID 5 high speed disk subsystem and a total of 4 GBytes of system memory. The database holding the extracted data consumed approximately 3 GBytes of disk space.

5. Analysis

5.1 Analysis of General Repository Evolution

The plot in Figure 2 demonstrates the distribution of file revisions over the projects lifetime. In evidence is a strong negative correlation that is approximately linear in the log domain for both the number of revisions experienced by a file and the number of files exhibiting those revisions. This analysis was repeated at previous instances in the project's history and the relationship was found to hold over time. The results of this analysis are shown in Table 2.

A simple linear regression model was fitted in an attempt to describe the file distribution. It considered the log of the number of revisions for a group of files being a dependant variable which is a function of the log of the number of files in the group and the total repository age (in days). The first of the two coefficients captures the distribution relationship at an instance in time, the second capturing the change in this relationship over the project's lifetime. The results of the model are shown below.

lm(formula = log.nbr_rev ~ log.nbr_files + age)

Residuals:

Min	1Q	Median	3Q	Max
-1.519179	-0.129190	0.007871	0.143578	1.224110

Coefficients :

	Estimate	Std. Error	t value	Pr(>|t|)
Intercept	4.73	3.57×10^{-2}	132.64	$<2 \times 10^{-16}$ ***
log.nbr_files	-0.45	5.12×10^{-3}	-87.34	$<2 \times 10^{-16}$ ***
Age	3.72×10^{-4}	1.79×10^{-5}	20.77	$<2 \times 10^{-16}$ ***

Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.2816 on 596 degrees of freedom

Multiple R-squared: 0.9322, Adjusted R-squared: 0.9319

F-statistic: 4095 on 2 and 596 DF, p-value: < 2.2e-16

Table 2 – Revision Distribution Model Coefficients

The adjusted R^2 figure is calculated to be 0.9319 which indicates that 93% of the total data variance is accounted for by the model. The F-statistic is 4095 on 2 parameters with 596 degrees of freedom. Since the critical value for the F-distribution at the p = 0.01 level is only F(2, 596) = 4.61and considerably less than the computed value of 4095, it would suggest that the model is highly significant. Moreover, the critical value of the t-distribution at the 0.01 level with 596 degrees of freedom is 2.33; we can therefore conclude that all parameters (including the intercept term) contribute significantly.

The resulting model may be re-formulated as shown in Formula 1. The parameters a, b and c correspond to the above log.nbr_files, age and intercept coefficients respectively.

$$f\left(\log(nrevs)|\log(nfiles), age\right) = a \times \log(nfiles) + b \times age + c$$

or

$$nrevs = nfiles^{a} \times 10^{b \times age + c}$$

Formula 1 – Model of Revision Distribution

A brief analysis of the model's residuals was also conducted and is shown in Figure 4. The plots illustrates that there is some variance from the model's theoretical predictions. Several leverage points, or outliers are in evidence and these significantly influence the model. Three of these are associated with the distribution of files having only one revision in each epoch. A case for removing these from the analysis could be made by employing the rational that they are baselines, not revisions.

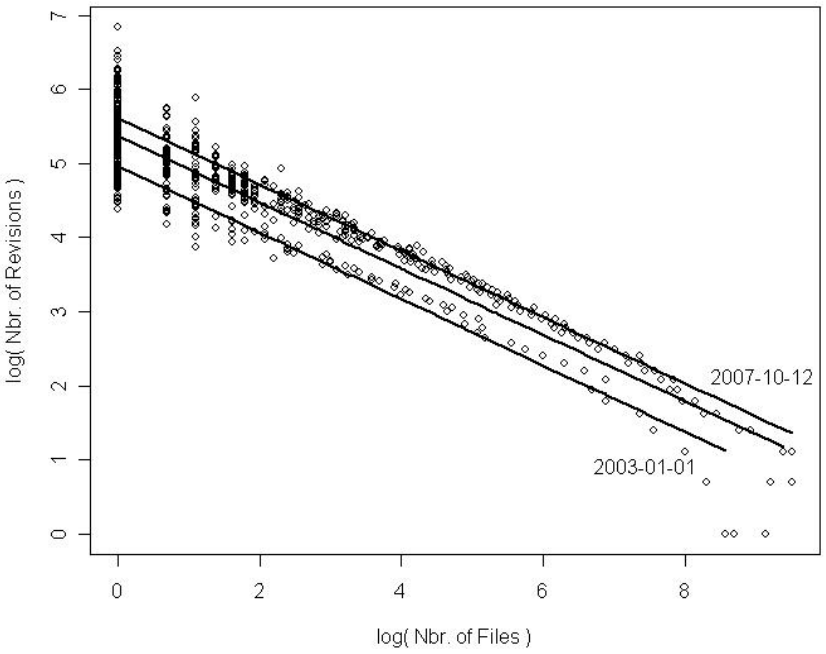

Figure 3 - Evolution of File Revision Distribution

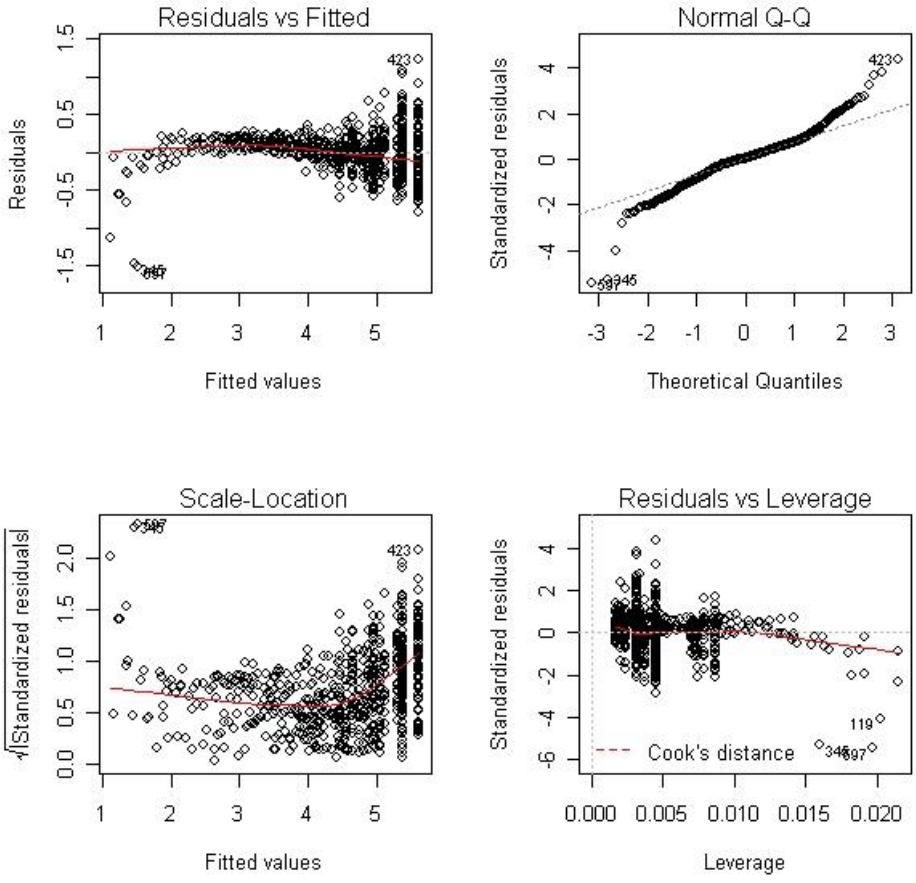

Figure 4 - File Revision Model Residuals

The remainder of variability arises from that region having 50 or more file revisions. This constitutes some 1.5% of all files in the code base. This region contributes to the fan-out characteristics visible in the top left plot. Typically, this type of observation can indicate heteroscedasticity or a difference in the variance of the model's error terms. One of the more common manifestations of this phenomenon is when the magnitude of the error term's variance is related to its associated dependant variable or one of the independent variables. In this instance, it is taken to be further evidence that an additional perturbing influence is present on the higher variability files.

5.2 Analysis of File Revisions

The time series for file check-ins for the Eclipse project is shown in Figure 5. The series shown is measured in days from 2001-04-28 and is plotted against the base 10 logarithm of the number of check-ins for the day in question. A LOWESS locally weighted polynomial regression for the data has been computed and is shown on the graph as a solid line. Also plotted is a LOWESS regression of the maximum depth of inheritance of the associated check-in. The diagram plots this function as a linear relationship to the y-axis.

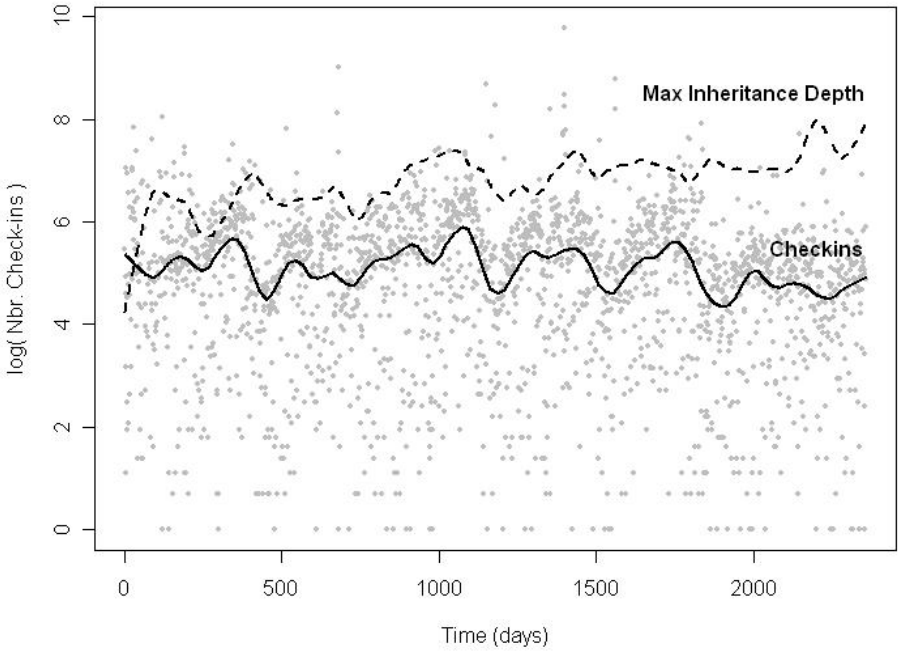

Figure 5 - Time Series of File Check-ins

A cyclical component to the time series of the number of files checked-in is evident from the plot. Its period of approximately 300 days would strongly suggest that the component is seasonally based and not necessarily due to code related factors. This however has not been fully investigated. Also in evidence is a slightly reduced rate of files committed in recent years though the affect is marginal. The code also appears to exhibit higher maximum depth of inheritance for more recent committals.

Measure	Value
Spearman's Rho	0.131 (0.131)
Kendall's Tau	0.095 (0.095)

Table 2 – Lowess Correlation

Caution is advised when drawing conclusions from this last observation. It is important to appreciate that max inheritance depth graphed is the result of a polynomial smoothing of a rather crude aggregate statistic. This value is the local maximum of the depth of inheritance as limited to the source code committed at that point in time. It does not provide any insight into the distribution of inheritance depths in the committed code. Nor does it necessarily accurately reflect the state of the repository as a whole. A statistical analysis addressing these shortcomings will be presented in a following paper.

For the present, a statistical correlation was computed between the two smoothing functions and is presented in the table below. Both indicate that the relationship between their variance is positively correlated but low.

6. Discussion

Apparent from the initial analysis is that the code base as a whole is evolving in a deterministic fashion. One might conclude therefore that findings arrived at for a specific period in the project lifetime can be extended to the project as a whole. The clear relationship between the distribution of file revisions across the project was something of a surprise. It would be interesting to confirm if this effect is present in other source repositories. Such a relationship could comprise a useful estimator of the level of future change.

This distribution can be described by a linear function in the log/log domain and accounts for 93% of the revision variance within the project. Particularly intriguing is the group of files towards the y axis in Figure 2 and Figure 2 having 60 or more revisions. These files comprise approximately 1.5% of the total source files and do not conform as readily to the linear model as the remaining code. A tantalising possibility is that the code in these classes has a higher level of cognitive dissonance though this has not been investigated further.

An investigation of the check-in history of files suggests that cyclical effects similar to that predicted by Dvorak are in evidence. However upon further analysis, the period of the cycles would indicate that this characteristic is most likely the result of seasonal effects. Furthermore, this cyclical signature could not be strongly correlated with class inheritance depths of the associated committals; though this analysis was confined to a rather crude maximum depth of inheritance metric.

7. Threats to Validity

The experimental design employed by this study was constructed so as to avoid laboratory scenarios and to observe software maintenance by professional developers. Whilst the authors believe that they have achieved a high degree of ecological and external validity, they also concede that no experiment is perfect. The author's have identified several potential threats to the validity of this study.

This study is directed towards a single subject; the Eclipse source code base. Limiting observations to one body of source code reduces both the effectiveness of conclusions drawn and their scope for external application. For example, an observation of low inheritance predominance may lead to the conclusion that the effect is the result of strategies exercised by the independent programmer. This supposition may be incorrect if some external agent (software quality assurance policy for example)

is discouraging its use. The obvious mitigation to this threat is to confirm this study's findings in other code bases.

A second threat results from the nature of the Eclipse's open source development methodology. The extension of conclusions based on the analysis of an open source project to all other development paradigms may not be valid in general terms. Indeed there is some evidence that growth patterns and code cloning in open source projects is atypical to proprietary source code evolution (Godfrey et al. 2000)(Godfrey et al. 2001). Conclusions should be qualified when applying these findings to other methodologies.

The last threat identified concerns the internal experimental validity in relation to the nature of the custom tools used in the analysis. Firstly, the tools were bespoke developed for the analysis and as such they are immature; as yet unidentified errors in their construction may influence the study's analysis and findings. Secondly, the tools are required to make certain inferences in relation to code linkages. This is an unavoidable consequence of the approach taken; the alternative being compilation, linkage and installation of every version of every project according to that project's build rules (which would be prohibitive). Naming collisions can and do occur. The author's have attempted to mitigate this threat by confining the analysis to those classes that have been fully disambiguated in terms of inheritance linkages. The authors believe that this should eliminate any potential threats in this regard.

8. Conclusion

This paper presented a study on the evolutionary traits of the Eclipse open source project. It concentrates on file revision traits and in particular whether a predicted cyclical effect is in evidence. Such an effect could signify readjustment instigated by a critical accumulation of cognitive dissonance within the source code. No such effect was observed; any cyclical observations identified were considered most likely due to seasonal effects.

A strong relationship governing the distribution of file revisions was found to be in evidence. This relationship seems to be stable over time and may provide a predictive tool for estimating future code modification. To the author's knowledge this has not been documented before and it is important to establish whether this effect is present in other code bases.

This study was inspired by several prior works which have investigated the issue of inheritance impact on code maintenance. The experimental design employed by this study was chosen with a view to reducing perceived weaknesses in the approaches of some of these previous studies. By employing repository mining techniques on a large well managed project, the authors hope to achieve a high degree of external and ecological validity to these conclusions.

While the question of whether class inheritance depth affects maintenance effort is an old one it is still relevant. Indeed, the ascendance of development methodologies favouring lightweight processes and self documenting code suggest that this is true more so today than when the question was first considered some 15 years ago. It is hoped that this body of work goes some way in furthering this debate.

9. Acknowledgments

A. Creegan would like to thank Dr. P. O'Shea for her help and suggestions in relation to this research. He would also like to thank Mike Meaney and Dean Butler for making server hardware available to perform the analysis.

10. References

Chidamber, S.R.; Kemerer, C.F. (1994), "A metrics suite for object oriented design," *IEEE Trans. on Software Engineering*, vol.20, no.6, pp.476-493, June.

Daly, J.; Brooks, A.; Miller, J.; Roper, M.; Wood, M. (1995), "An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software", *Empirical Software Engineering: An International Journal*.

Dunsmore, A.; Roper, S.; Wood, M. (2000), "Object-Oriented Inspection in the Face of Delcalisation", *International Conference on Software Engineering – ICSE'00*, pp. 467-476.

Dvorak, J. (1994), "Conceptual Entropy and its Effect on Class Hierarchies", *IEEE Computer*, pp. 59-63, June.

Erlikh, L. (2000), "Leveraging legacy system dollars for e-business," *IT Professional* , vol.2, no.3, pp.17-23, May/June.

Fioravanti, F.; Nesi, P. (2001), "Estimation and Prediction Metrics for Adaptive Maintenance Effort of Object-Oriented Systems", *Software Engineering, IEEE Transactions on*, vol. 27, no. 12, pp. 1062-1083, December.

Godfrey, M. W,; Qiang, Tu (2001), "Growth, Evolution, and Structural Change in Open Source Software," *IWPSE 2001*, ACM.

Godfrey, M. W.; Qiang Tu (2000), "Evolution in open source software: a case study," *Software Maintenance, 2000. Proceedings. International Conference on* , vol., no., pp.131-142.

http://archive.eclipse.org/arch/, March 2008.

http://www.antlr.org, March 2008

http://www.antlr.org/grammar/1152141644268/Java.g, March 2008

http://www.eclipse.org, March 2008

Lehman, M. M. (1980), "Programs, life cycles and laws of software evolution," *Proceedings of IEEE*, vol. 68, issue 9, pp. 1068-1076, September.

Li, W.; Henry, S. (1993), "Maintenance metrics for the object oriented paradigm," *Software Metrics Symposium, 1993. Proceedings., First International*, vol., no., pp.52-60, 21-22 May.

Meyer, B. (1988), "Object oriented software construction", ISBN: 0-13-629049-3, Prentice Hall Int., Hertfordshire, England.

Prechelt, L.; Unger, B.; Philippsen, M.; Tichy, W. (1998), "Re-Evaluating Inheritance Depth on the Maintainability of Object-Oriented Software", *Submission to Empirical Software Engineering: An International Journal*.

Romero, P.; du Boulay, B. (2004), "Structural knowledge and language notational properties in program comprehension", *Proceedings of the 2004 IEEE Symposium on Visual Languages and Human Centric Computing (VLHCC'04)*, IEEE Computer Society.

Soloway, E.; Pinto, J.; Letovsky, S.; Littman, D.; Lampert, R. (1988), "Designing documentation to compensate for delocalized plans", *Communications of the ACM*, Vol. 31, No. 11, pp. 1259-1267.

von Mayrhauser, A. (1995), "Program Comprehension during Software Maintenance and Evolution", IEEE Computer, pp. 44-55, August.

Wilde, N.; Huitt, R. (1992), "Maintenance Support for Object-Oriented Programs", *Software Engineering, IEEE Transactions on*, Vol. 18, No. 12, pp. 1038-1044.