Available online at www.sciencedirect.com
SCIENCE DIRECT®
The Journal of Systems and Software 72 (2004) 179–186
The Journal of Systems and Software
www.elsevier.com/locate/jss

Information theory-based software metrics and obfuscation

Steven R. Kirk [1], Samantha Jenkins [1]

Department of Informatics and Mathematics, HTU, P.O. Box 957, Trollhättan 461 29, Sweden

Received 20 January 2003; received in revised form 29 April 2003; accepted 9 May 2003

Abstract

A new approach to software metrics using concepts from information theory, data compression, complexity theory and analogies with real physical systems is described. A novel application of software obfuscation allows an existing software package to be analysed in terms of the effects of perturbations caused by the obfuscator. Parallels are drawn between the results of the software analysis and the behaviour of physical systems as described by classical thermodynamics.

Keywords: Software metrics; Information theory; Data compression; Kolmogorov complexity; Obfuscation; Entropy; Thermodynamics

1. Introduction

The use of software metrics to inform and guide almost all aspects of the software development process is now widespread. Hundreds of possible measures for software have been proposed, with varying degrees of theoretical and empirical justification, and have been incorporated into popular software development models such as COCOMO. There has, however, in recent years been a move towards creating a more rigorous theory of software measurement, with the aim of moving software engineering more in the direction of a true engineering science (Zuse, 1998; Fenton and Neil, 1999). The work described in this paper is intended, as a starting point of a longer-term effort, to assist in achieving this goal, informed by analogies from the physical sciences.

Research into the possibility of employing information theory in the definition of software metrics first appeared in the 1970s, with Hellerman (1972). Later, Halstead analysed software in terms of numbers of unique operators and operands, along with the total numbers of operators and operands (Halstead, 1997). His derived metrics, however, were rooted more in psychology than pure computer science, and have been extensively criticized (e.g. Jones, 1994). In the 1990s in particular, others have proposed metrics based on principles from information theory (Khoshgoftaar and Allen, 2001; Bansiya et al., 1999; Etzkorn et al., 1999), mainly intended as maintainability metrics. These metrics are generally based on the symbol entropy from classical information theory (Shannon, 1948), but with the 'symbols' comprising individual whole keywords and identifiers from the source code rather than individual characters. We will use the more familiar character symbol entropy (see later) in our investigations.

Our goal in this work is to investigate the properties of our new information theory-based software complexity measures, compare them with those of an existing (and well-established) software design metric, and investigate the possibility of using analogies with the thermodynamics of physical systems to discuss the evolution of software during the development process, with the goal of optimizing this process. Section 2 of this paper describes the concept of Kolmogorov complexity from information theory, an approximation of which we use as a basis for our software complexity measures. The concept of obfuscation is also explained, and its use as a post hoc method for increasing the complexity for a given piece of software outlined. One particular existing (and widely used) software design metric, which is used as a basis for comparison with our information theory-based metric, is then described. Our methods are described in detail in Section 3 and our results are shown

E-mail addresses: steven.kirk@htu.se (S.R. Kirk), samantha.jenkins@htu.se (S. Jenkins).
[1] Tel.: +46-520-475318; fax: +46-520-475399.

0164-1212/$ - see front matter © 2003 Elsevier Inc. All rights reserved.
doi:10.1016/S0164-1212(03)00217-6

in Section 4. Section 5 contains a discussion of our results, Section 6 contains some suggestions for future investigation, and our conclusions are given in Section 7.

2. Theory

The concepts which we will use in later discussion in this paper are defined below.

2.1. Kolmogorov complexity

The Kolmogorov complexity $K(x)$ of a string x is defined to be the length of the shortest binary program to compute the string x on a universal Turing machine (Li and Vitanyi, 1997). The string x can represent a literal string of text, or more generally the contents of any type of computer file, including executables. $K(x)$ can also be regarded as the minimum information needed to generate x by some computational process, and represents the absolute information of an individual object, as opposed to the average information over a random ensemble as dealt with by classical information theory. In a formal mathematical sense, however, $K(x)$ is non-computable and non-unique. Chaitin (1996) has used a modified version of LISP and a modified version of a universal Turing machine to make $K(x)$ computable, but the approach used in most studies involves creating a computable *approximation* to $K(x)$. One such approximation, already used in analysis of natural languages and mitochondrial DNA sequences (Li et al., 2001), is the size of the input data when compressed by a lossless data compression technique. This approach has been followed in this work (all following references to complexity in this paper refer to this approximation for K), and is described in detail in Section 3.4.2.

2.2. Obfuscation

The term 'obfuscation' in a computer science context refers to a method of modifying existing software, with the goal of generating software which, though functionally equivalent to the original from a user's perspective, is much more difficult for a programmer to reverse-engineer. Many software packages exist which will obfuscate C/C++, Java and more recently C# programs. The Java language was used in this study due to its relative simplicity compared with C++, its object-oriented nature and its widespread use as a means of generating architecture-neutral portable software. The latter attribute makes Java programs particularly straightforward to reverse-engineer, which may, depending on context, be undesirable to the original software developer.

One can distinguish three general categories of obfuscation, namely

- Name obfuscation—longer meaningful human-readable names for classes, methods and variables, which are often left embedded in the program executable (or class files, in the case of Java) can be replaced with short cryptic names.
- Flow obfuscation—the flow of control through methods can be altered by modifying conditionals, loop constructs, etc. to yield a more 'spaghetti-like' structure. While the Java language has no `goto` statement, the Java Virtual Machine bytecode instruction set does include a `goto` instruction. It is possible, therefore, to generate bytecode for which there is no simple corresponding Java source code.
- Structural obfuscation—the class and inheritance hierarchy of a program can be modified.

The latter category of structural obfuscation has not been widely incorporated in commercially available obfuscation software, although many different techniques for structural obfuscation have been envisaged and categorized (Collberg et al., 1997).

Obfuscation, therefore, provides an automated post hoc method for increasing the complexity of an existing piece of software—in general, the greater the degree of obfuscation, the greater will be $K(x)$. In this way, the effects of obfuscation provide a benchmark against which the complexity deliberately added by the software developer during the normal development process may be compared. If the results of obfuscation are taken to be the 'worst-case scenario' in terms of added program complexity, the developer can be informed if the complexity added due to their efforts approaches or exceeds that added by the obfuscator software, thus serving as an early warning of excessive program complexity, and hence future maintainability problems. This strategy is potentially useful for 'incremental' software development philosophies such as Extreme Programming (XP) (Karlström, 2001), where the changes between one development version of software and the next are usually small.

2.3. Design metrics

A recent popular approach for extraction of design metrics from object-oriented software has been developed by Martin (2002). At the package level, the *afferent* (Ca) and *efferent* (Ce) couplings are defined as

> Ca = The number of other packages that depend upon classes within the package,
> Ce = The number of other packages that the classes in the package depend upon

and a measure of the package *instability* I can be computed as Ce/(Ce + Ca), giving a range of values for I between $I = 0$ (completely stable package) and $I = 1$ (completely unstable package). This metric is an indi-

cator of the package's resilience to change, and forms the basis of comparison with our information theory-based metrics (see later). These particular metrics were chosen with a view to comparing the concept of design instability with an analogous concept based on thermodynamic arguments (see Section 5). The methods used to extract the design metrics from our test data sets are described in Section 3.4.1.

3. Methods

Our goal, therefore, is to calculate the design metrics and our new information theory-based metrics for a test data set of software and evaluate the effect of obfuscation of our test data set. The methods we employed are described below.

3.1. Data sets

A data set comprising four student projects written in Java was chosen. All of the projects implemented a graphical user interface to display various quantities measured by a weather station (air pressure, wind speed, humidity, etc.). An additional open-source package (JUnit, a unit testing package for Java programs (Erich Gamma, Kent Beck, JUnit version 3.8, http://www.junit.org), consisting of 101 classes) was added to the data set for the purpose of comparison.

3.2. Handling of JAR files

The obfuscator software used in this work (see next section) operates on (and outputs) Java archive (JAR) files, so the classes created in each of the student projects were packaged as executable Java archives. By default, JAR files are created with 'zlib' compression (Jean-loup Gailly, Mark Adler, The 'zlib' compression library, http://www.gzip.org/zlib/) applied individually to each of the class files enclosed in the archive. In order to provide a basis for comparison, therefore, all of the JAR files were unpacked and repacked with no 'internal' compression applied.

3.3. Obfuscation program

The obfuscator used in this study was Zelix KlassMaster (version 3.1.3, Zelix Pty Ltd., http://www.zelix.com). Controllable either using a GUI or a scripting interface, this software can perform name and flow obfuscations, and can also optimize JAR files to remove redundant methods, inner classes, etc. Name obfuscation can be independently applied to public, private, package and protected classes, fields and methods, with flexible inclusion and exclusion rules. Flow obfuscation (optional) is implemented in three levels: 'light', 'normal' and 'aggressive', corresponding to the cumulative application of three different proprietary flow obfuscation techniques (Svet Kovich, personal communication, Zelix Pty Ltd.). Many other refinements, such as encryption of string literals, are possible using the software but were not used in this study.

3.4. Calculation of metrics

3.4.1. Extraction of design metrics

Since the authors are not aware of any currently available software which will calculate software metrics directly from JAR files or raw bytecode, it was only possible to collect design metrics from the original (unobfuscated) Java source code. The metrics software used in this project was RefactorIT (Aqris Software, http://www.refactorit.com/). Ideally, a decompiler (to generate valid Java source code) would be used to allow the metrics software to extract the maximum possible information from the obfuscated forms of the test programs. Flow obfuscation, however, generated bytecode that had no valid source code equivalent, causing decompilers to fail. For this reason, the design metrics were only calculated for the original unobfuscated forms of the test programs.

3.4.2. Extraction of complexity and entropy data

Additional information about the JAR files were obtained by recording their sizes in bytes, and also the size of the output files produced when the JAR files were compressed using the compression software 'bzip2'—the latter size was used as an approximation for the Kolmogorov complexity K. 'bzip2' was chosen over the common alternatives 'gzip' and 'zip' as it gave, on average, smaller compressed files than these alternatives for the data files used in this work, and hence better approximations for K. In addition, the symbol entropy (where the symbol size was taken to be 1 byte or character) and serial correlation coefficients (the average probability of successfully predicting byte $n+1$ given byte n) of the JAR files were calculated using the 'ent' program (John Walker, ENT—A pseudorandom number sequence test program, http://www.fourmilab.ch/random/).

4. Results

Design metrics were first extracted from the original source code, and the original contents of each JAR file repacked into another JAR file with no individual 'zlib' compression. A 'bzip2'-compressed version of the repacked JAR file was created using the maximum compression level obtainable. This step allowed the compression software 'bzip2' to find the maximum number of exploitable similarities within the JAR file,

including similarities between classes, to achieve maximum compression. In all of the further description of the results in this paper, the acronym UJS is used to refer to the size in bytes of the uncompressed (repacked) JAR file, and BJS refers to the size in bytes of the 'bzip2'-compressed repacked JAR file.

The obfuscator software was then run multiple times with each original input JAR file, each time with different combinations and degrees of either name and flow obfuscation, but not both. The resulting output JAR files were then analysed using the procedure mentioned above to get UJS and BJS. In addition, the 'ent' program was run on each repacked version of the output JAR files to obtain the character symbol entropy of the file measured in bits per byte. For each original input JAR file, one output JAR file was produced in which the obfuscator did no actual obfuscation, but simply 'optimized' the file by removing redundant classes, line number tables, etc. This particular output JAR file was labeled 'opt' to denote 'optimization only', and was smaller than the original input file. This optimization was also applied by default to all the obfuscated output files. Due to the unavailability of the source code for the obfuscation software used in this project, we cannot be certain that all of the transformations carried out fit the formal description of equivalence-preserving transformations, but we confirmed that the obfuscated versions of the test programs operated identically and looked identical to the original programs from the users point of view.

The results of all of these calculations are shown as graphs in Figs. 1 and 2. Graphs of $\mathrm{BJS/BJS_{opt}}$ vs. $\mathrm{UJS/UJS_{opt}}$ (which we shall call *compression graphs*) and the entropy density (character symbol entropy in bits per byte, divided by 8, and denoted by ENTBPB) vs. $\mathrm{UJS/UJS_{opt}}$ (which we shall call *entropy density graphs*) were plotted for each input JAR file. All of the data on all of the graphs were fitted to fourth-order polynomial curves. In the following discussion, we will also denote the point on compression graphs where $\mathrm{UJS/UJS_{opt}} = 1$ and $\mathrm{BJS/BJS_{opt}} = 1$ as the *reference point*. The results are also summarized in Table 1.

5. Discussion

5.1. Examination of compression and entropy density graphs

As expected, using increasing degrees of flow obfuscation generated data points on the compression graph which were further to the right of the reference point, indicating that increasing the level of flow obfuscation increased the size of the JAR files. The rightmost points therefore correspond to 'aggressive' flow obfuscation. Similarly, increasing the amount of name obfuscation generated points on the same graph which were further to the left of the reference point, with full name obfuscation generating the leftmost points on the graphs. This corresponds to all names within the JAR file having their minimum possible length.

In Table 1 the quantity $(y_{\mathrm{ref}} - y_{\mathrm{Bmin}})$ gives the difference between the y-components of the reference point and the minimum in the compression graph. The symbol x_n is the x-coordinate of the data point corresponding to full name obfuscation. I_{av} is the design instability metric I averaged over all the packages in the tested code, and finally 'Rank' is the position that the projects have been judged to deserve, 1 being the best and 5 being the worst, and is shown for both the information theory-based metrics and the established design metrics. There is a compelling correspondence between the rankings according to these two different types of software measurement; the slight difference in ordering between the projects WSP2 and JUnit can be accounted for by observing that I_{av} is an average for each project, and the observed scatter in the individual package values of I for each of these projects is more than enough to account for the slight misordering.

Note that weather station project 1 (WSP1) has its $(y_{\mathrm{ref}} - y_{\mathrm{Bmin}})$ values marked as NA, as this data set has no obvious minimum. The lack of a minimum is an indication of the lower quality of the project, indicating that this project is not optimized and is unstable. This is also shown by the corresponding tabulated value of the design metric I_{av}.

The value of $(y_{\mathrm{ref}} - y_{\mathrm{Bmin}})$ directly relates to 'Rank'; the smaller the value of $(y_{\mathrm{ref}} - y_{\mathrm{Bmin}})$ the higher the rank, which reflects the level of optimization of the JAR files contained in the projects. The quantity x_n is an indication of variable name lengths used in the projects, so the larger the value the shorter the variable names. The extreme values of x_n only are in agreement with the rank, showing that there are more important effects than just length of variable names.

5.2. Thermodynamic interpretation

Large software projects evolve over time, both after deployment and during the development process. As this dynamic process evolves, certain parts of the software will become less flexible, more fault-prone and more difficult to re-engineer. These failing components may have an adverse effect on other parts of the software system and may in the extreme case cause the system to fail. There is a compelling analogy between this behaviour and phase changes in a physical system, which can be described in terms to thermodynamics. For example, a strong analogy can be made between the software development process and the industrial production of a metal alloy designed to have specific technical material properties. On cooling from a molten liquid, small

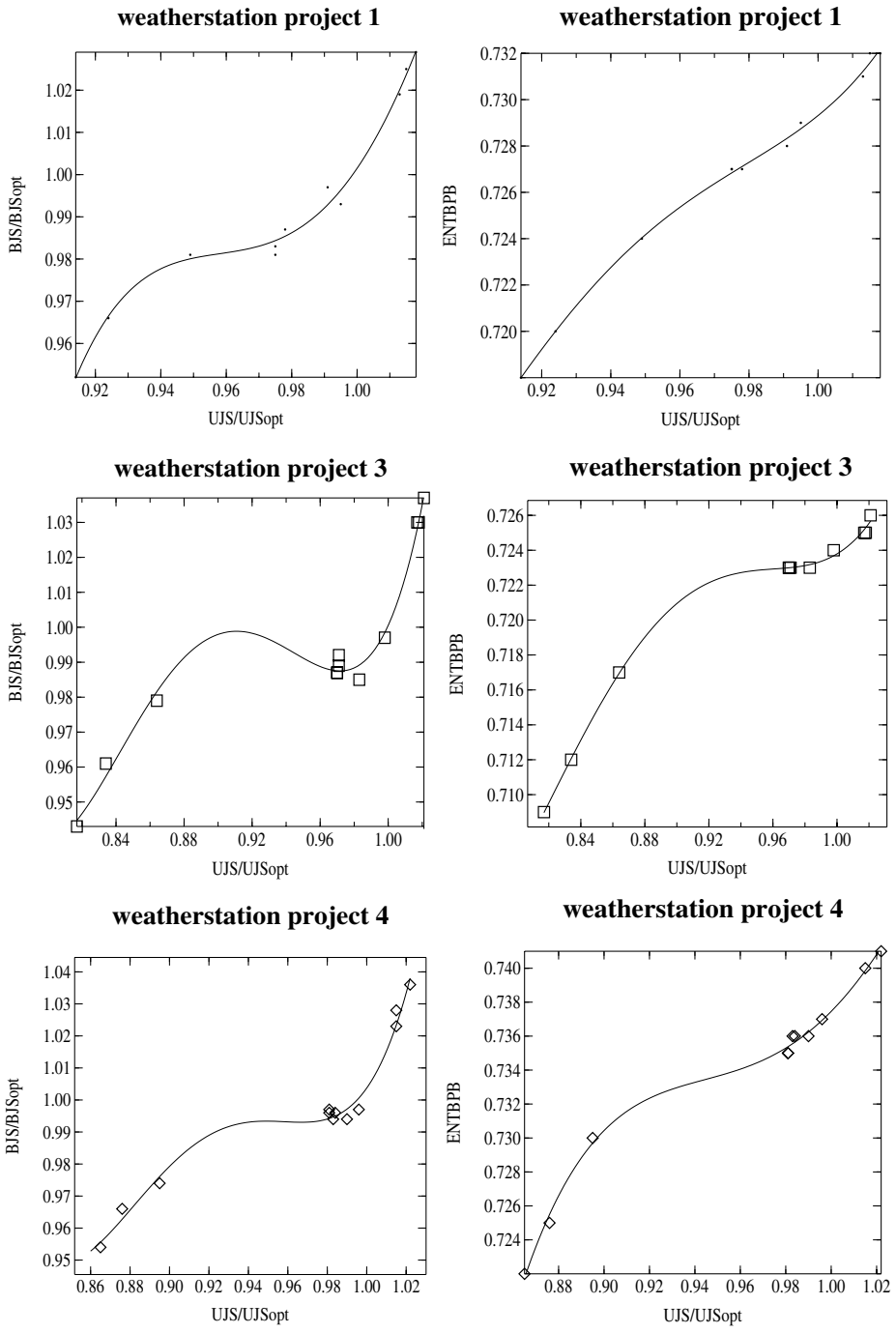

Fig. 1. Graphs of BJS/BJS_{opt} vs. UJS/UJS_{opt} (left) and entropy density (right) against UJS/UJS_{opt} for weather station projects 1–3.

'crystallites' with known compositions may form and grow, until the system becomes solid (here we may equate 'crystallites' with software classes or packages). This is usually only the start of the alloy production process, however, and the further 'annealing' and temperature processing used to encourage the growth of 'crystallites' with specific desirable properties and sizes at the expense of other types and sizes of 'crystallite' can be compared directly with the processes of optimization, refactoring, addition of features, etc. which comprise the software development process. The analogy is especially apt for modern 'agile' or 'extreme programming' software development methodologies, in which the software is continuously tested, refactored and redesigned as the development process proceeds. The same mathematical framework which predicts, according to the laws of thermodynamics, the physical properties and microstructure of the alloy, may therefore also be

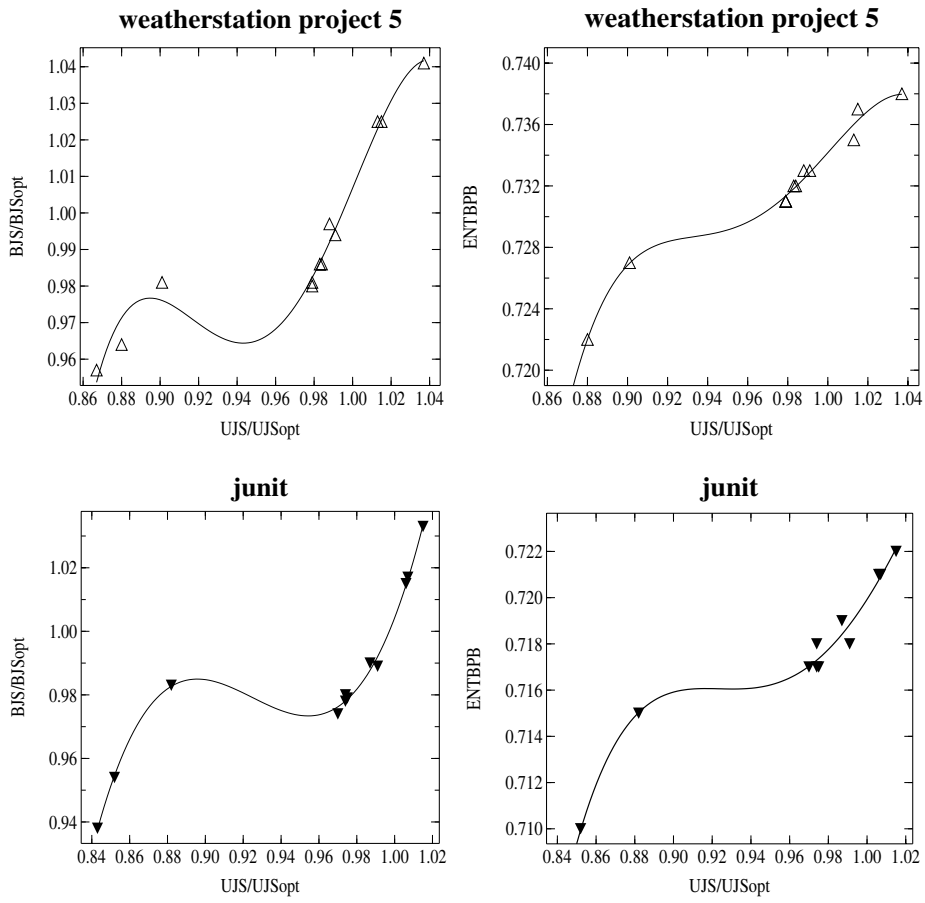

Fig. 2. Graphs of BJS/BJS_{opt} vs. UJS/UJS_{opt} (left) and entropy density (right) against UJS/UJS_{opt} for weather station project 4 and JUnit.

Table 1
Software rank table

Project	$(y_{ref} - y_{Bmin})$	x_n	I_{av}	Rank $(y_{ref} - y_{Bmin})$	Rank (I_{av})
WSP1	NA	0.91	1.00	5	5
WSP2	0.017	0.82	0.91	2	3
WSP3	0.007	0.87	0.83	1	1
WSP4	0.036	0.87	0.98	4	4
Junit	0.023	0.84	0.87	3	2

The weather station projects are denoted by WSP. Other quantities are defined in Section 5.1.

applicable to the creation and development of computer software.

In this section we explain our results by identifying parallels between measurable physical quantities (pressure, temperature, volume, entropy, etc.) used in the classical theory of thermodynamics (treated in some depth in Waldram, 1989) and our information theory-based software metrics. These parallels are outlined in Table 2, and are used in the following discussion.

Examination of the compression graphs with the quantities in Table 2 shows that as the plot is followed downwards towards x_n, the rate of change of the 'pressure' ∂P increases. Travelling towards the centre of the plot where the curve is flat in most cases (i.e. gradient is zero) or close to flat, then ∂P is close to zero. The gradient of the entropy density graph is proportional to the 'pressure' P for constant 'temperature' T; this is why there is generally excellent correspondence between the x-coordinates of the minimum in the compression and entropy density graphs (see Table 1). In terms of practicalities it can be seen that less effort will be required to move along the compression graph where ∂P is close to zero than when it increasing (as it does when moving towards x_n away from the centre of the plot). This occurs for larger values of BJS/BJS_{opt} and where the gradient is zero or very small.

Since projects with reference points much above the minimum e.g. weather station project 4 (WSP4) are considered to be relatively poorly written, then relatively little effort should be required to move them closer to the

Table 2
Relationships between derived software metrics and quantities in classical thermodynamics

Derived software metrics	Thermodynamics
BJS/BJS_{opt}	$1/(\partial P/\partial V)_T$
UJS/UJS_{opt}	V
ENTBPB	S
Gradient of BJS/BJS_{opt} vs. UJS/UJS_{opt}	$\partial/\partial V[(1/V)\partial V/\partial P]_T$
Gradient of ENTBPB vs. UJS/UJS_{opt}	$(\partial S/\partial V)_E = P/T$

$(\partial P/\partial V)_T$, V, S, T and E refer to the compressibility, volume, entropy, temperature, and energy, respectively.

reference point, since the y-coordinate increases more quickly than the x-coordinate. Nevertheless, it should still be reasonably straightforward to move the more efficiently written projects (WSP2 and WSP3) along the graph where the gradient is small, the goal being to produce a smaller, 'denser' project which will have a lower entropy.

Examination of the entropy density graph shows that the entropy S decreases as x_n is approached, a reduction in entropy representing an increase in order of the system. It is more difficult to reduce entropy than to increase entropy, so it is to be expected that more work is required to produce a better project. A project with lower entropy is likely to be more maintainable and reusable.

6. Future work

This study suggests some possible directions for further work—these are summarized below.

- Rewriting some of the weather station projects using shorter class, method and variable names. This procedure is expected to produce a closer approach of the reference point to a minimum (or close to zero gradient point) in the compression graph. This procedure will allow a more detailed investigation of the properties of this region of the graph. Changing the reference point in this way will enhance the favourable features in the entropy density graphs and compression graphs, e.g. removing the maximum near the origin of the entropy density graph and 'lowering the barrier' to potential code improvements.
- Flow obfuscation will be combined with name obfuscation simultaneously. Preliminary tests have indicated that applying both full name obfuscation and increasing degrees of flow obfuscation generates points on the compression graph which indicate a trend back towards the right which does not coincide with the pure name-obfuscation curve. This behaviour is reminiscent of the phenomenon of *hysteresis loops* in magnetism. Applying the classical thermodynamics analogy again, it is possible to calculate the area in the loop as the work done, so the smaller the area of the loop, the better the quality of the project.
- Formalise the classical thermodynamic theory of software metrics (developed in this work), and extend it to a statistical thermodynamic theory of software metrics. This may be achieved by making close analogies between atoms and software objects, i.e. using the formalism of statistical thermodynamics to describe assemblies of interacting software objects, instead of physical systems consisting of interacting atoms.

The process of software refactoring is in many ways the opposite of the process of obfuscation. The effect of refactoring on the information theory-based metrics introduced in this paper warrants further investigation.

7. Conclusions

This work has indicated the usefulness of applying some of the vast well-developed theoretical base of the physical sciences to the measurement of software. Using the concepts of lossless data compression, obfuscation and the Kolmogorov complexity, we have drawn close parallels between our information theory-based software complexity metrics and measurable quantities in real physical systems. Obfuscation has proved to be very useful in creating the software equivalent of the 'energy surface' associated with physical systems, i.e. the basis for forming a quantitative understanding of how the system changes when perturbed. This was important in forming an absolute quality (and maintainability) judgment on the all of the projects and packages. We have also demonstrated some information theory-based metrics, confirmed that they predict similar trends in a software test data set as established design metrics, and demonstrated the potential of analysis of software using an approach inspired by the classical thermodynamics of physical systems.

These methods allows early assessment of the design quality of a software project using a twofold approach: firstly by examining the proximity of the reference point to any minima, and secondly by inspection of the entropy density plot; the lower the values of the entropy density the better, and the absence of any maxima in the entropy density graph is also a favourable sign. These methods show promise in the application of monitoring the software development process, with the goal of providing an early warning of potential future problems.

Acknowledgements

The authors would like to thank the KK Foundation for funding the project (Grant no. 2001/204), Stefan Mankefors for his helpful comments on the manuscript, Richard Torkar for discussions and information, and Christer Selvefors for providing the student project data for analysis.

References

Bansiya, J., Davis, C., Etzkorn, L., 1999. An entropy-based complexity measure for object-oriented designs. Theory and Practice of Object-Oriented Systems 5 (2), 111–118.

Chaitin, G.J., 1996. How to run algorithmic information theory on a computer. Complexity 2 (1), 15–21.

Collberg, C., Thomborson, C., Low, D., 1997. A taxonomy of obfuscating transformations, Technical Note #148, July 1997. Available from <http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html>.

Etzkorn, L., Bansiya, J., Davis, C., 1999. Design and code complexity metrics for OO classes. Journal of Object-Oriented Programming (March/April).

Fenton, N.E., Neil, M., 1999. Software metrics: successes, failures and new directions. Journal of Systems and Software 47, 149–157.

Halstead, M.H., 1997. In: Elements of Software Science, Operating, and Programming Systems Series, vol. 7. Elsevier, New York, NY.

Hellerman, L., 1972. A measure of computational work. IEEE Transactions on Computers 21 (5), 439–448.

Jones, C., 1994. Software metrics: good, bad, and missing. Computer 27 (9), 98–100.

Karlström, D., 2001. Introducing Extreme Programming—An Experience Report, SERP'01, October.

Khoshgoftaar, T.M., Allen, E.B., 2001. Empirical assessment of a software metric: the information content of operators. Software Quality Journal 9, 99–112.

Li, M., Badger, J.H., Chen, X., Kwong, S., Kearney, P., Zhang, H., 2001. An information based sequence distance and its application to whole mitochondrial genome phylogeny. Bioinformatics 17 (2), 149–154.

Li, M., Vitanyi, P., 1997. An Introduction to Kolmogorov Complexity and Its Applications, second ed. Springer-Verlag.

Martin, R.C., 2002. Agile Software Development, Principles, Patterns and Practices. Prentice-Hall.

Shannon, C.E., 1948. A mathematical theory of communication. Bell System Technical Journal 27, 379–423, and pp. 623–656.

Waldram, J.R., 1989. The Theory of Thermodynamics. Cambridge University Press, Cambridge.

Zuse, H., 1998. A Framework of Software Measurement. Walter de Gruyter, Berlin, New York.