Stability and Volatility in the Linux Kernel

John Champaign, Andrew Malton, Xinyi Dong
School of Computer Science
University of Waterloo
{jchampaign,ajmalton,xdong}@uwaterloo.ca

Abstract

Packages are the basic units of release and reuse in software development. The contents and boundaries of packages should therefore be chosen to minimize change propagation and maximize reusability. This suggests the need for a predictive measure of stability at the package level. We observed the rates of change of packages in Linux, a large open-source software system. We compared our empirical observations to a theoretical 'stability metric' proposed by Martin. In this case, we found that Martin's metric has no predictive value.

1 Introduction: Packaging and Software Change

In software development terminology, a *package* is a "basic development unit which can be separately created, maintained, released, tested, and assigned to a team" [1, Chap. 7]. *A large software system can be split into smaller units, called packages, that make system comprehension, customer delivery, maintenance and continued development easier*. The focus on maintenance and release management (including testing) distinguishes packaging from other software structuring concerns, such as modules, classes, subsystems, program families, and namespaces. In practice a package is a subset of the source files of the system, and a *packaging* is a partitioning of the set of source files. If we have (or have developed) a software system encoded in a set of source files, then there arises the question of how to package them, that is, what packaging to choose. In general there are exponentially many distinct partitionings of a set of n elements[1]. The appropriate choice is based on how packages depend on each other and how they change from release to release. A packaging determines a dependency relation between packages, induced by the dependency relation between source files. Any change to a source file requires a new release to the package it belongs to.

In his recent book [2], R. Martin of OMI Inc. adds 'reused' to the above list of properties of a package. He points out that dependency management during reuse, especially reuse of multi-use packages, must be combined with release management. Therefore we should choose a packaging so that

1. each package is reusable,
2. no package contains two or more reusable subsets, and
3. the package dependency relation is well-founded (cycle-free).

The first prescription is simply supporting reuse at the package level; the second is meant to maximise flexibility, and the third of course enables packages to be released independently.

In order to minimize change propagation, it's helpful to distinguish between the empirical fact of change and the intent or purpose of change as a design evolves. Some software units are designed to change often; others are expected to change rarely. Developers should be conscious of these intents and reflect them in the overall design and documentation. During evolution and maintenance, some software units change often in fact; others change rarely. Developers should be aware of the rate of change and manage packaging and release accordingly.

Martin designates packages which are intended to be

[1] The number of distinct partitionings is called the *Bell number*. It is between 2^n and $n!$. There is no good closed form for it.

Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)
0-7695-1903-2/02 $17.00 © 2002 IEEE

easy to change as *unstable*. (We would prefer a term that is less suggestive of poor quality – such as *soft*.) Examples of unstable ("soft") packages include configuration and build scripts, customization modules, and business rules which reflect changing external conditions (such as the tax rules for the current year). (Martin mentions the example of a procedure which reports the version number [2].) By contrast, packages which are intended to change rarely, and allowed to be difficult to change, are called *stable* (and we naturally would prefer *hard*.) Examples of stable ("hard") units are 'core' data structures and methods, and interface modules. Once a package has been designated as hard (or "set in stone"), then its design decisions can be safely allowed to lead the rest of the design, without further indirection, abstraction, or information hiding: the rest of the system can depend on them to any desired degree.

But our best-laid plans often go off track, so we must observe change as it actually happens, as well as how it was planned. We accept Martin's use of the term *volatile* to describe those packages which actually change often during evolution and maintenance. In these terms, we can control maintenance costs by ensuring that volatile packages are soft, so that the change we expect is easy to carry out: equivalently, we may ensure that hard packages are nonvolatile, so that the difficulty of changing core structures and interfaces rarely arises. (Conversely, should soft packages be volatile? This seems less important. However, checking that soft packages are volatile reassures us that the investment in softness (by means of wrapping, interface design, modularity enforcement, *etc*.) hasn't been wasted.

Now *change propagation* is this: a unit changes, forcing its dependents to change also. When changing a unit we have to consider its dependents; we have to assess the impact of the planned change on them, and possibly expand the scope of the change to include them. Thus *dependent units are made more volatile* by change propagation, and in the same way, *units which are depended on are made harder* by change propagation. In order to achieve the cost control described in the previous paragraph, we should therefore try to arrange for dependent units to be soft (because they tend to be volatile) and for depended-on units to be nonvolatile (because they tend to be hard).

Martin distills these observations into another packaging principle, which he calls *Stable Dependency Principle*, to the effect that "a package should only depend upon packages that are more stable than it is." [2] Martin's assumption is that *the harder (more stable) a package is, the less volatile it tends to be*, and therefore dependence only on harder packages would tend to result in dependence on nonvolatile packages, as desired. Though plausible, this is an assumption which needs validation. Perhaps there are other causes of volatility which are more influential than instability is. Similarly Martin assumes that *the more dependent a package is, the softer it tends to be*. Since, as we noted, dependency tends to cause volatility, Martin's two assumptions are equivalent.

The main reason for formulating the Stable Dependency Principle is this: we cannot directly know the future volatility of a package, but we can estimate hardness from an examination of the package dependency. The second form of Martin's assumption above says that softness increases with dependency. Thus, Martin formulates his "stability metric" directly in terms of the dependency relation between packages, as follows. Consider a node (that is, a package) p in the dependency graph. It has incoming edges (Martin calls them "afferent dependencies") and outgoing edges ("efferent dependencies"). Write A_p for the number of incoming and E_p for the number of outgoing. Then the contextual stability S_p of p is the proportion of incoming edges to the total: $S_p = \frac{A_p}{A_p + E_p}$. A package on which no other package depends has a contextual stability of 0, and by assumption maximally tends to be volatile, due to its dependence on other packages. A package which depends on no other packages has a contextual stability of 1, and by assumption maximally tends to be nonvolatile due to its dependents. Now Martin recommends the practice of conforming to the Stable Dependency Principle with respect to *contextual* stability: that is, a package should only depend upon packages that are more *contextually* stable than it is.

For a metric to be considered valid, it must provide reasonable measurements of a desired variable in known situations. The desired measured variable is 'hardness to change'. Thus to validate this metric a system is required that has a range of packages with varying hardnesses. The empirical hardness of each package must be estimated, then compared to the contextual stability. If there is a high correlation between the two, we can conclude that the metric is reasonable. The more situations studied, the greater the confidence in the metric.

This suggests two research directions. The first is to explore the relationship between hardness and volatility for a large code base under current maintenance, looking for evidence for Martin's assumption that hard packages are nonvolatile and volatile packages are soft. The second is to explore the metric itself looking for correlation between hardness and contextual stability: does the metric actually measure what is intended? (For more on the relationship between metric and measure, see Zuse [20].) In the present paper we obtain some preliminary results for a large, high-quality, evolving, open-source system (aha! Linux!) by

Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)
0-7695-1903-2/02 $17.00 © 2002 IEEE

comparing the volatility, measured in terms of actual change between releases, to the contextual stability as defined by Martin. If both Martin's assumptions are correct, we should find at least some directional correlation: volatile packages should tend to have low contextual stability, and nonvolatile packages should tend to have high contextual stability. But this is not what we find…

1.1 Related Work

Packaging is treated under the theory of configuration management in software engineerings texts [6,7,8,9,10]. Packaging in the setting of software evolution was studied by Oreizy [20] and by van Deursen and de Jonge [13].

Our work was originally inspired by Martin's account of packaging in object-oriented software development [2]. Although we have concentrated on the Linux kernel, study of other architectural settings would provide much-needed perspective, as pointed out by van Deursen and de Jonge.

Previous work by the Software architecture Group at the University of Waterloo used a similar approach to ours for the generation and analysis of source-based data. Base facts are extracted using compiler techniques [16,20]. We considered Bowman and Holt's work [11] in the decisions of how to analyze the Linux kernel. Godfrey and Tu's work [4,5] helped during considerations of how to measure change.

2 Choosing a Guinea Pig

In software reverse-engineering and evolution studies the term *guinea pig*[2] is used to refer to subject software systems accessible as source code. Inevitably open-source systems are more frequently guinea pigs than closed-source systems are. Our criteria for a system to empirically evaluate change were these:

1. source code available in many releases – in order to estimate volatility and compute contextual stability –
2. reasonably mature – to avoid special cases of software systems in early-release frenzy –
3. packaged according to Martin's principles – in order to give his assumptions the best chance – and
4. healthy, with a solid design and development history, rather than "aged" as defined by Parnas [3].

Our estimate of volatility would be most accurate with a long but cautious release history, and a minimum of special events (large migrations or architectural restructurings). Our estimates of hardness would be more accurate for a large code base having many dependencies.

While it doesn't satisfy all these criteria we feel the Linux kernel was a good choice for a guinea pig. It is open-source, of course. It has been in active development for 12 years (since 1991), and all its releases are available for download. It is publicly acknowledged as a high quality software system[3]. It has been released 499 times and consists of about 850 packages with currently (v2.4.18) about 3.7 MLOC of C source code. It is mature, stable, and widely used, and certainly shows no signs of Parnas aging (often manifested by issues such performance degradation, fail-stop behaviour, abnormal termination, increased maintenance difficulty, and an increase in fault introduction [3]). It has been studied by software evolutionists already [11, 5, 21].

It's not easy to assess the degree to which Linux is packaged according to Martin's principles. Linux is not formally object-oriented, and so there is an immediately visible architectural mismatch with Martin's development methods. Linux is not organized for reuse.[4] There is no notion of "package" conceived separately from the architectural and modular structure. Therefore, for our purposes we adopted the directories of the source tree as "packages", since they are the units tracked by change management in the open-source project. What is more, directories represent the division, by the developers, of the large software system into more manageable subunits that can be assigned to specific developers or groups of developers. This is analogous to packaging in the sense we are using it. On this view Linux has historically had about 1500 packages, not all of which are present in any given release.

3 Measuring the Volatility of Linux Packages

To assess a packaging strategy we need to quantify the volatility of packages. Having accepted the term 'volatile' for packages which 'change often', we naturally wish to define the *volatility* of a package as 'the probability that it will change on the next release of the source'. In a release-management régime where only changed material is released (the 'patch' style) it is the probability that the

[2] From their use in experimental biology. Given the size of some software systems, perhaps 'capybara' would be more appropriate than 'guniea pig'.

[3] See for example `msnbc-cnet.com.com/2100-1001-985221.html` or `www.catb.org/~esr/writings/`

[4] E. S. Raymond discusses in "Homesteading the Noosphere" the strong aversion to forking (reusing) that exists in the open source community.

Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)
0-7695-1903-2/02 $17.00 © 2002 IEEE

package is included in the next release. For a CVS-managed[5] régime, which includes a release attribute, it is the probability that the content of a package is different between a pair of successive randomly chosen releases.

Anyone familiar with software evolution and maintenance in practice knows that the volatility of a given region of the design depends on the long-term software life cycle (see Bennet and Rajlich [19] for discussion). Equally familiar to programmers, however, is the presence of 'never touched' and 'always changing' source files within the local culture of a software project. These experiences are our reason for preferring a guinea pig system which is mature and not subject to catastophic restructurings. In such a case, we assume that the change rate is fairly steady, and that we can therefore estimate the volatility by counting releases.

An important phenomenon in software evolution is the movement and copying of source material (not only source files but portions of source files) between packages. This occurs, for example, when a module overburdened with responsibilities is refactored by splitting into two coupled modules. An accurate analysis of origins [5,18] for the release history of a project is difficult to obtain. Nevertheless, it might be objected that code from the original module had migrated unchanged to the new modules, and therfore ought not to be counted to the rate of change. For the measurements reported here, we deliberately chose to ignore this issue and treat the source in each package as self-originating. From the point of view of change management the origin of source material is irrelevant. If a package changes (even by receiving unmodified code from some other package, or by losing code unmodified to some other package) then impact assessment, change propagation, or at least recompilation, is needed. These are the very phenomena we are trying to measure.

Therefore we model the release history as follows. Let there be given a fixed set P of packages – or more precisely of package 'identifiers' since their contents vary between releases. A release r is a partial function from P to possible package contents – this may be thought of as source file names and source text, or just as a package version number: anything that can be compared between releases. Given two releases r_1 and r_2, and a package p released in either of them, then we say that p is *unchanged* between the releases when $r_1(p) = r_2(p)$ (both defined and equal) – otherwise p is *changed*. A release history $(R,\prec)$ is a set R of releases (of P) together with a partial order : $r_1 \prec r_2$ when release r_1 historically precedes release r_2. When $r_1 \prec r_2$ and there is no r_3 such that $r_1 \prec r_3 \prec r_2$ then we say that r_1 is a *predecessor* of r_2 and r_2 is a *successor* of r_1. A package $p \in P$ is *changed for a release* r if it is changed between r and any of r's predecessors. The set of change releases C_p of a package p is $C_p = \{r \in R \mid p \text{ is changed for } r\}$. This is a subset of the set R_p of releases of p, that is $R_p = \{r \in R \mid r(p) \text{ is defined}\}$. The *volatility* of $p \in P$ is just the ratio $V_p = \frac{|C_p|}{|R_p|}$.

With this definition of volatility, a package appearing in an initial release and unchanged in any subsequent release has a volatility of 0. A package introduced after an initial release and changed in every subsequent release has a volatility of 1 (and so does the special case of a package just introduced in a most recent release). These extrema satisfy the intuition, but some consequences may be less attractive. A package appearing in a merge release (several branches closed off at once) is reckoned to have changed for that release unless its contents at the end of all those branches are the same as its contents after merging (this situation never appeared in the Linux kernel). A package which is much changed in some experimental branch but never changed for production releases will nevertheless appear volatile. A package which is renamed will seem to have disappeared (its history ended) and a new one appeared (with no history of volatility). Lastly, this definition is sensitive to the notion of "contents" of a package. If the contents of a package is the names of its source files, then organizational volatility is measured; if the contents of a package is the raw text in all its source files, then the slightest change even to comments or spacing will appear to be a change. However, this measure of volatility is easy to compute, and concurs with the manager's need to know if a package "has changed at all since last time". One prefers to err on the side of caution when assessing impact of change. Our method used file size when considering change, and therefore there is a slim chance this method would miss a change if the change preserved the exact size of each source file changed. This type of change seems rare.

For the Linux code base, we obtained 499 releases containing a total of some 1500 "packages" (really source directories). We represented the contents of each package as a "fingerprint" consisting of the names of any subdirectories and the sizes in bytes of any text files. This approach recognizes (almost!) all textual changes to source files, and any movement of files or subdirectories, but without counting changes to subdirectories (which are packages in their own right). When a subdirectory is moved to a different package, a change will be registered in four packages (the original parent, the new parent, the original subdirectory, and the new subdirectory). As noted

[5] For information about CVS consult `www.cvshome.org`.

previously, there is no origin analysis.

Some remarks regarding text files is in order, as the releases of Linux include text files which are not programming language source files. Documentation files (having extension `txt`) were considered part of packages in which they appear, as they record group knowledge about the software architecture. In the Linux release management structure they are for the most part isolated in documentation directories, and therefore changes to documentation alone have little effect on the volatility of

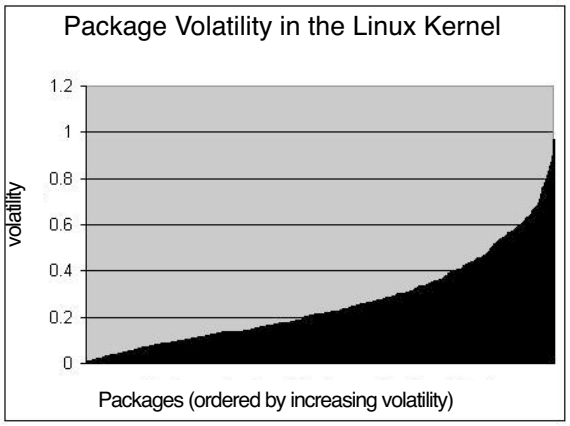

Figure 1 - Volatility Profile. Choose a volatility level on the vertical axis; the horizontal distance to the intersection with the curve at that level shows the proportion of packages which are at most that volatile. The 50% level is at volatility 0.23, so that half of the packages change less than a quarter of the time.

the system as a whole. Build scripts (having extension `sh`) and Makefile files[6] were also considered part of packages as they contain essential (albeit operational) data about the system's architecture. Changes to any of these classes of text file we consider to be changes to their enclosing packages.

Having assigned a volatility in $[0,1]$ to each package, we sorted the packages by volatility, displaying the resulting positive curve as Figure 1. This may be called the 'volatility profile' of the Linux kernel, as it shows for each level of volatility (height on the vertical axis) how many packages (width on the horizontal axis) are at least that volatile.

[6] For information about Make see `www.gnu.org/software/make`.

4 Measuring the Contextual Stability of Linux Packages

For reasons stated above we would prefer to measure the hardness of Linux packages, but the actual difficulty of change cannot be assessed without detailed knowledge of the change process and history. Instead, we compute the value of Martin's stability metric and use this as a substitute estimate of relative hardness.

The contextual stability of a package is defined in terms of its dependence *in a single release of the system*. This is in contrast to volatility which is defined over the release history. It is no surprise that contextual stability should be localized in time, because its purpose is to give an estimate of softness (or nonvolatility) which can be made on the basis of the system's packaging *as it is now*. However, for our purposes it means we must select a typical release of Linux and assess the contextual stability for that particular release. We chose release 2.4.18 as being a typical recent release.

In order to compute the contextual stability we need to know all the packages (directories) and all the ways in which they depend on each other. Although this might be roughly assessed by creative use of Unix tools (`grep`, `sed`, `awk`, `sort`, *etc*.) a much more accurate picture is obtained by 'build time view' data collection [16,17].

The build-time view ("BTV") technique collects facts detected during an actual build of the system. It is based on an instrumented version of GNU Make. During normal processing make notices dependencies between 'targets', many of which are actually source files. Some dependencies are explicitly stated in make files; others are implicit in Make's built-in rules. This is one of the many reasons why dependencies are hard to extract from static source files. During the processing of Linux's build scripts, additionally, some scripts are dynamically generated and those and others are dynamically executed – further complicating the task of discovering dependencies from the build scripts and source code. Lastly, the Linux build process makes use of dynamically created symbolic links to represent aspects of the target architecture. For all these reasons, tracking the actual build process was essential to getting an accurate view of dependencies between "packages".

By extracting dependencies between packages based on the execution of the build process, we risk failing to notice two kinds of dependency which may exist in practice. Firstly, it may happen that some source file say `F.c` includes some header file say `G.h`, but the dependency is not noted explicitly or implicitly in any make script. Secondly, we will miss any dependency which is not triggered by building the top-level target.

Fortunately, these limitations don't prevent us from getting accurate file dependencies for the Linux kernel. It happens that file inclusion dependencies are dynamically computed by Linux's build process (using the tool `mkdep`). Hence the first risk is no threat in our case. (The first risk is technically an error in the build process, but can be benign, especially when the files are all in the same package or change in concert with some third file whose dependency *is* recorded. Hence users of our technique must take care of this point when working on other systems.) We addressed the second risk by covering as many key parts of the build process as possible. We built the target `modules` which generates modules, such as AFS and the Ethernet drivers, that are not included with the kernel source package, and we built the main target `bzImage`. The Linux kernel approach to handling multiple architectures is to create a symbolic link to an architecture specific directory. We considered all architectural configuration that could be linked (so that something depending on the symbolic link would depend on all architectures that contain the file it is looking for). Device drivers are handled in the Linux kernel as seperate software systems to be dynamically loaded by the kernel. Hence there appear to be multiple, semi-autonomous systems bundled in each kernel source distribution. Using the BTV package we obtained dependencies between these packages also.

Having computed a contextual stability in $[0,1]$ for each package of release 2.4.18, we sorted the packages by stability, and displaying the resulting positive curve as Figure 2. By the very shape of this it is immediately apparent that contextual stability measures something other than volatility for the Linux system.

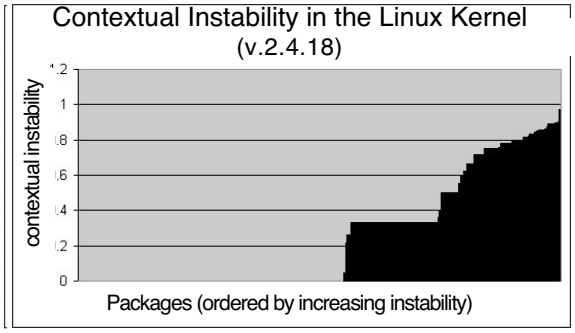

Figure 2 – Stability Profile. More than half the packages have contextual instability 0, which means that we found in them no static dependency on other packages. Much of this is device driver code (see [5] for discussion) A large number of packages have contextual instability 0.33, typically because of one incoming and two outgoing dependencies.

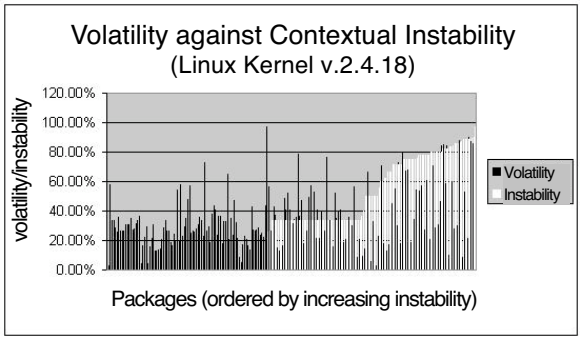

Figure 3 - This is Figure 1 augmented with the computed contextual instability displayed (in white) for each package in v.2.4.18. If there had been the expected correlation, the white bars would tend to be longer on the right and shorter on the left.

5 Findings

In this section we compare the two measures of change computed in the previous sections. We also relate the change measures to a high-level view of the software architecture.

We had begun with the assumption that Linux's directory structure might be taken as an example of Martin-principled packaging. We found that although the directory structure was similar to Martin's packages, it did violate the principles in some instances. For example, in the 2.4.18 kernel there are three pairs of circular dependency (between `linux` and `linux/sunrpc`, between `linux` and `net`, and between `net` and `net/irda`.) This seemed a small exception given the number of packages in a typical release.

5.1 No Correlation

We had sought positive correlation between volatility and contextual instability, since if Martin's assumptions are correct for Linux we would have seen that packages which actually change slowly tend to have more packages dependent on them than they have dependencies on others; and packages which actually change quickly tend to the reverse situation. To visualize this we displayed the contextual stabilities of packages against the volatility profile (see Figure 3) and conversely the volatility of packages against the contextual stability profile (see Figure 4). It is immediately clear that these functions are quite unrelated. These results cast doubt on the validity of Martin's assumptions, but provide a valuable approach for

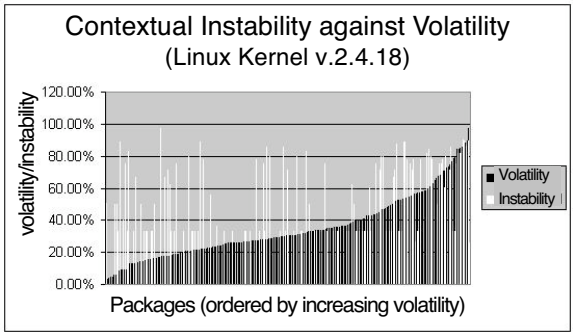

Figure 4 - This is Figure 2 augmented with the observed volatility of each package in v.2.4.18. The data are similarly uncorrelated.

verifying such metrics.

From the failure to correlate these two measure, and the discussion above in **2**, we can conclude that for the evolving Linux system, one of the following is true: either

1. a certain number of packages are easy to change despite many others depending on them, or
2. a certain number of packages are changed frequently despite being hard to change

Future work must find an independent way of measuring hardness in order to discover which.

5.2 Architectural Volatility

Godfrey's previous work on the growth of Linux [4,5] has shown areas of high volatility in the architecture. We having obtained the foregoing negative result then sought some correlation between the architectural structure of the system and its change rate. This is displayed in Figure 5, in which the packages are again displayed in increasing order of volatility, but now their architectural classification (subsystem) are broken into 5 parts.

It may be seen that the four different subsystems differ in volatility. Given that these results were determined using a long run of versions, spanning many years, these results are not due to simply to chance or (temporary) local architectural restructuring. Instead they give insight into how likely change is in these various parts of the system. We had originally hoped that packages might tend to fall into "hardness categories". This may be viewed as a partial fulfillment of that hope, in so far as we find the subsystems of Linux tending to fall into "volatility categories". Future work must explore this connection more closely and discover how architectural roles can be predicted by volatility, and *vice versa*.

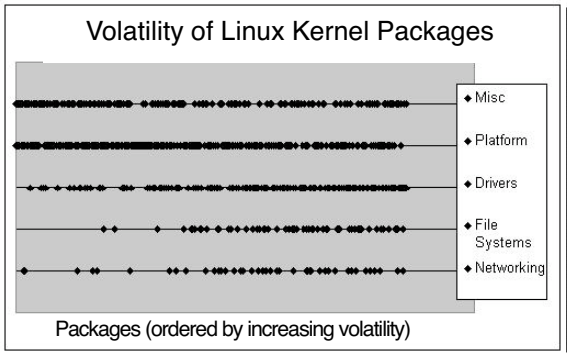

Figure 5 - Subsystem Volatility Profile. Here is shown the subsystem each package (on a separate horizontal line) belongs to; its relative volatility is indicated by its relative position on the x axis, as in Figure 1.

Acknowledgements

The authors thank Michael Godfrey and the Software Architecture Group at the University of Waterloo for feedback and thoughts provided on early versions of this work. We are additionally indebted to the open-source developers of the Linux kernel for making their work available to all.

References

[1] D. F. D'Souza, A. C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*. Object Technology Series. Addison-Wesley 1998.

[2] R.C Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall 2002.

[3] D. L. Parnas. Software aging. In *Proc. ICSE '94*, Sorrento, 1994

[4] M. Godfrey, Q. Tu. Growth, Evolution and structural change in open source software. In *Proc. IWPSE-01*, Vienna, 2001.

[5] M. Godfrey, Q. Tu. Evolution in open source software: a case study. In *Proc. ICSM-00*, San José, 2000.

[6] B. Bruegge, A. H. Dutoit. *Object-Oriented Software Engineering*. Prentice Hall, 2000.

[7] G. M. Clemm. Replacing version control with job control. In *Proc. 2nd IWSCM*, ACM, Princeton, 1989. pp. 162-169.

[8] R. J. Leach. *Introduction to Software Engineering*. CRC Press, 2000.

[9] R. S. Pressman. *Software Engineering* 3rd Edition. McGraw-Hill, 1997.

[10] I. Sommerville. *Software Engineering*. Addison-Wesley, 1989.

[11] I. T. Bowman, R. C. Holt. Linux as a case study: its extracted software architecture. In *Proc. ICSE '99*, Los Angeles, 1999

[12] J. K. Hollingsworth, E. L. Miller. Using content-derived

names for configuration management. In *Proc. 5th SIGSOFT*, ACM, 1997.
[13] A. van Deursen *et al*. Feature-based product line instantiation using source-level packages. In *Proc. SPLC2*, Springer-Verlag, 2002
[14] T. Dean *et al*. Union schemas as the basis for a C++ fact extractor. In *Proc. WCRE 2001*, Stuttgart, 2001.
[15] P. Oreizy. Decentralized software evolution. In *Proc. IWPSE 1*, Kyoto, 1998.
[16] R. C. Holt *et al*. The build/comprehend pipelines (position paper). In *2nd ASERC Workshop on Soft. Arch*. Banff, 2003
[17] Q. Tu, M. W. Godfrey. The build-time software architecture view. In *Proc. ICSM-01*, Florence, 2001
[18] M. W. Godfrey, Q. Tu. Tracking structural evolution using origin analysis (position paper). In *Proc. IWPSE-02*, Orlando, 2002.
[19] G. Bennet, V. Rajlich. Software evolution: a road map. In *Proc. ICSE 2000*. Limerick, 2000.
[20] H. Zuse. *A Framework of Software Measurement*. Walter de Gruyter, Berlin, 1998.
[21] Y. Xie, D. Engler. Using redundancies to find errors. In *Proc. 10th SIGSOFT*, ACM, 2002.

Proceedings of the Sixth International Workshop on Principles of Software Evolution (IWPSE'03)
0-7695-1903-2/02 $17.00 © 2002 IEEE