An Empirical Study of Software Packaging Stability

by

John C. Champaign

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, December 2003

©John C. Champaign, 2003

I hereby declare that I am the sole author of this thesis.
I authorize the University of Waterloo to lend this thesis to other institutions or individuals for the purpose of scholarly research.

John C. Champaign

I authorize the University of Waterloo to reproduce this thesis by photocopying or other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

John C. Champaign

The University of Waterloo requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

Acknowledgements

Thanks to all members of the SWAG group at the University of Waterloo for their thoughts, encouragement and good times while this thesis was in preparation.

Special thanks go to my supervisor, Andrew Malton, for guiding me through the very educational process of producing this document.

Abstract

Packaging, a subset of "configuration release management (CRM)", deals with the best techniques for dividing a source base into parts that can be maintained and released independently.

Robert Martin proposes six principles in his book *Agile Software Development* that promise superior results for packaging. In this work we seek empirical validation of his "Stable Dependency Principle" and its benefits.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this work we examine a method of packaging software systems proposed by Robert Martin. Part of his approach involves packaging such that it will be clear how difficult it is to change a package, and consequently how often these packages are expected to change.

He offers a software measurement promising this predictive ability. His *Stable Abstraction Principle* is a measurement of the *Instability* of a package, namely how difficult it is to change it. It is a reasonable leap to suppose that given a lengthy development history, areas of a code base that are easier to change will change more often, and areas of a code base that are harder to change will change less often. This gives us a measurement that purports to predict likelihood of change.

We try to empirically validate this principle, by considering what it is measuring and finding other methods of determining the same information. In this case since he provides a measure for likelihood of change, we looked at historical change rates and compared them to what he predicted they would be.

1.1 Packaging

Packaging is a method of partitioning a software system such that it is easier for developers to think and talk about it. This has obvious similarities to modules, subsystems, and components. The main difference between packaging and modules or subsystems is that a package is something that should be capable of being taken from one system and included in another. Module and subsystems have the implicit assumption that they are tightly coupled with the system they reside in. The primary difference between components and packages, is that components strive to be far more generic than packages. It is perfectly acceptable to expect a software system to make fairly large changes in order to include a package. Components on the other hand are tailored to be easily dropped into many diverse environments.

In addition to partitioning a system for developers, packages also effect maintenance and release. Being able to move a code base through quality assurance in smaller units has similar advantages to being able to develop it in smaller units. Releasing systems in packages rather than as monolithic wholes allow for smaller customer releases, since only packages that have been modified need to be released.

[1] A more rigorous definition of packaging from software development terminology is: a basic development unit that can be separately created, maintained, released, tested, and assigned to a team [37].

In practice a package is a subset of the source files of the system, and 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[2]. An appropriate choice for separating the software system is based on

[1] The following portion of this section was adapted from [35]

[2] 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.

how packages depend on each other and how they change from release to release. Packaging determines the dependency relation between packages, induced by the dependency relation between source files. Any change to a source file requires a new release of the package it belongs to.

In his book *Agile Software Developement*[52], Robert Martin of Object Mentor suggests that a package is chosen such that:

1. each package is reusable
2. no package contains two or more reusable subsets
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 maximize flexibility, and the third enables packages to be released independently.

1.1.1 Hardness and Softness

It is 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, 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 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[52]. FIXME - add page number By contrast, packages which are intended to

change rarely, and are 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.

It should be mentioned that although hard and soft would intuitively be mutually exclusive, our nomenclature does allow a package that is both hard and soft (or which has low hardness and softness). A package that was both hard and soft would be one that is expected to stay the same, but if for some reason it needed to be changed, would be easily changed. Such a package would be considered over-engineered as the development team has wasted effort making something easy which will rarely occur. The reverse situation would be no better, as a package with both low hardness and softness would be one that is expected to change often but is difficult to change.

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; the changes we expect are easy to carry out. 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. Equivalently, we may ensure that hard packages are non-volatile, so that the difficulty of changing core structures and interfaces rarely arises.

1.1.2 Change Propagation

General change in software systems is discussed in section 2.4. Change propagation is a change in a software unit that forces its dependents to change also. When changing a

unit it is important to consider its dependents, 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 non-volatile (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 [52]. 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 non-volatile 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 that we cannot directly know the future volatility of a package, but Martin says 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 software measurement directly in terms of the dependency relation between contained 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 of p, S_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 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 non-volatile due to its dependents. 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.

It has been shown FIXME - add citation that historical changes in software systems are good predictors of future changes. In this work we are more interested in a predictive tool that can be used without analysing the development history. That is, predict future change using only one release of the software system.

1.2 Motivating Example

MacroHard software had been having success with its word processing software "Letter". Many of its customers had expressed interested in buying other office applications that functioned in the same way as Letter, such as spreadsheet, presentation and web page design software.

MacroHard realized that there was duplicate functionality between the applications, such as text editing, spell checking and a help system. They considered different alternatives for taking advantage of their existing code base, and using it to build these other applications.

The first idea they considered was to make it all part of one very large application. Marketing vetoed this idea because they believed there were markets to sell parts of the suite in, even though they might offer it as a bundle. The lead architect was opposed to this as well as he believed that the project would be more manageable if these parts were kept separate (although still inter-operable).

The second idea they had was to copy the relevant code to use as a basis of the other suites and then build on this. The head of quality assurance objected to this on the

grounds that it would means similar code would exists in multiple places, each having to be maintained independently. She gave the example of the wasted man power spent in tracking down a bug that appeared in this fundamental code in 5 different projects, and the savings provided by keeping this code in one place.

The third idea was to abstract all the common functionality that was needed for all or most of the application, then using relevant code from Letter build generic components for each of these which developers could then easily plug into their systems. These components could then be used by third party developers, or sold to other companies developing office applications. The head of marketing was behind this idea technically, but told everyone that they didn't want to enable third party developers to get that close to the code (his belief was that maintaining a competitive advantage required a small API to be exposed). In addition, he definitely didn't want to help competitors get their projects to market faster using these components, so he said that the company wouldn't be selling these externally either.

At this point the lead developer joined the conversation and said that he could create these components, but there would be a lot of wasted development effort making them so generic if they were going to be used in house. His claim was that it was ok if effort was required to integrate this fundamental code into the different projects as they would have that expertise in house.

After listening to all the concerns, the wise and benevolent Chief Technical Officer finally spoke. Packaging is the answer to this issue he proclaimed. Like with component, we will abstract out the functionality we need for these fundamental concerns. A new team will be formed from developers who have been working on Letter and are familiar with the relevant code. This new team will then maintain this fundamental base, and treat each of the office application development teams as their customers. They will put the code into understandable parcels, which we will call packages, which provide a set functionality needed by the different applications. If new functionality is required by a subset of two or

more of the applications, this will also be developer by the core team by building on what they already have, and packaging this for use by the relevant teams.

The most fundamental packages in the fundamental codebase, such as the spell checker, will provide a interface for the teams to use that will stay the same as much as possible. Although it will be stable, if it is absolutely necessary to change it we can. This will cause a lot of work for the various teams however, as they will all have to modify how they interact with the spell check package. He called this change propagation, where changing one package requires changes to another and said it needed to be minimized.

The more specialized packages should depend on these stable packages, and in turn can change more often as they will impact fewer applications that depend on them. The applications themselves will also be created as packages, as marketing has ideas for specialized version of each, such as our upcoming "Legal Letter", which is a word processor tailored for lawyers. These application packages will be able to change much more freely however, and this is were the multitude of differentiating features we need to be competitive in the marketplace should be added.

The added advantage to all this is that we can lower our overhead when providing updates to customers. Only the relevant packages that have changed for the applications they are using need to be shipped to them. As changes are made, if the majority of these are restricted to the application packages, there is less need for validation, as the core packages should remain stable (that is unchanged) for longer periods of time. The smaller amount of data to push out to customer will translate into lower bandwidth costs for us.

And all were impressed with the wisdom of the CTO, and it was made so.

1.3 Thesis Organization

The rest of this thesis is organized as follows: Chapter 2 provides a background on agile software development and Martin's packaging principles. In addition it gives some back-

ground on measurement in general and software measurement specifically. The background also includes an overview of configuration management which is the larger context that packaging and this work is set in. Chapter 3 presents an overview of our experiment, how we approached evaluating the software measurement and the systems we choose to study. Chapter 4 presents the results, our interpretation of them, and related work. Chapter 5 summarizes these findings, discuss the limitations and future directions of the work.

1.4 Major Thesis Contributions

In this thesis we provide a technique for empirically validating evolutionary software measurements, specifically likelihood of change. A straightforward technique is provide for evaluating change at the package level, using historical development information (nothing more then each release of the system), which could easily be modified for modules, subsystems or classes.

Our results are useful in deliberations about where to apply Martin's software measurements.

Chapter 2

Background

2.1 Agile Software Development

Agile software development initially arouse as an attempt to unify many of the emergent development methodologies that embrace change. These include: Extreme Programming (XP), Agile Modeling (AM), Scrum, The Crystal Methodologies, Feature Driven Development (FDD), Dynamic Systems Development Method (DSDM), and Adaptive Software Development (ASD) [39, 48].

Agile software development values: [4]

1. Individuals and interactions over processes and tools
2. Working software over comprehensive documentation
3. Customer collaboration over contract negotiation
4. Responding to change over following a plan

The prime focus of Agile software development is to quickly deliver software that meets customers' needs. Improved satisfaction is achieved using techniques such as adapting

to changing requirements, frequent (working) software release, and regular interactions between developers and customer representatives [39]

Agile software development attempts to achieve these goals by placing a higher importance on code and a lesser importance on documentation. They can be said to be adaptive rather than prescriptive [39]. That is, rather than trying to plan the entire development process at the initial stages, agile development would encourage the team to start immediately, and make decisions when they are forced to. The rationale for this is that at this point they will have more domain knowledge, and therefore be in a better position to make the decision. Indeed, often the project requirements are defined only for the next 2-12 weeks [48].

By building the process on the strengths of the people involved (their expertise, competency and teamwork) in lieu of a formalized, rigid process [39] agile software development promises significant gains in terms of development speed and quality.

This is clearly a departure from traditional development. The configuration management information presented later in this chapter highlights some of the effects of packaging. This new approach is not without critics [14, 53].

2.2 Martin's Agile Packaging Principles

Over the years researchers and practitioners have proposed new approaches for creating software. Successful proposals, such as object-oriented development, dramatically affect how we execute this process and receive acceptance among practitioners. While Brooks cautions us that there is no silver bullet (an innovation which will make something dramatically easier) in software development [33], improvement is still possible.

Packaging is one area where improvement is being sought. As mentioned in the introduction, the goal of packaging is to partition a code base in a way that makes it "easy to understand, test, maintain and extend". While there is generally agreement that dividing

a software system has value [42, 58, 64], there is no consensus on the best way.

In his book[52], Robert C. Martin applied years of development experience and proposes six principles for packaging source code [52]. There has been widespread excitement about Martin's work [8, 9, 10, 17, 20, 24], to the degree that it has become the text for undergraduate computer science courses [8, 9, 10], taught in industrial seminars [20], and used as the basis for source analysis software [17, 24].

In the following sections we explore each of Martin's principles, focusing on how to empirically validate or invalidate each. The remainder of the work is devoted to consideration of the *Stable Dependencies Principle* and carrying out its validation. Validation of the remaining principles is left as future work.

2.2.1 Acyclic Dependencies Principle

Allow no cycles in the package dependency graph. [52, pg 256]

If there are cyclical dependencies in a group of packages, then any change to any of these packages effects all of them. As well, it will always be a challenge to integrate them with the rest of the system, since any dependence relationship with one of the packages in the cyclical dependency will have an implicit relationship with the entire cycle. If there is a graph without cycles, then integration becomes much more simple (treat packages that you depend on as suppliers, and packages that depend on you as customers). [52]

Identifying where this principle has been followed is simple. If there are cycles in a software system's dependency graph than it has not been followed, if there are no cycles than it has.

The promised payoff for following this principle is that you will avoid the "weekly build" FIXME - add citation of weekly build problem problem where every change made to code involved in a circular dependency breaks another part, triggering a series of corrections.

To validate the gains provided by this principle would require examining a system that

has a cycle in its dependency graph. A comparison would then be made between the number of bugs or maintenance time devoted to the packages in the cyclical dependency and compare them to the packages not involved.

2.2.2 The Stable Dependencies Principle

Depend in the direction of stability. [52, pg 261]

Some packages should be easy to change, some should be hard. Packages that are easy to change should depend on packages that are less easy to change. Validation and release become easier the "higher" a package is in the dependency graph (the less packages that directly or indirectly depend on it).

Having other packages depend on a package may tend to make it more stable, as more effort is required to reconcile changes with all the dependent packages. Depending on other packages may tend to make a package less stable, as changes made in the depended-upon package may propagate to the depending package.

How easy a package is to change is presented by Martin as a characteristic called "instability" which he calculated as:

$$I = \frac{E_p}{E_p + A_p}$$

> where I stands for Instability of package p, E_p represents the number packages that depend on p and A_p represents the number of packages p depends on. [52]

In a system following the *Stable Dependencies Principle* there should be a correlation between depth in the dependency tree and the level of stability of the packages.

The gain promised by following this principle is that change propagation will be minimized, as the packages with a higher likelihood of change will have less dependents. It

makes it easier for new developers to decide where to make changes in the system (preferably to packages with the smallest number of dependents) as they gain familiarity and understanding of the system.

Validation of this principle involves comparing the instability, or predicted rate of change, with the volatility, or observed rate of change, to see if they correspond. The rest of the present work details this process.

2.2.3 The Stable Abstraction Principle

A package should be as abstract as it is stable. [52, pg 264]

Tightly coupled with *Stable Dependencies Principle*, the *Stable Abstraction Principle* claims that instable packages should be highly abstract. This is based on the idea that abstract classes should formalize the design at the code level, and therefore should not be easily changed. [52]

Determining whether a package follows the *Stable Abstraction Principle* involves calculating the packages' volatility (calculated Instability, which is detailed in section 2.2.2, is one technique for doing this) and comparing this to the calculated abstractness of the package. Abstractness is calculated using the following formula:

$$A = \frac{P_a}{P_c}$$

> where A stands for abstractness of a package, P_a represents the number of abstract classes in that package and P_c represents the total number of classes, both abstract and concrete, in that package. [52]

The package can be said to follow the SAP if there is a correlation between the abstractness and the stability of a package (high abstractness implies high stability and vice versa).

The gain promised by following this principle is to avoid two problematic development situations. The first is abstract packages that are instable (nothing depends on them). The problem with this situation is that abstract packages typically specify an interface or protocol rather than implementing functionality. If few packages depend on it, then there is little use in specifying it. The second situation occurs when a concrete package is highly stable (many things depend on it). The problem here is that since the implementation details are in the package, we are going to have to modify it to do maintenance. However, every time we modify it we run the risk of breaking one of the many packages that depend on it directly or indirectly.

Verifying these gains requires a demonstration that concrete, stable packages are difficult to change. A suitable technique doing so would be to measure change propagation, that is the number of change request required to correct a problem introduced by another change request, and length of time required to implement changes. These measurements would then be compared to packages that were judged not to be concrete nor stable.

2.2.4 The Reuse/Release Equivalence Principle

The granule of use is the granule of release. [52, pg 254]

All of a Package should be usable by the same audience and each release should be treated as a separate product. This means that it is guaranteed to be supported for a certain length of time, and that the user has the option of staying with an old version rather then updating.

This is a part of *System Building* as detailed later in the chapter.

If a package is being used by multiple audiences, there will be pressure on it to change in different, perhaps mutually exclusive, ways for each group of customers. A preferable approach is to have the commonalities moved into a new package, and have specialized packages fulfilling the needs of each of these customers groups. These specialized packages

may in turn depend on the package containing the commonalities. The customer of the common package can be considered package developers. This is an example of version control as detailed in the configuration management section (2.4).

The rational for treating each release separately is simply that customers may not want to upgrade for functionality enhancements. Should they make this reasonable decision to avoid the upheaval of upgrading they should still be provided with bug fixes. All current releases need to be considered each time a bug fix is produced. After a negotiated length of time it is reasonable to retire a release and no longer target it for any maintenance.

This can be viewed as a management issue related to the released customer base and advertised service rather than a technical consideration. [52]

To determine if a project was following this principle would simply require asking one of the developers their policy on maintenance for obsolete packages. The gain made from following this principle is increased customer satisfaction as they do not feel they are relying on a mercurial release schedule.

Validating this principle would be challenging as it is primarily a political issue. A convincing approach would be to conduct user satisfaction surveys for a variety of projects and see if there is a correlation of increased customer satisfaction in project that adopted this principle.

2.2.5 The Common Closure Principle

A change that affects a package affects all classes in that package and nothing else. [52, pg 256]

"A package should not have multiple reasons to change. Any change made to the software system should be localized to one package." [52, pg 256]

Identifying if this principle has been followed requires examining the maintenance history and determining the number of packages that required change to fulfill a change

request (bug fix or new feature). If this principle has been followed, a change request will typically involve only one package.

The gain of following this principle is that it makes it easier to track changes. As well it makes changing the code base easier for developers, as they can be reasonably certain that everything they need to change is within a single package and they can safely ignore the rest of the system.

To validate this requires finding a system that has some packages following the *Common Closure Principle* (change requests are isolated to that package) and some that do not (are changed along with other packages to satisfy a change request). Compare the average time to implement a change involving a *Common Closure Principle* package and compare it to the average time to implement a change involving non-*Common Closure Principle* packages.

2.2.6 The Common Reuse Principle

If you reuse one class in a package, you reuse them all. [52, pg 255]

Classes that tend to be used together should be in the same package. Within this package, they should have many dependences on one another. Any change to any class in a package requires a new release, with required distribution and validation, of the whole package (such as a JAR or DLL). This is intuitive as a package is the smallest unit of release; individual class files should not be released. Additionally, any change in a package requires all packages that depend on it to be revalidated, therefore unnecessary classes should not be included. Classes in a package should be inseparable. [52]

A slightly different interpretation of this principle could be that for any functionality the package provides, **every** class in the package should be involved in fulfilling it. This would be helpful when deciding when to break up a package (e.g. consider a package of 10 classes, 2 of which aren't used in a specific operation. These two classes should be moved

into a new package, which the original package can then use).

Identifying if this principle has been applied in a project would entail comparing the coupling of classes within packages to the coupling of classes in different packages. Numerous techniques are available for evaluating coupling [40, 56, 61, 32].

The gain provided by this principle is that validation, release and distribution are all simplified when a package is treated as a unit.

Once packages had been identified which follow this principle, validation could be achieved by comparing the maintenance time, measured either by bug count or by developer hours spent, devoted to packages that follow this principle to those in the same system that do not.

2.3 Software Measurement

Arguably the most interesting of the six Principles, the *Stable Dependencies Principle* and *Stable Abstraction Principle* allow developers to automatically evaluate systems. Interpreting the results can be challenging without an awareness of the limitations of measurements.

The purpose of establishing and tracking measurements is to provide a means of evaluating how well a particular activity or process is performing against a set criterion. As Lord Kelvin said in "Electrical Units of Measurement" (1883):

> "In physical science a first essential step in the direction of learning any subject is to find principles of numerical reckoning and methods for practically measuring some quantity connected with it. I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it, but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meager and unsatisfactory kind; it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the stage of science, whatever the matter may be." [2]

Often a sign of maturity in a branch of knowledge is a well established set of measurements standards. Even the layperson understands the meaning of velocity (distance divided by time, a physics measurement) and the Celsius scale (a temperature measurement based on the freezing and boiling points of water). As software engineering comes of age, there is an ever increasing need to establish measurements which quantify properties of a software system.

Work done in this area, such as uncommented lines of code and cyclomatic complexity, improves software understanding [54]. This has led researchers and their industrial counterparts to search for measurement that provide insights on other characteristics of a software system.

2.4 Configuration Management

An understanding of some of the issues of configuration management is needed to understand the context of this work and these are presented below. Whereas software maintenance consists of engineering tasks carried out after software has been delivered to the customer, configuration management involves control and tracking activities that begin with the commencement of the project and end when the software is taken out of service [60].

Preventing the chaos that uncontrolled change could result in can be seen as the primary purpose of configuration management. It involves assessing the impact and cost of a change, deciding if that change should be implemented, and rebuilding the software after the change is applied. Changes are managed throughout a product's life-cycle [62].

Configuration management is closely aligned with quality assurance; the usual path of creating software is design to development to quality assurance to configuration management [60, 62].

It is typically the case that software exists in different versions, for multiple platforms

or incorporating client specific customizations. Configuration management is responsible for keeping track of the differences between these variants, and for deriving them in a controlled manner. Sometimes it is also responsible for customer delivery [62].

Variance in terminology unfortunately requires multiple terms to be introduced, although usage will be consistent within this work.

Baselines

> "A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis of further development, and that can be changed only through formal change control procedures." [1]

As a project moves through and beyond the prototype stage, those involved learn about the application domain. As knowledge increases about specific issues, better approaches become apparent and lead to changes away from the initial approach. A very readable article on acquiring domain specific knowledge is "The Five Orders of Ignorance" by Phillip G. Armour [28].

Declaring a baseline allows a process greater control over changes at the cost of flexibility. The baseline becomes the accepted state, and formal procedures must be followed in order to change it. This stands in stark contrast to the rapid, sometimes reckless development characterizing the prototyping phase. The advantages this offers includes less volatile behavior and a more predictable evolution. However changes are much easier before a baseline is put into place and the formalization will slow down the the system's growth [60].

There is some disagreement on when the baseline should be imposed. Many feel that the baseline (as well as quality assurance and configuration management) does not become involved until after the initial development (when the project is released and enters maintenance) [62].

The baseline marks a change in a product's evolution. It ends the wild hacking of the early development, and put a rigid change process in place.

With respect to packaging, baselines are highly relevant as it will also include information about how the different packages depend and interact on one another. This needs to be formalized so that developers know what the standards are within the system. This should take place when the system is initially packaged, often when functionality from one system is needed to be generalized to support a product family.

Identification/Planning

As a integral part of the software process, configuration management planning is carried out along with the overall project planning [62].

This task involves the naming and organization of basic objects (such as a section of requirement specifications, the code of a module, or a suite of test cases) and aggregate objects (collections of basic objects). Each is detailed with a name, description, and a list of resources. Relationships between objects must also be identified. Finally, how an object evolves must also also be tracked (using tools such as CVS) [60, 38].

Beyond this the likelihood of change needs to be identified (what should stay stable, what can be changed at will) [34]. This work is especially relevant to this last issue, as our intention is to validate a technique for determining likelihood of change. Planning is further relevant to packaging during the period of time in a system's evolution where decisions are being made about how to package the various functionalities of the system and determine their inter-dependencies. These decisions will change over time, but an initial state needs to be determined, and as mentioned above used as a baseline.

Often this also involves the definition and maintenance of a database tracking information about the configuration [62].

Version Control

> "Configuration management allows a user to specify alternative configurations of the software system through the selection of appropriate versions. This is supported by associating attributes with each software version, and then allowing a configuration to be specified [and constructed] by describing the set of desired attributes." [36]

Versions of a system are the complete state of all parts at a certain point in time. Within a version there can be multiple variants (e.g. a version of Linux has variants for various architectures such as Intel or PPC) [60]. Many versioning systems and notations exist. An example of such is presented in section 3.3.

It is possible to further break version control down into Promotion, Release, Branch, and Variant (see glossary for definitions) [34].

Lehman suggests that an interleaving of releases, where one release changing system's functionality is followed by a release repairing faults, is a strong approach [50]. The Linux kernel follows such an approach, as detailed in section 3.3.

Packaging allows the codebase of a software system to be used in far more dynamic ways then historically allowed by monolithic architectures. The isolation of concerns and clear interfaces make it much easier to pull out a set of functionality from one system and use this as part of another. Version control is the formalization of these various configurations and in a packaged system, the version that a user installs will be expressed as a collection of specific package releases.

Change Control/Process Management

This is simply a procedure that is followed to implement changes. This is required after a baseline is declared, but is sometimes implemented before. The change process can be simple or complex, depending on the organization, size and age of the system.

If a change is approved (after considering its validity, technical impact, cost, and strategic and organizational impact), a "change order" is generated which details the change, constraints to be respected, and criteria for audit and review. The change is "checked out", then "checked in" after passing tests and being reintegrated with the project. Proper records of changes made are needed. In the simplest form this could simply be a change log at the beginning of each source file.

This process allows both access control (who can do what to the code) and synchronization control (while one person is working on a section of the project to implement a change, others wanting to work on that section must wait). Change control becomes more formalized and rigorous after the baseline is set, and even more formal and rigorous again after customer release [60, 62].

General policies should be formalized, to keep developers on track and to reign in any rogues. Notification events (who gets told when something changes) can also be specified [34].

Configuration Auditing is a quality assurance activity that ensures that all the needed steps are taken to deal with a change (enforcing change control) [60].

Change control is highly relevant to packaged systems. Where changes are made is important information for considerations of change propagation, and a history of changes can be used to reconsider dependency relations between packages. If a package changes less often then the package it depends on, it may be worthwhile considering inverting the dependency, using a technique such as Martin's *Dependency Inversion Principle* [52, pg 127].

Reporting/Status Accounting

As changes are made to a system, a centralized repository keeps track of what was done, who did it, when they did it and what will be affected. This keeps a large organization connected and lets the left hand know what the right is doing [60]. Individual components,

work products, and change requests are all recorded and can be queried by developers or management to see the current state [34].

Such a centralized repository effects every facet of development, but as mentioned above packages have a greater need for recorded interfaces, both to manage their change and to advertise their functionality to system developers. If members of the development team don't know how to use a package, its implementation has less benefit to the organization.

System Building/Build Management

This is the task of taking the various parts of a software system and combining them into a whole that executes on a specific platform for a specific customer (creating a release). A problem arises in the discrepancy between logical units of a system, and the actual objects that make up those units. This can be solved by using an incremental approach (where the units are responsible for building themselves) or using a module interconnection language that describes the structure [62]. In situations where an automated build is possible, people working in this area attempt to minimize the recompilations needed [34].

Taking a collection of packages, potentially of various versions, and creating a functioning version to be released to the customer would be a part of this activity.

2.4.1 Integration

After separate parts of a software system have been tested and proven to work, they need to be put together into a working whole. This is also required when different programs need to inter-operate on a computer system being rolled out. Four approaches to this are:

1. Big-bang

   In this approach the system as a whole is deployed, and the developers optimistically hope it will work. When it doesn't, the components causing the problems are identified and repaired. This avoids the overhead of the other approaches for a system

that works properly, or is close to working properly. Unfortunately if it doesn't work properly it can consume large amounts of time and other resources and thus can be considered high-risk.

2. Bottom-up

   This approach is based on the assumption that any software system is a collection of simpler modules brought together to form a larger system. The lowest-level modules are evaluated using tests specifically written for them which provide a specific input and expects a specific output. This simulates the demands higher-level modules would put on them. Once convinced that the low-level modules are working, testing moves on to the components that depend on these and repeats the process. This can be quite an expensive process. Additionally, the product can not be seen until the very end, while with other processes you can see the system partially working as you progress with integration.

3. Top-down

   Works by providing function stubs for all modules supporting the highest-level module in the system (the module that isn't called by anything else). This top level module is tested using the reliable output from these stubs, until determined correct. One-by-one the stubs are replaced by the module they represent. New function stubs are written for the modules these second-level modules call. This continues until the entire system has been integrated. A disadvantage to this approach is that function stubs can't provide the full functionality they replace, otherwise the system would simply use the stub. Therefore a module hasn't been fully tested until the stub is replaced with a module, and this more extensive testing may reveal new bugs in the higher-level module.

4. Inside-out

A combination of Bottom-up and Top-down, this integration approach works by picking a package (often one in the middle of the system), then applying the techniques from Bottom-up and Top-down to work through the rest of the system. [49] This provides both the advantages and disadvantages of these two approaches.

Packaging can follow any of these integration approaches. Each of them is more straightforward for packages as it is expected that every package will have passed its own unit tests before integration. This focused purpose for each package, and individual validation promises to make the system integration easier. In situations where system integration fails, the problem is determined, then tests for this issue are added to the package unit tests. Such a process has the intention of making future integrations easier, to the point where it becomes reasonable to attempt Big-bang integrations.

2.4.2 Change

Once a system is integrated, the problem is keeping everything working together. Any changed element has the potential of bringing the entire system to its knees. The Bottom-up and Top-down approaches outlined above allow us to somewhat isolate system changes. Only modules lower then the changed module need to be considered when applying the Bottom-up approach, while conversely only the higher modules need to be considered when applying the Top-down approach. Using the exact same tests, and comparing them to the previous results (called regression testing) yields helpful information in tracking down the source of the problem [49]. When the difficulties brought about by change are outlined, some naively ask "why not just leave the software alone and stop changing it?" Alas, it is the nature of software to change, and change it will. Some sources of change include:

1. New business or market conditions dictating change (to product requirements or business rules)

2. New customers with unique demands on the system

3. Reorganizations that change project priorities or team structure

4. Budget or scheduling constraints [60]

5. Changing environment (hardware or software) system is operating within

But beyond this as software is used, the possibility of subtle bugs presenting themselves is ever present and will necessitate change to fix them.

Packages change just like any other piece of software. The advantage of breaking a larger system into smaller, focused units is that it helps gather the code devoted to a specific concern in one place, which makes implementing changes easier. Regression testing coupled with unit testing can provide a fairly convincing validation of packages.

2.4.3 Delivery

Of some interest when considering configuration management is how the software will actually be put into the customers' hands. In general there are three types of customers, each with their own considerations:

1. Single or limited number of users

   In this case, the software is typically heavily customized for these particular users, is delivered on some sort of media, and usually the developers take responsibility for the installation.

2. Retail customers

   These is that vast majority of software purchase, where a user buys a box that contains CDs, introductory information, and a manual. The customer is responsible for installation, although the vendor should be available to help with difficulties.

3. Electronic delivery

> For knowledgeable users this requires the software to be retrieved using such methods as FTP, e-mail, or other such electronic delivery methods using no media. Users are usually expected to install, configure and customize such software themselves. [49]
>
> A more modern approach, web-based services is a form of electronic delivery that involves deploying software which can be accessed using a standard web browser or other client. This allows the software to actually be running on the suppliers hardware, with a carefully specified user interface connecting this to the user.

Each of these delivery methods must be considered, and can often provide a justification for some of the complexities configuration management entails.

Electronic delivery is the ideal vehicle for providing updates. By breaking a system into packages, a smaller amount of data needs to be transfered to the customer for updates (just the packages in which changes were made). In the unlikely event that updates were provided on media or through retail channels, these gains would still hold (updates might fit on a floppy instead of a CD, or a CD instead of a DVD).

2.5 Related Work

Packaging is treated under the theory of configuration management in software engineerings texts [34, 36, 49, 60, 62]. Packaging in the setting of software evolution was studied by Oreizy [57] and by van Deursen et al. [66].

Our work was originally inspired by Martin's account of packaging in object-oriented software development [52]. Although we have concentrated on the Linux kernel and IBM's Eclipse IDE, a study of other architectural settings would provide much-needed perspective, as pointed out by van Deursen et al[66].

Previous work by the Software architecture Group at the University of Waterloo involved the generation and analysis of source-based data, and was useful for this work. Base

facts were extracted using compiler techniques [47, 68]. We considered Bowman and Holt's work [31] in the decisions of how to analyze the Linux kernel and Godfrey and Tu's work [45, 46] helped during considerations about measuring change.

Martin's principles have been implemented in a commercial software package called OptimalJ (`http://javacentral.compuware.com/pasta/`) and in an open-source alternative, jmove (`http://jmove.sourceforge.net/`).

Chapter 3

Methodology

We desire a method of measuring hardness to change.

To validate such a method test systems are required that have 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 method is reasonable. The more situations studied, the greater the confidence in the method.

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 non-volatile and volatile packages are soft. The second is to explore the software measurement itself looking for correlation between hardness and contextual stability: does the software measurement actually measure what is intended? [68].

3.1 How to Validate Software Measurements?

Using a measurement is usually quite straightforward, however determining if a measurement is valid can range from trivially easy to quite challenging. It is simple to validate meters as a measurement for length. It can easily be shown that something with a longer measurement in meters is longer then something with a shorter measurement. For example, if two men were measured, one being 172 cm and the other being 167 cm, it would be clear that the man who was 172 cm is taller then the other.

An example of a measurement that is more challenging to measure would be the relative hardness of two substances. While it's possible to see with the naked eye that one man is taller than the other, and their relative height, we don't possess an inherent sense that tells us how hard object are. Mohs proposed a scale of 10 commonly available minerals, ordered them in increasing hardness[1], with hardness defined as a harder mineral will scratch a less hard mineral, then used these as benchmarks for hardness. Such a scale, while useful, is certainly less intuitive than meters, as it has unusual properties such as being neither linear nor logarithmic[2] [19]. How to determine the hardness of minerals outside the scale is also uncertain. An alternative measure of hardness would be how much pressure is require to crack or make an indentation in a sample of a given size. Since each of these measurements would give different results and a different scale, it can be challenging to name one of them to be the definitive method of measuring hardness.

An example of an intuitively obvious software measurement would be using lines of code (LOC) as a measure of the size of a software system. Most people would agree that a piece of software that is 1000 lines of code is larger than one which is 500 lines of code. Even this simple example can demonstrate potential pitfalls of measurement. Many languages allow multiple instructions on a single line, therefore if the 500 line program had more than 2 instructions on every line it would actually be a larger program, violating the measurement.

[1] with talc being 1 and diamond being 10 [19]

[2] "corundum is twice as hard as topaz, but diamond is almost four times as hard as corundum" [19]

In spite of this potential inaccuracy, we still view lines of code as a reasonable measurement and find it, or a modified form, in use in academia and industry [45, 68, pp 42-53].

When attempting to validate a software measurement that is more complex than lines of code, a compelling approach that presents itself is to compare the measurement with other techniques for measuring the same quality. If the software measurement can provide similar measurements in a more straightforward fashion, then we can conclude that we have a valuable new measurement tool. [68, pg 410]

In this work, we proceed in such a manner to validate Martin's *Stable Abstraction Principle*. As mentioned in the introduction, Martin proposes a measure of difficulty of changing (referred to as instability) any individual package within a source base; an alternative measure for the same quality is needed.

Therefore we seek to measure Martin's instability using the empirically observed rate of change in various packages of a software system and comparing this to the predictions of his software measurement. Measuring instability using Martin's measurement is a simple case of picking a representative version and applying it.

Measuring the rate of change empirical is more time consuming, which is the advantage Martin's measurement provides. The most straightforward approach, and the one followed, was simply to inspect each release, and compare each package with that of the previous release. A tally is maintained for each package, and its rate of change is simply the number of releases it changed between compared to the number of releases it appeared in.

We compare the volatility, measured in terms of actual change between releases, to the contextual stability as defined by Martin. If Martin's assumptions are correct, we should find at least some correlation: volatile packages should tend to have low contextual stability, and non volatile packages should tend to have high contextual stability.

Using this technique we choose appropriate test systems and apply this approach.

3.2 Choice of Guinea Pig

[3] In software reverse-engineering and evolution studies the term guinea pig is used to refer to subject software systems accessible as source code.

Our criteria for systems to empirically evaluate change were these:

1. Maturity

   We are interested in software systems that have existed long enough to have gone through many release cycles, have had numerous unanticipated new features added, and have experienced prolonged maintenance activities. These are all challenges that a typical software system faces, so a test system that has avoided some or all of these issues will not give us appropriate results.

   Additionally, software system growth can be erratic at different points in its development [29]. A system experiencing early-release frenzy would not be a reasonable candidate.

2. Size

   A system of trivial size (e.g. Hello World) will not be reasonable for this study, simply because its small size allows it to be easily understood, maintained and released by developers. This negates any advantages packaging would offer.

3. Well Engineered and Healthy

   We want the systems examined to be properly designed and engineered. This was evaluated using the general reputation of the projects and the people working on them.

   We wish to avoid systems that are displaying signs of software aging as defined by Parnas [59].

[3] This section was adapted from [35]

4. Source Code Availability

   While it may have been possible to purchase access to a code base, or to have entered into an agreement with a company to analyze their proprietary system, it was felt that the benefits of doing so were minimal. Instead we focused on open-source systems which tend to have a highly available source history. We required the system to have many versions available, over its entire development history.

5. Appropriate Packaging

   The most difficult desired attribute that we would like is a system that has been packaged according to Martin's Packaging Principles [52]. The less desirable degrees of fulfilling this consideration included finding a system that was object-oriented and had been packaged according to non-Martin principles, or a system that had been broken into parts that could be considered analogous to packaging.

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

3.2.1 Linux as Guinea Pig

While it does not satisfy all of the above criteria we felt 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 most of its releases are available for download. It is publicly acknowledged as a high quality software system [6, 26]. At the time of the experiment 499 releases were available. A representative release (version 2.4.18) consisted of about 850 packages, with a size of approximately 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 behavior, abnormal termination, increased maintenance difficulty,

and an increase in fault introduction [59]). Indeed, some research[45] has shown it to maintain super-linear growth well into development, a fact that apparently violates one of Lehman's laws [45]. It has been studied by software evolutionists already [67, 31, 45, 50].

It is not easy to assess the degree to which Linux is packaged according to Martin's principles. Linux is not object-oriented, and so there is an immediately visible development mismatch with Martin's development methods (which assume the systems are object-oriented). 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 managed by specific developers or groups of developers. This is analogous to packaging in the sense we are using it. From this view Linux has had about 1500 packages, in its development history (not all of which are present in any given release).

The are some easy criticism of Linux as a test system. These include:

1. Not Object-oriented
2. Not Packaged
3. Not Developed using Martin's principles

It is possible for some to claim that Martin intend these ideas to be applied exclusively in a Java or C++ setting, since those are the languages used in his text. Following this line of thought, since the Linux kernel is written in C it might be argued that it is an inappropriate test bed.

It needs to be pointed out that object-oriented languages make certain program structuring *easier*, but it does not allow or disallow techniques in absolute terms. If something is

[4]E. S. Raymond discusses in Homesteading the Noosphere the strong aversion to forking (reuse) that exists in the open source community [6]

possible in an object-oriented language, then by applying developer discipline it is possible in a procedural language.

While Linux is not packaged in the strictest sense, the source code is divided into many subdirectories. These directories collected code that tends to be tightly coupled, and are often assigned to semi-autonomous development teams. This is a similar role to what packages play and it is fair to say that they can be considered like packages.

Finally, due to its relatively recently release, there were no systems available to the authors that followed the packaging principles laid out in Martin's book[52]. Therefore it is unfair to make a critism of Linux that would be true for any other system that could be studied.

3.2.2 Eclipse as Guinea Pig

Our other guinea pig was Eclipse, an open-source Java IDE that is highly extendable due to its plug-in architecture.

Eclipse had many of the same qualities that made Linux attractive.

With the project having undergone steady development in the open-source community for over 2 years (Eclipse was released to the Open-Source community on Nov 7th 2001 [13]), we felt comfortable considering this mature. It was felt that its markedly different development style compared to Linux made the two good test systems.

Beyond this, as a fully featured IDE it is clearly of non-trivial size. Coming out of IBM we feel confident considering it well engineered. It is open source, therefore the source code is available. Finally, it was felt that the plug-in structure which was used even for the core elements of the program were reasonably analogous to packages.

Eclipse was not developed using Martin's principles, however as mentioned above there were no systems that presented themselves which had been and fulfilled our other criteria. Additionally Martin rationalizes his approach from the pressures dependency places on the change process, and such pressures would certainly be present in both of these systems.

3.3 Linux Kernel Development and Versioning

As the best known open-source project, Linux is a free Unix operating system. Since early in its history it has been released under the GNU public license [16]. The implications of this is that the source code must accompany each release, and that anyone who makes modifications to this source code must release these modifications along with the original anytime they distribute or publish the software; distributing only the binaries of a program is not acceptable.

Under the leadership of Linus Torvalds, this project has grown to the point where it threatens the largest of proprietary operating systems [5, 23]. Allowing anyone to freely use the software with the responsibility of giving back to the project has allowed strong growth [45], and respected code quality [6, 26],

The Linux kernel versioning follows a Z.Y.X numbering scheme. If Y is an even number it is considered a stable branch and only bug fixes are applied to it. If Y is an odd number it is considered an unstable branch and new features are introduced and innovations are applied. This follows Lehman's recommendation for following a repair release with a feature addition release [50]. At some point in the unstable branch, a feature freeze is announced, after which no new features are considered until after the release of a new stable version. Following this a code freeze is eventually announced, which only allows severe bug fixes to be applied. Shortly after this, a new stable version is released and development continues. It is also possible to have Z.Y.XpreI where it is the Ith "pre-release" of Z.Y.X. Finally, the stable release is created, after which a new unstable branch is created and the process continues [3].

There has been some unusual movement between versions. Version 1.2.0 was the first time the stable/unstable versioning system was used. It was based on the unstable version 1.1.95. At this point in time, development focused on the stable version exclusively for about 3 months, then the unstable "product line" was created again as version 1.3.0. 1.3.0

was unusual in that it was created as an independent release rather than allowing the possibility of deriving it by patching an older version. Some Usenet posts[5] conjecture that it is most similar to 1.2.10 which was released on the same day, however the two have dramatically different byte sizes when archived, so the similarity is hardly absolute. The advice offered was to download the complete release rather than trying to patch an old version, further indicating that this release was significantly different from its predecessors. The stable version 2.0 was based on unstable version 1.3.100. Unstable version 2.1.0 was based on stable version 2.0.21. The two lines diverged at this point (Sep/Oct 1996) and they began maintaining parallel development streams. Table 3.1 summarizes this information.

Kernel Version	Based on
1.2.0	*1.1.95*
1.3.0	UNKNOWN[6]
2.0	*1.3.100*
2.1.0	**2.0.21**
2.2.0	*2.1.132*
2.3.0	**2.2.8**
2.4.0	*2.3.51*
2.5.0	**2.4.15** (exactly the same)

†Stable in **bold**, unstable in *italics*

Table 3.1: Linux Kernel Version History

Stable version 2.2.0 was based on unstable version 2.1.132. The 2.0.Z line continued in parallel to this to, terminating with version 2.0.39. The new unstable version 2.3.0 was based on stable version 2.2.8, which had underwent exclusive development for about 3 months (same as previously).

[5]These can be found by using the search term "linux 1.3.0" at `http://groups.google.ca`

[6]Presumably some version 1.2.X - 1.2.10 was released the same day as 1.3.0, but they are quite different, other versions of 1.2.X were released at later dates

This listing ends in 2000.

The change log for 2.5.0 says that it is exactly the same as 2.4.15, so they diverge at that point. 2.3.51 was used as the basis for the pre-release sequence for 2.4.0 and was incidentally the end of the development line for 2.3.X.

At the time of this writing, the latest stable release is version 2.4.22, which the current unstable release is 2.6.0-test9 which is the end of the 2.5 unstable branch (they are preparing to use it as a basis for 2.6, the new stable branch).

3.4 Eclipse Development and Versioning

Eclipse was begun by IBM, who still play a lead role in its development, but as an open-source project now has developers from all over the world.

The Eclipse charter describes the project as:

> "a collaborative software development project dedicated to providing a robust, full-featured, commercial-quality, and freely available industry platform for the development of highly integrated tools.
>
> Eclipse is a kind of universal tool platform - an open extensible IDE for anything and yet nothing in particular. The real value comes from tool plug-ins that "teach" Eclipse how to work with things - Java files, web content, graphics, video - almost anything one can imagine. Eclipse allows tool builders to independently develop tools that integrate with other people's tools so seamlessly you can't tell where one tool ends and another starts.
>
> The success of Eclipse depends on how well it enables a wide range of tool builders to build best of breed integrated tools. But the real vision of Eclipse as an industry platform is only realized if these tools from different tool builders can be combined together by users to suit their unique requirements, in ways

that the tool builders never even imagined.

The mission of the Eclipse Project is to adapt and evolve the Eclipse technology to meet the needs of the Eclipse tool building community and its users, so that the vision of Eclipse as an industry platform is realized." [11]

Much like Linux development, Eclipse is a meritocracy. As people make more contributions of greater quality they are given more decision making power within the project. The break down of roles provided in the charter are:

1. Users

   These are simply any person who wishes to use the freely available IDE. They are encouraged, but not required, to participate in interacting with other users in newsgroups and on mailing list and to provide bug reports. [11]

2. Developers

   These are simply users who have contributed code or documentation. Developers are expected to be pro-active in reporting problems, and are encouraged to participate in user discussions and design decisions which involve their area of expertise. [11]

3. Committers

   "These are developers who have consistently provided a beneficial impact on the project and are therefore offered more influence and responsibilities. They are given access to the official source repository (hence the name) and are required to monitor newsgroups, mailing lists or other user/developer discussion areas. They are expected to respond in a timely fashion to issues requiring their input and are often given a vote on issues that affect the project." [11]

4. Project Management Committee (PMC)

> "The Eclipse PMC is responsible for the strategic direction and success of the Eclipse Project. This governing and advisory body is expected to ensure the project's welfare and guide its overall direction. The PMC is responsible for overall development direction, conflict resolution, development processes and infrastructure, and the overall technical success of the project."

Nominations for a new PMC member can be made by any existing member, and must be approved by an unanimous vote. The goal is to keep this committee quite small. [11]

Another interesting facet of the Eclipse project is that it is built as a plug-in architecture. As the fundamental building blocks of the system, Eclipse can be considered the sum of its constituent plug-ins, where each plug-in contributes functionality to the resulting system. It dynamically loads plug-ins from the subdirectory eclipse/plug-ins as they are needed. Each plug-in contains a manifest (plugin.xml) that describes its contents to the Eclipse runtime. [25]

It was considered to include third-party developed optional plug-ins in our experiment. It proved to be challenging to reconcile their release schedule with that of the core packages and to determine whether they changed based on changes to depended upon packages or not. It is entirely possible that developers of such plug-ins might refuse to support a new version of the core IDE. In such a case the plugin is isolated from changes in the new version. Incorporating these additional plug-ins is left as future work.

The versioning of Eclipse is more straightforward than that of the Linux kernel.

FIXME - Add some story about dev history

Version	Based on	Date Released
1.0	N/A	Nov 7, 2001
2.0	1.0	Jun 27, 2002
2.0.1	2.0	Aug 29, 2002
2.0.2	2.0.1	Nov 7, 2002
2.1	2.0.2	Mar 27, 2003
2.1.1	2.1	Jun 27, 2003

Table 3.2: Eclipse Version History

[12]

3.5 Measuring Volatility

[7] 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 regime where only changed material is released (the patch style) it is the probability that the package is included in the next release. For a CVS-managed regime [7], 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[29]. Equally familiar to programmers is the presence of never touched, 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 catastrophic restructuring. 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,

[7]This section was adapted from [35]

for example, when a module overburdened with responsibilities is refactored by splitting into two coupled modules. An accurate analysis of origins [44, 45] 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 therefore 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 (as mentioned in the configuration management section), is needed. These are the very phenomena we are trying to measure.

A similar defense can be applied to the criticism that our method does not differentiate between the size of change. Large architectural changes and trivial non-functional changes (such as correcting the spelling in a source comment) are treated the same.

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 that 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, p) 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 $C_p = \{r \in \mathrm{R} \mid p \text{ is changed for } r\}$. This is a subset of the set R_p of releases of p, that is $R_p = \{r \in \mathrm{R} \mid r(\mathrm{p}) \text{ is defined}\}$. 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 or in the Eclipse project). 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.

3.6 Measuring Contextual Stability

[8] For reasons such as those detailed in section 3.5 FIXME - is this the right reference? we would prefer to measure the hardness of packages, but the actual difficulty of change

[8] This section was adapted from [35]

cannot be assessed without detailed knowledge of the change process and development history.

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 non-volatility) 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 and assess the contextual stability for that particular release.

We chose release 2.4.18 as being a typical recent release of the Linux kernel.

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 [47, 65].

Xinyi Dong's 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 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 second 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 separate 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.

We decided to use version 2.1 of the Eclipse project to measure contextual dependencies. The structure of the Eclipse project makes determining dependencies quite straightforward, as every package includes a XML file called build.xml which explicitly states the package (plug-in) dependencies in a tag. In addition to this, every package has an implicit dependency to org.eclipse.core.boot and org.eclipse.core.runtime. Simply by parsing the build.xml file and adding the implicit dependencies, we were able to easily construct the dependency information for each version of Eclipse, and use this to calculate the Instability for each.

Chapter 4

Results

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.

4.1 Linux Results

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.

For the Linux code base, we obtained approximately 500 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 textual changes[1] to source files, and any movement of

[1]With the unlikely exceptions mentioned in Section 3.5

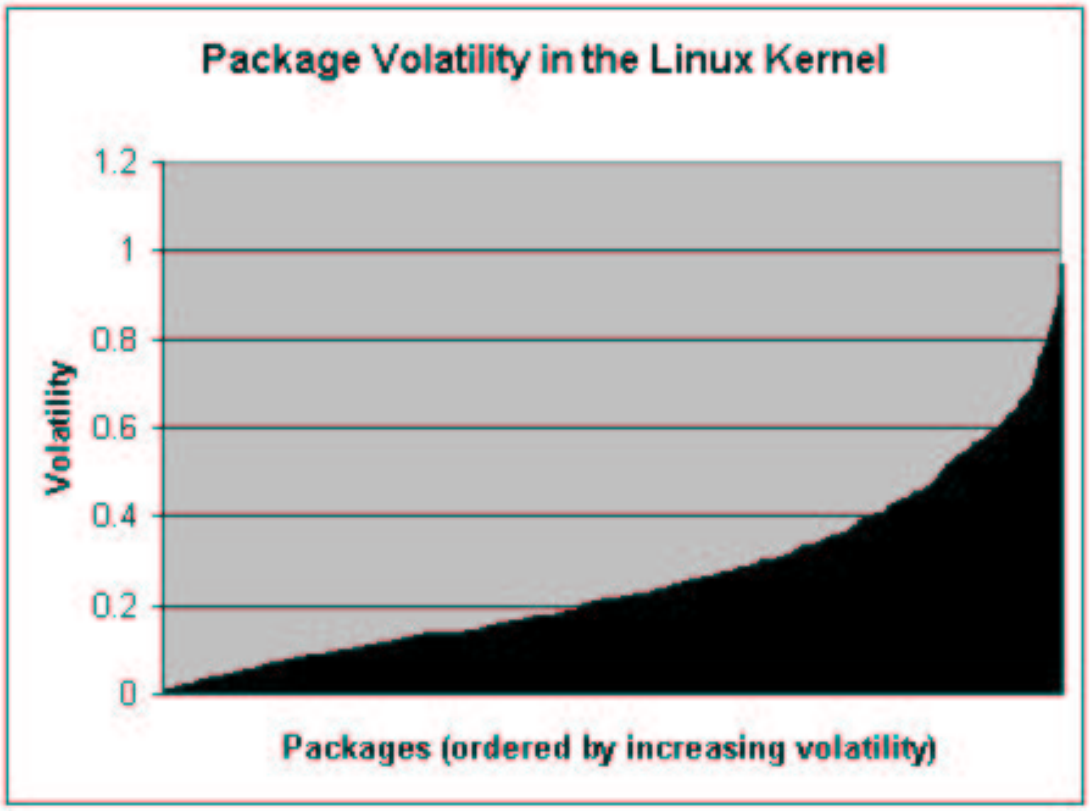

Figure 4.1: Volatility Profile

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.

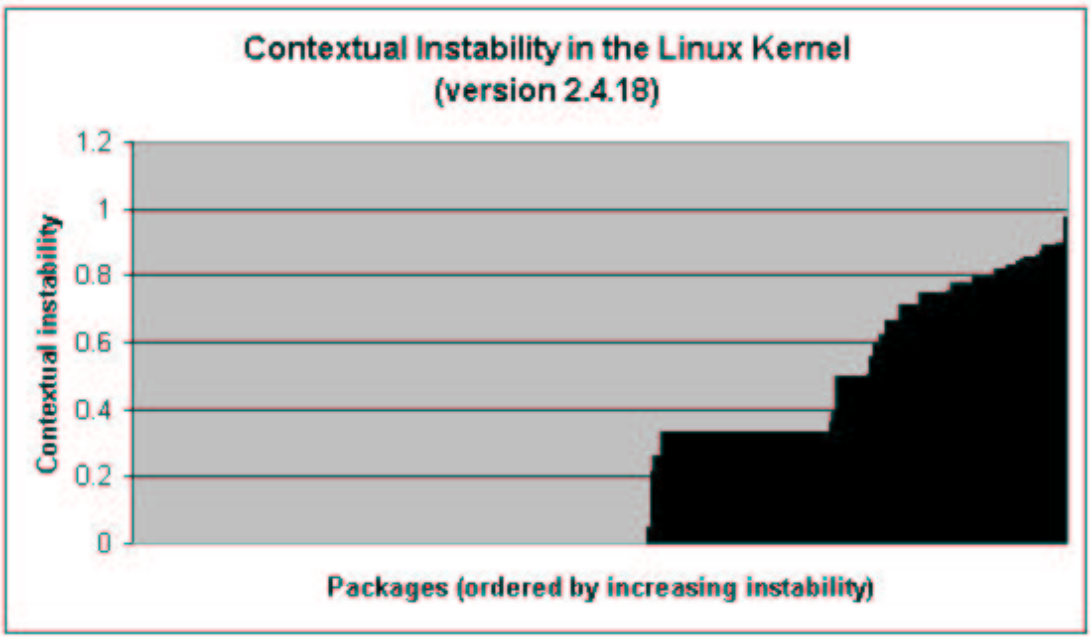

Figure 4.2: Linux Stability Profile

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 [46] for discussion). A large number of packages have contextual instability 0.33 resulting in the plateau seen, typically because of one incoming and two outgoing dependencies.

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 previously, there is no origin analysis. [44]

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 the system as a whole. Build scripts (having extension `sh`) and `Makefile` files 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 are considered 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 4.1. 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.

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 in figure 4.2 the very shape of this it is immediately apparent that contextual stability measures something other than volatility for the Linux system.

Following this the data presented in figures 4.1 and 4.2 were combined and presented together in figures 4.3 and 4.4. In each of these figures both measures are presented for every package that was present in 2.4.18. It can be seen at a glance that when the packages are sorted by volatility or stability that there is low correlation between the two measures. No further numerical analysis was needed.

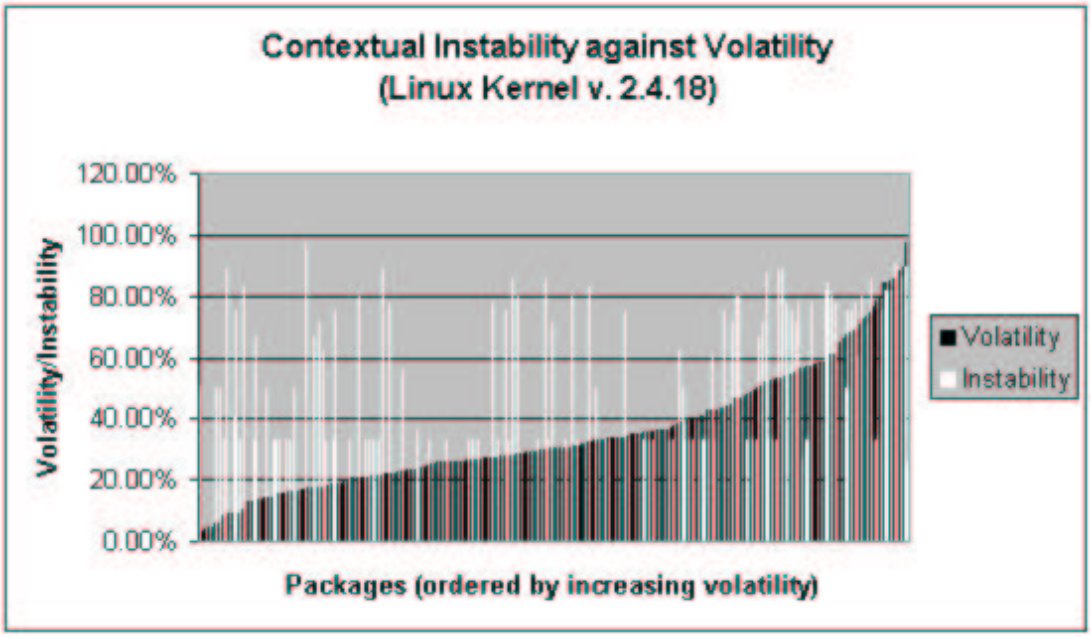

Figure 4.3: Volatility / Stability Comparison
This is figure 4.1 augmented with the computed contextual instability displayed (in white) for each package in v.2.4.18. If there had been the expected correlation, the white bars would tend to be longer on the right and shorter on the left.

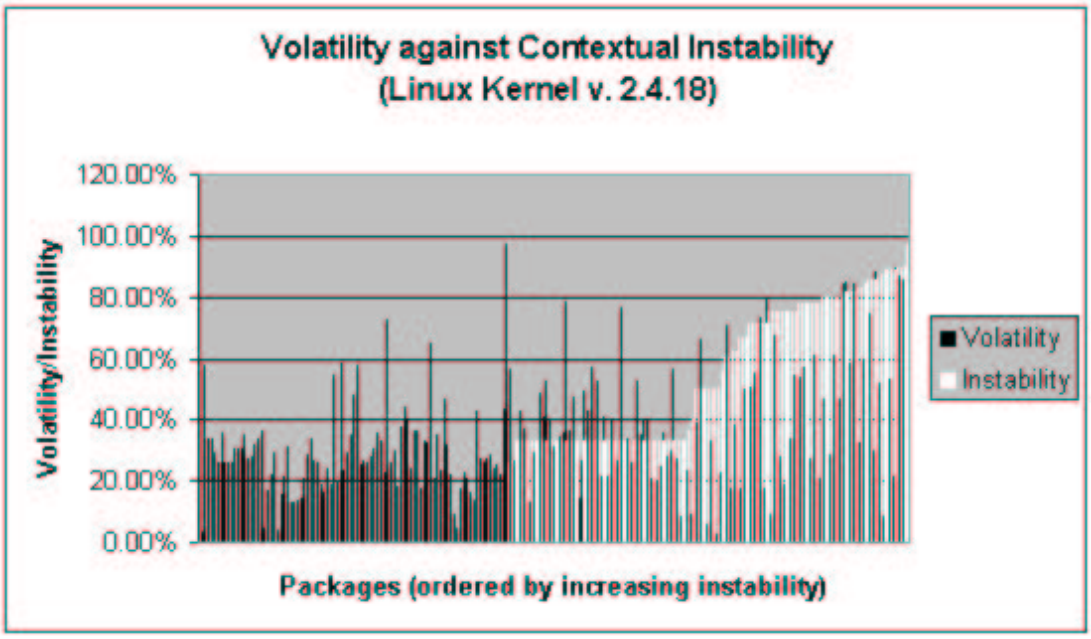

Figure 4.4: Linux Stability / Volatility Comparison
This is figure 4.2 augmented with the observed volatility of each package in v.2.4.18. The data are similarly uncorrelated.

4.1.1 Subsystem Volatility

It may be seen that the four different subsystems differ in volatility. Given that these results, show in figure 4.5, were determined using a long run of versions, spanning many years, these results are not due 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.

Architectural Volatility

Godfrey's previous work on the growth of Linux [43, 46] has shown areas of high volatility in the architecture. We having obtained results that show some correlation between the architectural structure of the system and its change rate. This is displayed in figure 4.5, in which the packages are again displayed in increasing order of volatility, but now their architectural classification (subsystem) are broken into 5 parts.

4.2 Eclipse Results

We used the 6 official releases of Eclipse at the time of running the experiment. These were 1.0, 2.0, 2.0.1, 2.0.2, 2.1, and 2.1.1 (see Table 3.4). Since this time version 2.1.2 was released. In these we found 92 packages.

Version 2.1 was used to calculate contextual dependencies.

As mentioned in section 3.4 there was some consideration of including plug-ins that had been developed independently of the core packages in our experiment. The rationale

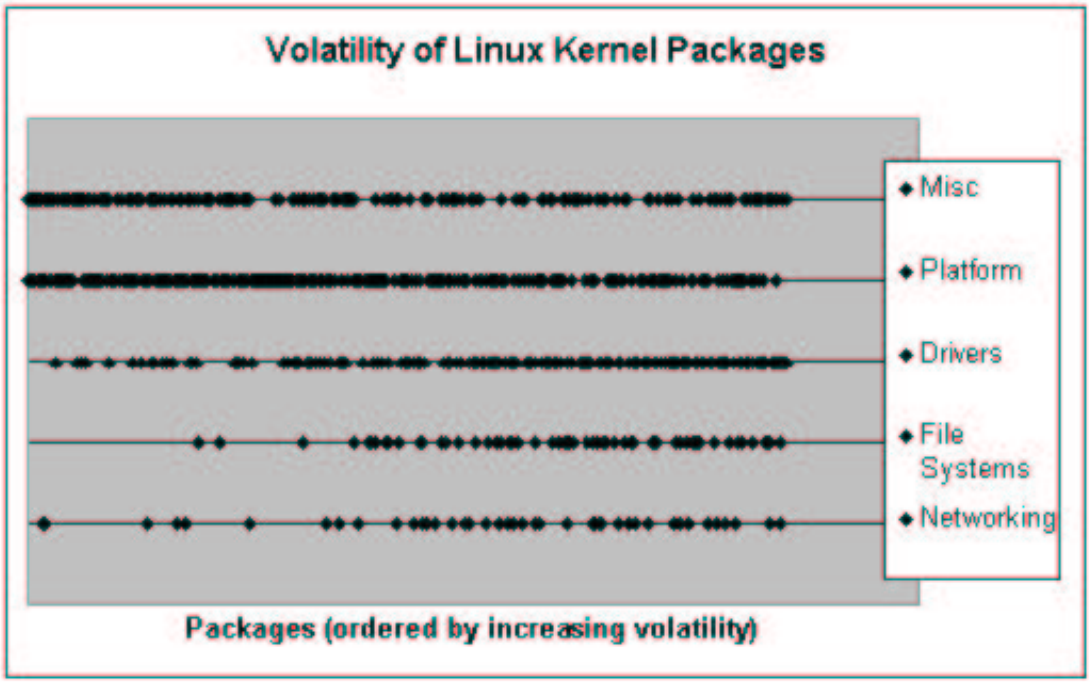

Figure 4.5: Linux Subsystem Volatility Profile
Here is shown the subsystem (on a separate horizontal line) each package (the dots) belongs to; their volatility is indicated by the relative position on the x axis, as in figure 4.1.

for not doing so was that it was highly likely that such packages would follow a release schedule separate from the core packages and that the problem of connecting the two could be problematic, which makes it an interesting project for future work. Consider the case where a core package that an external plug-in depends on changes in a way that breaks the external plug-in. The developer of the external plug-in can easily demand that his users continue to use an older version of the IDE and avoid updating his plug-in if the users accept this. In such a case the plug-in seems to be non-volatile, when in reality its dependency should have made it more volatile that it appears to be.

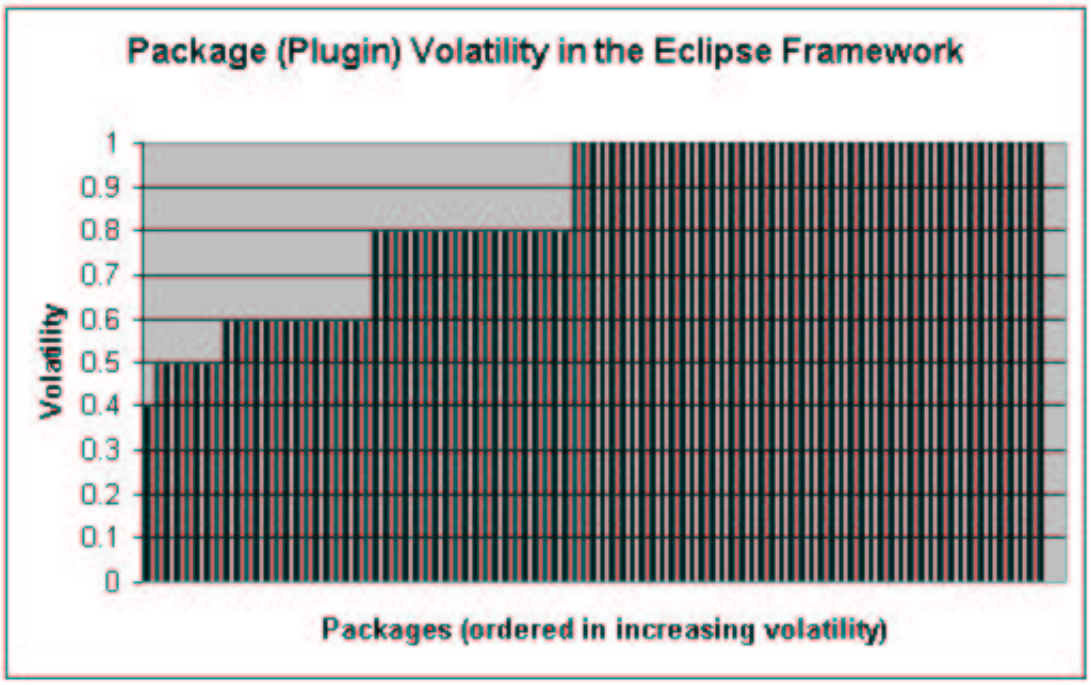

Figure 4.6: Eclipse Volatility

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 25% level is at volatility 0.6, so that a quarter of the packages change less than 60% of the time.

The fact that about half of the results had a volatility of 1 is indicative of heavy development and implies that these packages were changing every release.

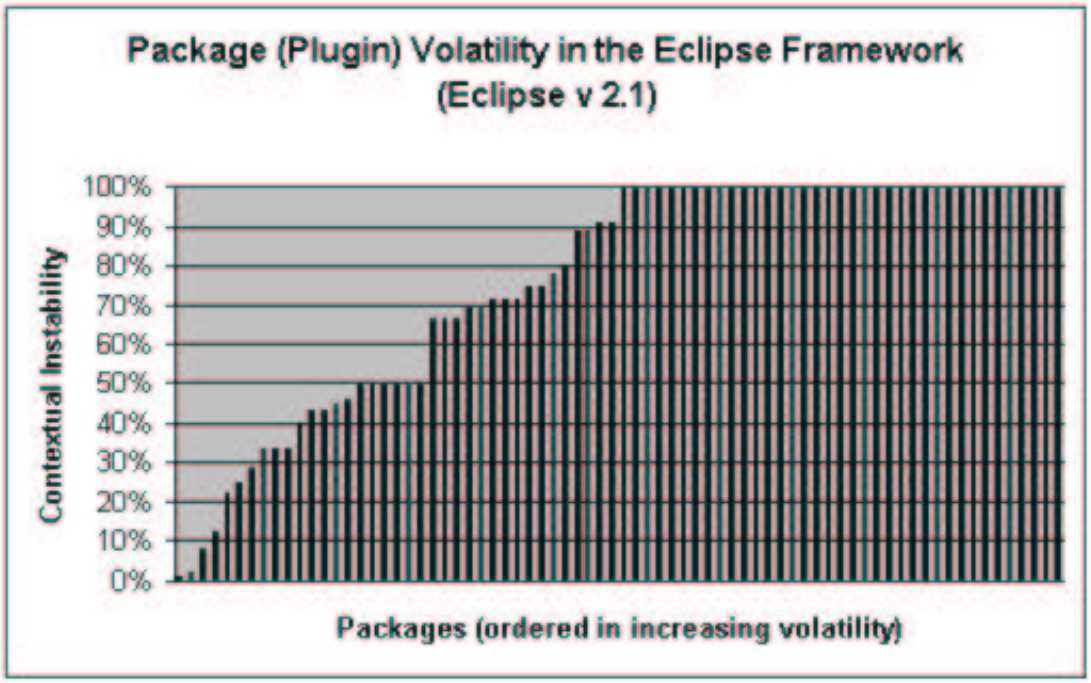

Figure 4.7: Eclipse Stability

Like figure 4.6 it can be seen that approximately half the packages making up Eclipse changed every release.

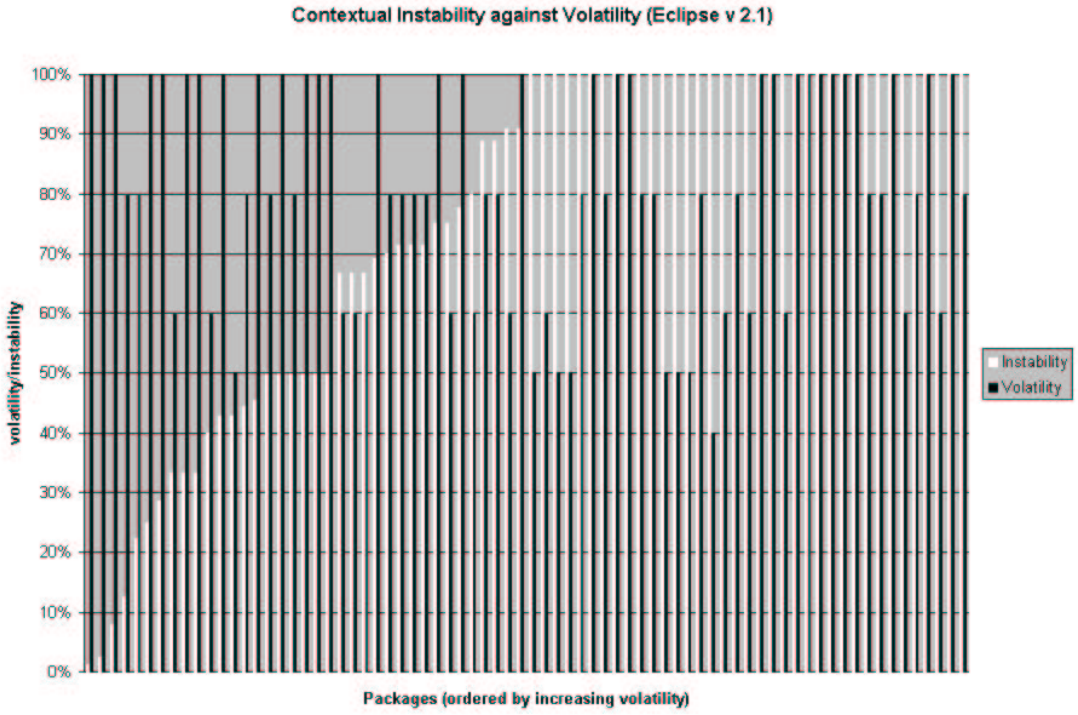

Figure 4.8: Eclipse Volatility/Stability Comparison

When the volatility and stability data are plotted on the same chart we expect that there will be a correlation between the two. After ordering in increasing volatility (the white) we can clearly see that the expected correlation, high stability (black) values to the right and low stability values to the left, is not present.

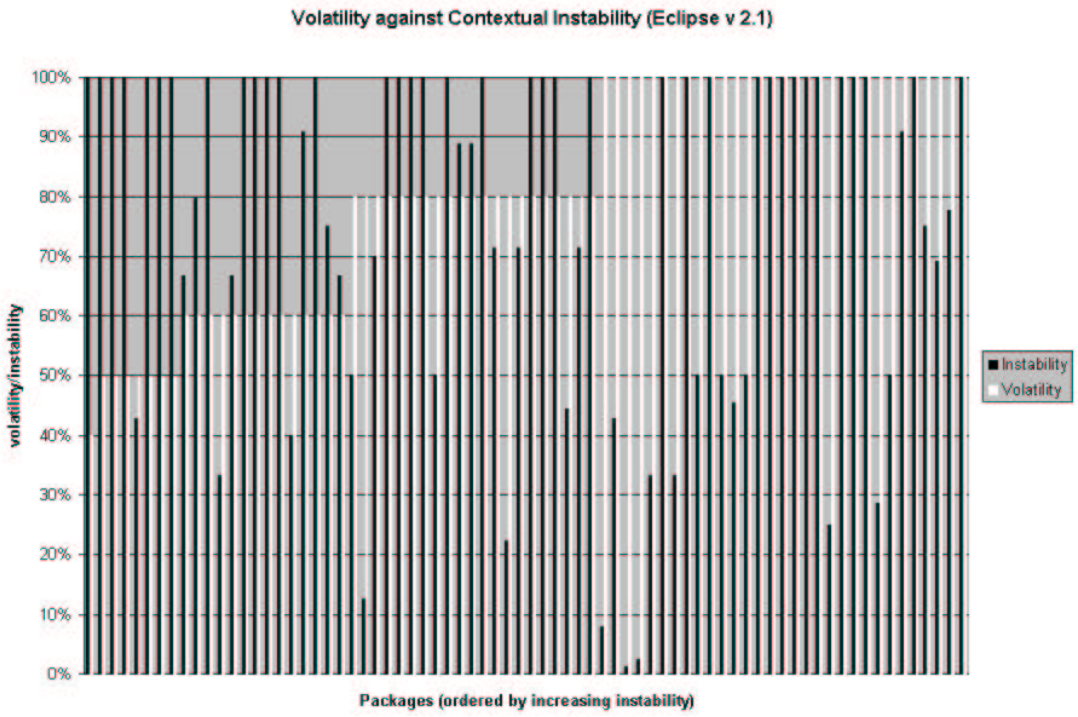

Figure 4.9: Eclipse Stability/Volatility Comparison
After ordering the data in increasing instability (switched to white in this graph) we still fail to see any correlation between the sets of data.

Chapter 5

Conclusion

5.1 Findings

No Correlation

We had sought positive correlation between volatility and contextual instability, since if Martin's assumptions are correct we would have seen that packages which change slowly tend to have more packages dependent on them than they have dependencies on others; and packages which change quickly tend to the reverse situation. To visualize this we displayed the contextual stabilities of packages against the volatility profile (see figure 4.4) and conversely the volatility of packages against the contextual stability profile (see figure 4.3). This work was repeated on the Eclipse project yielding similar results (see figures 4.2 and 4.2). 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 verifying such software measurements.

From the failure to correlate these two measure, and the discussion above in section 3.2, we can conclude that for these systems, 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 Limitations of Results

Martin's "Stable Abstraction Principle" has great intuitive appeal as it seems to balance pressures that we would expect to have a great deal of influence over the change rate of packages. Rather then ignoring his work, a more productive path forward would be to investigate what factors affect the change rate of the packages we saw in these systems to the degree that his software measurement no longer seemed to hold.

It is possible, although in our opinion improbable, that the influence of the open-source development process may have affected the results by introducing such an unknown factor.

5.3 Future Work

During work on the Stable Dependency Principle approaches for validating Martin's other principles presented themselves. These other principles and validation approaches are presented in section 2.2.

Should a system present itself, it would be worthwhile to repeat this experiment on a closed source system.

Additionally, as new software measurements purporting to measure change in the manner of Martin's software measurement present themselves this technique offers a possible validation approach.

As mentioned in section 4.2 work including Eclipse plug-ins, taking into consideration the variable release schedules, would be a reasonable extension to this work.

There is potentially interesting work to be done if this experiment was modified to look at "windows" of releases, rather than complete development histories as we have done. An example of such would entail looking at 10 release before and after a selected release and repeating our experiment on this subset. An alternative technique for windowing would be to look at how a system changed within a major release (such as the 2.4 series for the Linux kernel). It can be assumed that certain things were being accomplished within such a sequence and Martin's metric might predict this on a smaller scale.

5.3.1 Change Factors

It is my feeling that a predictive measurement of likelihood of change for packages exists. The factors which would influence such a measurement include:

1. Dependence relation

   Every item that a package depends on increase the likelihood of change. Change propagation makes this fairly certain, for when a package depends on another there is without a doubt some way that the depended upon package could change that would necessitate a change.

2. Dependents relation

   Every item that depends on a package decreases the likelihood of change. Political pressures from the development teams of these packages would inhibit the depended upon package from changing. It is my belief that Martin exaggerated the impact of this is his *Stable Dependency Principle*. While developers may prefer for the depended upon packages to remain unchanged, in many situations they would also like new features from these packages, and in fact may encourage or demand changes. Most

computer users will recklessly upgrade their systems to try to add new functionality, optimistically expecting the changed system to work perfectly. System administrators are far more skeptical of this process and resist any changes to their systems for exactly this reason; they know the changes introduce problems that must be fix. Developers, known for their optimism, unfortunately fall closer to the user camp on this issue.

3. Importance within system

   More important packages are more likely to change. Again a political issue, it is clear that developers like to impress their peers and that the higher profile a part of the system is, the more likely someone will find and implement an improvement. We see in the Linux kernel code that has remained unchanged for months or years at a time. This code doesn't remain unchanged because it is the foundation of the kernel, its actually the reverse, this is often abandoned code that no one uses, therefore there is no pressure or incentive for it to be improved or changed in any way. There is no motivation to remove it, so it simply remains.

4. Age

   Newer packages are more likely to change then older packages. A recently created package will typically receive more attention then an equivalent package that has existed for a longer time. There is some motivating force that pushed for its creations, and it is highly likely that this motivating force continues to promote change in the package.

5. Popularity

   More popular pages are more likely to change. Closely related to **Dependents relation** and **Importance within system**, the more people who are using a package, the more input there is for bug fixes and feature requests.

6. Size of interface

   The larger the package interface, the more likely to change. A package having a larger interface will provide more services, and thus will have more feature requests. In comparison, a package with a small, focused way to interact with it will be less likely to change as it provides a more concrete service.

7. Size of package

   The larger the package, the more likely to change. Whether measured in files, LOC or bytes, the larger a package the more code it contains, and the more things that could require a change.

8. Number of suppliers

   More suppliers of depended upon packages implies more change. If there are a greater number of organizations supplying depended upon packages, there are more reasons that they may change, in turn propagating a change out to the depending package.

9. Number of customers

   More customers depending upon package implies more change. If there are a greater number of organizations depending upon packages, there are more reasons that they may change, in turn demanding changes from the depended upon package.

10. Domain

    Packages making up applications in different domains will change differently, as will packages that are a part of different subsystems

    As shown in Section 4.1.1 different parts of a system will change differently. As well, the packages making up an operating system would change in a different manner than those making up a database or an office application.

11. Development Style

    Change will be more focused in proprietary development over open source Since the motivation for change comes from one place (management) in proprietary development, it would be expected that change motivation in open source systems, coming from each of the developers, would be more diverse, and hence be more spread out in the system.

Each of these factors will contribute to the likelihood of change to varying degrees. It would be interesting for future work to try to evaluate the contribution of each, and from these characteristics provide a new measurement predicting change.

5.3.2 Magnitude of Change

In our work we considered change a binary occurrence, it either happened or it didn't. It would be interesting to evaluate the magnitude of expected change for each package. The first challenge of such a measurement would clearly be how to evaluate change in a software system, as a very simple change in the source code can have a major effect on the functionality (such as making it stop working), while widespread changes can have no effect on the functionality (such as adding comments or commands that don't accomplish anything).

A simplifying approach would be to consider magnitude of change to simply be change events and assume that on average each of these contributes an equal amount of "package change". Alternatively the change events could be classified (by labels such as "bug fix" or "feature implementation") and each category could be assign a change impact number.

Appendix A

Glossary of Terms

branch A temporary split to a source base which allows concurrent development (often appearing in a high growth software project as the "weekly build") [34, 52]. FIXME - add page number in Martin

"declaration of classes and objects within modules." [30, pg 142]

component "A constituent part" [18], "an artifact that is one of the individual parts of which a composite entity is made up; especially a part that can be separated from or attached to a system; spare components for cars; a component or constituent element of a system" [21]

A family of routines for a given job that doesn't exact a penalty in unwanted generality. Can be safely used as a black box. [55]

Individual parts of a program are completely generic. Instead of having a specialized set of methods and fields they have generic methods through which the component can advertise the functionality it supports. [21]

framework The Gang of Four describe a Framework as a set of cooperating classes that make up a reusable design. The framework dictates the architecture of the applica-

tion, while the developer provides the functionality. [41]

The dictionary defines framework as "a basic conceptual structure (as of ideas)". [18]

free software Software in which the user is able to run, study, adapt, redistribute, improve and release the software. Many of these activities require access to the source code, which must therefore be distributed with the software. [15]

module The result of project segmentation. Each separate, distinct partitioning should have well-defined inputs and outputs. Modules can be tested and maintained independently. [42]

namespace Scope. Basically defines a section of a codebase that can make calls within itself easily, but anything outside must follow a more formal interaction with it. [63]

open source Although there are subtle differences [22, 27], this is basically another term for free software.

package "A (source) package is a basic development unit which can be separately created, maintained, released, tested, and assigned to a team." [37] "The unit of release (e.g. a jar file)." [52].

promotion A version created for developers [34].

release A version create for users [34].

repository A library of releases [34].

software configuration Comprised of source code, executables, documentation and data (basically all information produced by the software process). [60]

subsystem "An abstraction used for aggregating collections of related components in a software system". [51]

toolkit According to the Gang of Four, a toolkit is a "set of related and reusable classes designed to provide useful, general-purpose functionality." Rather then providing a design, they simply offer functionality. [41]

variant Within a version there can be multiple variants (such as a version of Linux has variants for various architectures such as Intel or PPC) [60].

Versions intended to coexist indefinitely [34].

version The complete state of all parts of a software system at a certain point in time [60].

workspace A library of promotions [34].

Bibliography

[1] *IEEE Standard Glossary of Software Engineering Terminology.*

[2] *Popular Lectures and Addresses.* 1981-94.

[3] Linux kernel version history. `http://ftp.cdut.edu.cn/pub2/linux/kernel/history/index.html` 2000.

[4] Manifesto for agile software development. `http://agilemanifesto.org/`, 2001.

[5] Microsoft responds to the open source memo regarding the open source model and Linux. `http://web.archive.org/web/20010417195837/www.microsoft.com/ntserver/nts/news` 2001.

[6] The cathedral and the bazaar. `http://www.catb.org/~esr/writings/cathedral-bazaar/`, 2003.

[7] Concurrent Versions System - the open standard for version control. `http://www.cvshome.org/`, 2003.

[8] CS 246SE textbook information. `http://www.student.cs.uwaterloo.ca/~cs246se/1035/texts.h` 2003.

[9] Department of computer science - University of Victoria - SENG 221: Software architecture and development methods. `http://www.csc.uvic.ca/courses/fall2003/seng/seng221.html`, 2003.

[10] Eastern Oregon University: Computer science/mulitmedia studies - CS344: Systems analysis. `http://cs.eou.edu/CSMM/fpratter/CS344.html`, 2003.

[11] Eclipse project charter. `http://www.eclipse.org/eclipse/eclipse-charter.html`, 2003.

[12] Eclipse project downloads. `http://downlaod.eclipse.org`, 2003.

[13] eclipse.org - what's new history. `http://www.eclipse.org/whatsnewhistory.html`, 2003.

[14] Extreme programming (XP) - an alternative view. `http://www.softwarereality.com/ExtremeProgramming.jsp`, 2003.

[15] The free software definition. `http://www.gnu.org/philosophy/free-sw.html`, 2003.

[16] Gnu general public license. `http://www.gnu.org/copyleft/gpl.html`, 2003.

[17] jmove. `http://jmove.sourceforge.net/`, 2003.

[18] Merriam-Webster online. `http://www.m-w.com`, 2003.

[19] Mohs scale of mineral harness. `http://en2.wikipedia.org/wiki/Mohs_hardness_scale`, 2003.

[20] Object Mentor - training. `http://www.objectmentor.com/courses/index`, 2003.

[21] Online dictionary - HyperDictionary.com. `http://www.hyperdictionary.com/dictionary/compon` 2003.

[22] Open Source Initiative OSI - FAQ:advocacy. http://www.opensource.org/advocacy/faq.php, 2003.

[23] Open Source Initiative OSI - halloween documents. `http://www.opensource.org/halloween/`, 2003.

[24] Optimalj - package structure analysis tool. `http://javacentral.compuware.com/pasta/`, 2003.

[25] Pde does plug-ins. `http://www.eclipse.org/articles/index.html`, 2003.

[26] Study lauds open-source code quality. `http://msnbc-cnet.com.com/2100-1001-985221.html`, 2003.

[27] Why "free software" is better than "open source". `http://www.gnu.org/philosophy/free-software-for-freedom.html`, 2003.

[28] Phillip G. Armour. The five orders of ignorance. *Communications of the ACM*, 43(10):17–20, 2000.

[29] Kieth H. Bennett and Vaclav Rajlich. A staged model for the software life cylce, 2000.

[30] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 1994.

[31] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a case study: its extracted software architecture. In *Proceedings of the 21st international conference on Software engineering*, pages 555–563. IEEE Computer Society Press, 1999.

[32] Lionel C. Briand, Premkumar T. Devanbu, and Walcelio L. Melo. An investigation into coupling measures for c++. In *International Conference on Software Engineering*, pages 412–421, 1997.

[33] Frederick P. Brooks, Jr. No silver bullet: essence and accidents of software engineering. *Computer*, 20(4):10–19, 1987.

[34] Bernd Bruegge and Allen H. Dutoit. *Object-Oriented Software Engineering.* Prentice Hall, 2000.

[35] John Champaign, Andrew Malton, and Xinyi Dong. Stability and volatility in the linux kernel. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE)*, page 95. IEEE Computer Society, 2003.

[36] Geoffrey M. Clemm. Replacing version-control with job-control. In *Proceedings of the 2nd International Workshop on Software configuration management*, pages 162–169. ACM Press, 1989.

[37] Desmond F. D'Souza and Alan C. Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach.* Addison-Wesley, 1998.

[38] A. Gustavsson FIXME find out what A stands for. Maintaining the evolution of software objects in an integrated enviornment. In *Proceedings of the 2nd International Workshop on Software configuration management*, pages 114–117. ACM Press, 1989.

[39] Fachhochschule Mannheim Frauke. Frauke paetsch.

[40] Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE)*, page 13. IEEE Computer Society, 2003.

[41] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Longman Publishing Co., Inc., 1995.

[42] Richard Gauthier and Stephen Pont. *Designing Systems Programs.* Prentice-Hall, 1970.

[43] Michael Godfrey and Qiang Tu. Growth, evolution, and structural change in open source software. In *Proceedings of the international workshop on Principles of software evolution (IWPSE)*, 2001.

[44] Michael Godfrey and Qiang Tu. Tracking structural evolution using origin analysis. In *Proceedings of the international workshop on Principles of software evolution (IWPSE)*, pages 117–119. ACM Press, 2002.

[45] Michael W. Godfrey and Qiang Tu. Evolution in open source software: A case study. In *International Conference on Software Maintenance (ICSM)*, pages 131–142, 2000.

[46] Michael W. Godfrey and Qiang Tu. Evolution and structural change in open source software. In *Proceedings of the international workshop on Principles of software evolution (IWPSE)*, 2001.

[47] Richard C. Holt, Michael W. Godfrey, and Andrew J. Malton. The build / comprehend pipelines. In *Second ASERC Workshop on Software Architecture*, 2003.

[48] Udo Kelter, Marc Monecke, and Markus Schild. Do we need 'agile' software development tools? In *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*, pages 412–430. Springer, 2003.

[49] Ronald J. Leach. *Introduction to Software Engineering.* CRC Press, 2000.

[50] Manny M. Lehman. *Program Evolution: Processes of Software Change*, chapter 12, pages 247–274. Academic Press, London, UK, 1985.

[51] Spiros Mancoridis and Richard C. Holt. Extending programming environments to support architectural design. In *Proceedings of International Workshop on Computer-Aided Software Engineering*, pages 110–119, 10– 14, 1995.

[52] Robert C. Martin. *Agile Software Development Principles, Patterns & Practice*. Prentice Hall, Upper Saddle River, New Jersey, 2003.

[53] Pete McBreen. *Questioning Extreme Programming*. Addison-Wesley, 2002.

[54] Thomas J. McCabe. A complexity measure. *Trans. Soflw. Eng. SE*, 2(4):308–320, 1976.

[55] Doug M. McIlroy. Mass produced software components. In *Proc. Nato Software Eng. Conf.*, pages 138–155, 1968.

[56] A. Jefferson Offutt, Mary Jean Harrold, and Priyadarshan Kolte. A software metric system for module coupling. *The Journal of Systems and Software*, 20(3):295–308, March 1993.

[57] Peyman Oreizy. Decentralized software evolution. In *Proceedings of the International Conference on the Principles of Software Evolution (IWPSE)*, 1998.

[58] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[59] David L. Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, pages 279–287. IEEE Computer Society Press, 1994.

[60] Roger S. Pressman. *Software Engineering*. McGraw-Hill, third edition, 1997.

[61] Chandrashekar Rajaraman and Michael R. Lyu. Reliability and maintainability related software coupling metrics in c++ programs. In *Proceedings 3rd IEEE Inter-*

national Symposium on Software Reliability Engineering (ISSRE'92), pages 303–311, 1992.

[62] Ian Sommerville. *Software Engineering*. Addison-Wesley, 1989.

[63] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1992. P C++ 97:2 1.Ex.

[64] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The structure and value of modularity in software design. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108. ACM Press, 2001.

[65] Qiang Tu and Michael W. Godfrey. The build-time software architecture view. In *International Conference on Software Maintenance (ICSM)*, pages 398–407, 2001.

[66] Arie van Deursen, Merijn de Jonge, and Tobias Kuipers. Feature-based product line instantiation using source-level packages, 2002.

[67] Yichen Xie and Dawson Engler. Using redundancies to find errors. In *Proceedings of the tenth ACM SIGSOFT symposium on Foundations of software engineering*, pages 51–60. ACM Press, 2002.

[68] Horst Zuse. *A Framework of Software Measurement*. Walter de Gruyter, 1998.