Abstract

Software development is rapidly changing and software systems are increasing in size and expected lifetime. To cope with this, several new languages and development processes have emerged, as has stronger focus on design and software architecture and development with consideration for evolution and future change in requirements.

There is a clear need for improvements, research shows that the portion of development cost used for maintenance is increasing and can be as high as 50 %. We also see many software systems that grow into uncontrollable complexity where large parts of the system cannot be touched because of risks for unforeseeable consequence. Therefore a clearer understanding of the evolution of software is needed in order to prevent decay of the systems structure.

This thesis approaches the field of software evolution through an empirical study on the open source project Portage from the Gentoo Linux project. Data is gathered, ratified and analysed to study the evolutionary trends of the system.

These findings are seen in the context of Lehman's laws on the inevitability of growth and increasement of complexity through the lifetime of software systems. A set of research question and hypotheses are formulated and tested.

Also, experience from using open source software for data mining is presented.

Keywords: Software evolution, software metrics, empirical study, open source.

Preface

This thesis is written as part of my master degree at the Department of Computer and Information Service at the Norwegian University of Science and Technology in Trondheim.

This thesis builds on earlier work where I explored the research field of software evolution and perform a small empirical study on the evolution of the architecture of Portage from the Gentoo Project. For this study the same system is examined more closely using different metrics.

By writing this report, the author hopes to get a better insight of software evolution and empirical study.

I would like to thank my supervisors, Reidar Conradi and Thomas Østerlie for valuable feedback and support and the developers of Portage for the work that have provided me with data for my analysis.

_____

Andreas Tørå Hagli
16. June, 2005
Trondheim, Norway

Contents

Chapter 1

Introduction

Most computer systems experience some form of software evolution over time. Lehman's first law of software evolution [Lehm74] states that software systems that address problems in the real world must be continually adapted and changed if it is to offer satisfactory service. Functionality is added and quality attributes improved. Efforts are made to maintain control over the system, and one of them is to plan an overall architecture of the system and have its basic structure intact. But, the architecture also has to evolve to accommodate for unforeseeable events and to adapt to changes in its purpose and focus.

In the software industry today, maintenance is becoming a larger and larger part of the cost of software development. Evans says in [Evan01] that "evolving and maintaining computer systems is expensive. This cost can be anywhere from 50 % of total programming effort [Lien80] to 75 % of total available effort [McKe84]. In addition, the proportion of effort devoted to maintenance has been increasing: from 35-40 % in the 1970s, through 40-60 % in the '80s up to 70-80 % in the 1990s [Pfle91]". In order prevent such high maintenance cost and rather focus on business issues, we must therefore explore the nature of these changes and learn how to foresee and handle them better.

There are various definitions of what activities are included in software evolution. The definition adopted in this paper is all activities and phenomena associated with adding functional and improving non-functional requirements that cause physical and conceptual changes to the structure of the software system. It is important to note that although evolution is often a negative side-effect of a wanted change, it can also be, and is often, a necessity for the system.

This thesis describes the research field of software evolution perform an empirical study related to earlier findings on the inevitability of growth and increasement in complexity [Bela76]. A set of research question and hypotheses are formulated and to see whether the same trends can be found in a system developed almost thirty years later.

The thesis uses an open source project call Portage from the Gentoo Project and tries to provide experience from using an open source project for data material. The system uses the same tools commonly used in most open source projects.

The remainder of this thesis is organized as follows. Chapter 2 describes the background of software evolution as a research field and research done by the author prior to this thesis. Chapter 3 describes the context for the data used in an empirical study to answer the research questions and test the hypotheses in chapter 4. Chapter 4 formulates a number of research questions and hypotheses about software evolution

and describes the metrics used. Chapter 5 presents the results from the empirical study. Chapter 6 discusses there results. Chapter 7 concludes the report and gives suggestion for further work.

Chapter 2

Background

2.1 Software evolution

Although discussion of software evolution has existed for a long time, it's only recently that it has become a well known phenomenon in mainstream research. As Lehman states in [Lehm01b] "the software evolution phenomenon was first identified as such in early 70s. It is reflected in an intrinsic need for continuing maintenance and development of software use to address an application or solve problem in real world domains. Until recently, however, it did not arouse general interest".

The definition of software evolution is debated, different authors have different ideas on what should be considered as evolution. Some consider evolution to be something that happens during the whole lifetime of the software system while some limits it only to a certain development phase [Rajl01]. Some consider software evolution to be the phenomena behind all changes in the software system, while others consider it to involve only abstract changes and thus not include things like correction of earlier mistakes and porting to a new platform. Also, some limit software evolution to the software system itself, while others includes the development process and organizational structure.

Lehman et. al. says in [Lehm01a] that "the more common approach sees the most important evolution issues as those concerning the methods and means whereby a software system may be implemented from ab initio conception to operational realisation. The focus of this approach is the how of software evolution". A less frequent approach "is concerned with the what and the why of evolution. It addresses the issues of the nature of the evolution phenomenon, its drivers and impact." Yet another view on software evolution is used by some authors, "they regard it as being limited to software change and implicitly exclude, for example, defect fixing, functional extension, restructuring." Still others consider it "a stage in the operational lifetime of a software system, intermediate between initial implementation and servicing".

A central issue in defining software evolution is whether to differ between software evolution and software maintenance. Software maintenance includes the minimum set of activities required in order to keep the software system running as it is with the functionality it currently holds. This includes fixing bugs and security updates and making sure the software system works on new hardware and platforms. If not maintained, the software will be considered outdated and possibly useless after some time. Software maintenance is important and numerous scientific studies of large-scale software systems have shown that the bulk of the total software-development cost is devoted to software maintenance [Mens01]. Even when software maintenance is not considered part of the software evolution, it is a tightly related phenomenon. Another related term is software

decay, which is defined as decreasement in the "quality" of the system from a developers perspective, this includes lack of documentation and understanding of the system and low correlation between the actual and the ideal architecture.

Software systems are developed in different phases, e.g. making requirement specification, initial design, writing code, testing and installing. But execution of these is rarely sequential, errors are uncovered and underlying assumptions are changed after installation of the system. The process is one of successive transformation. "It is driven by human creative and analytic power as influenced and modified by developing insight and understanding, but feedback from later steps that leads to iteration over earlier steps, together with changes in the external world that must be reflected in the system, also play a role" [Lehm01a]. Different areas on the development are in constant interaction, e.g. the process affects the physical software system which again affects the releases and so fourth.

Lehman describes five areas of software related evolution [Lehm01a], each of them interact with, impacts and affect the others. "If software evolution is to be mastered, they must be understood and mastered individually and collectively. They must be planned, driven and controlled." The five areas are:

1. Implementation of a software system from an initial functional concept to the final, released, installed version. Often the relative benefits of alternatives can't be established beforehand, and need realistic trials. This evolution is thus driven by feedback mechanisms and evaluation of changes made to the system.

2. At the next level up, there is a sequence of versions, releases and upgrades of a software system. Driven by a release process where changes are made to remove defects and implement improvements and extensions. This is usually referred to as "maintenance".

3. Applications, or activities, to support the development of software systems that address problems in the real world. There is an unending process to meet new functionality, procedures, need, opportunities and so forth when dealing with a loose user community.

4. The process of software development is the aggregate of all activities the implement one or other of the above levels of evolution. An estimated 60 % to 80 % of lifetime expenditure on a software system in incurred after first release [Pigo96]. It is therefore important to make improvements that produce gains in quality, cost, and reduced development time.

5. Modeling, using a variety of approaches is an essential tool for study, control and improvement of the process. The models facilitate reasoning about it, to explore alternatives or assess the impact of change, for example. The process evolves. So must the model of it.

This rapport, a wide definition is used for software evolution. It includes the changes to a software system from the design to the end. It involves changes in the software system, in business requirements and the process of development and organizational structure. Of the five areas defined above, this rapport has special emphasis on the first and second level; the physical software system and its releases.

2.2 Why changes occur

Ideally, at the beginning of software development, when creating an architecture, all concerns for future changes in the systems requirements are considered and accounted for. However, this is rarely the case. The reason for this is that future requirements are not always known, unpredicted new requirement, tight deadlines and changes in the goals and purpose of the software system is common. Software evolution is considered unavoidable and often important, like when the business model is extended and a new marked is met. Although software evolution is often considered unwanted, it should often be considered natural and a necessity.

We distinguish between changes intended to add new functionality and changes intended to improve quality attributes. New functionality can be to add supporting payment with credit card, adding print support, a new graph that provide some information or similar. Improved quality can be to improve the overall performance, increase the uptime of a system, increase the security or similar.

But changes can't merely be tracked back to a specific action; often it is important to consider the environment. Obviously the development process and practices plays an important role in preventing or encouraging changes to the software system. Andy Hunt et al. [Venn03] note the importance of fixing small unknown errors. His point is that if a development team does not pay careful attention to sustaining a stable high quality, they will "technical debt", meaning that a known technical problem is not fixed. Postpone fixing these errors often leads to more technical debt and often abdication of responsibility. The software therefore ends up in a spiral of increased technical debt and decay.

The background for Hunts opinions is a theory called the Broken Window Theory based on an experiment done to see what cause neighborhoods with similar demography to evolve in different direction with respect to crime. Hunt explains the study like this: "The researchers did a test. They took a nice car, like a Jaguar, and parked it in the South Bronx in New York. They retreated back to a duck blind, and watched to see what would happen. They left the car parked there for something like four days, and nothing happened. It was not touched. So they went up and broke a little window on the side, and went back to the blind. In something like four hours, the car was turned upside down, torched, and stripped–the whole works."

From this a theory was developed that says that one broken windows lead to more broken windows and further worse criminal act in and exponential fashion.

2.3 Laws of Program Evolution Dynamics

In [Bela76], Belady and Lehman describes finding from an analysis of a large software system. They discovered an upward trend in size, complexity and cost of maintenance. These and more detailed observations encouraged the search for models that represented laws in software evolution.

Three laws where formulated:

1. **Law of continuing change:** A system that is used undergoes continuing changes until it is judged more cost effective to freeze and recreate it.

2. **Law of increasing entropy:** The entropy of a system (its unstructuredness) increases with time, unless specific work is executed to maintain or reduce it.

3. **Law of statistically smooth growth:** Growth trend measures of global system attributes may appear to be stochastic locally in time and space, but, statistically, they are cyclically self-regulating, with well-defined long-range trends.

These laws have been used in many papers and formulated in different ways usually reffered to Lehman's laws of software evolution. In this thesis these three laws will be described as Lehman's first, second and third law respectively.

These findings where presented almost thirty year ago, but are still subject to discussion. Especially the second law is subject to a lot of controversy on the subject of software evolution. This thesis will look at the same issues to test it on the Portage system from the Gentoo Project and see whether the laws are still are relevant.

Especially the second law will be considered. While Lehman's first law deals with the inevitability of change, this deals with the consequence of that. This comes from the addition of functionality like more configurations, handling of special cases, adding user styles and supporting more languages. This done in the original design of the system degrades it and results in reduction of manageability [Lehm80].

Stated by Lehman, "The accumulation of gradual degradation ultimately leads to the point where the system can no longer be cost-effectively maintained and enhanced." Not only functional maintenance becomes necessary, but in addition structural maintenance is needed. Clean-up and re-engineering efforts are needed.

The rationale given by Lehman is that "if one tries to do the necessary changes to a system in a cost-effective way, this is what happens. If there are multiple objectives, one cannot fulfill all of them optimally". The design can't change for every change. Also over time, original design ideas becomes forgotten.

2.4 Goal-Question-Metric

The background for the research questions and hypotheses is the ideas from the Goal-Question-Metric (GQM) paradigm [Basi94]. GQM is a method used to define measurement on the software project, process, and product used to define and evaluate a project. It defines a measurement model on three levels:

Conceptual level (goal): A goal is defined, for a variety of reasons, with a model of quality, a point of view, and for a particular environment.

Operational level (question): A set of questions is used to describe the object studied in a manner that relates to achieving a specific goal.

Quantitative level (metric): A set of metrics is associated with every question in order to answer it in a measurable way.

GQM states that data collection should proceed in a top-down rather than a bottom-up fashion. However, [Moha04] gives three reasons for why bottom-up studies are useful. Firstly, most data is collected in repositories with the goal of providing a service, not data for a GQM paradigm. Secondly, companies that start to use GQM might want to use older data and relate this data to goals (reverse GQM). Thirdly, although a company might measure according to defined goal, the measuring practice itself needs improvement from bottom-up studies.
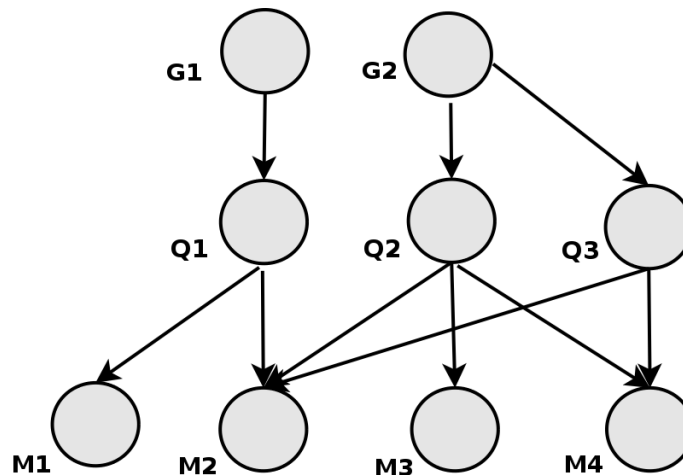
Figure 2.1: A Goal-Question-Metric hierarchy

2.5 Software metric

A software metric is a measure of some attribute of software or its specification. Example of software metrics are lines of source code and bugs per line of source code. They have long been studied as a way to assess the quality of large software systems [Fent97]. Software metrics are used when a certain property of a software system is of interest, a set of one or more attributes are then considered to represent that property and a metric used to extract values for the attributes.

Description of software metric

Measurement

Attribute → Measure

Metric

Example of software metric

Add values

removed lines (3)
added lines (8)   → 11

Metric for
change effort
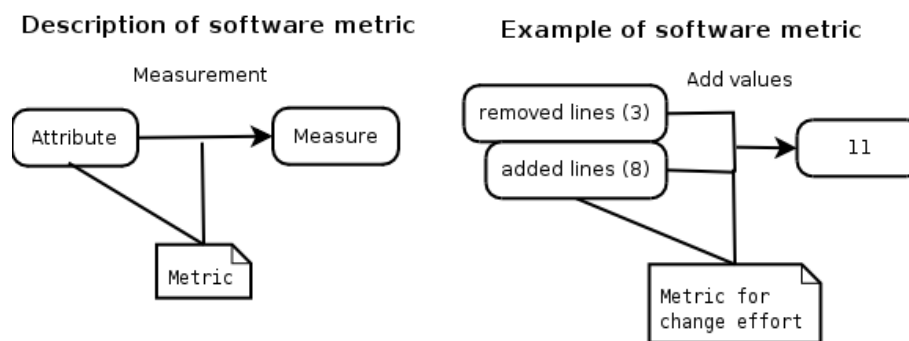
Figure 2.2: Software metric

Figure 2.2 describes the term metric. When an attribute is measured, you get a measure/value. In this process, a metric is defined as the attribute and the applied way of performing the measurement. The metrics describes what the attribute is suppose to be measured as, like a scale or a set of values, and how the measuring is performed, e.g. manually or automatically. Metrics can be used in a number of ways to support software evolution and software engineering in general. It is simple, precise, general and scalable and provides quantitative results and the results can be duplicated and compared.

Mens says in [Mens01] that "improving software quality, performance and productivity is a key objective for any organisation that develops software. Quantitative measurements and software metrics in particular can help with this, since they provide a formal means to estimate software quality and complexity".

When using metrics to understand the software system and evolution better it is

important to know what you are looking for. Mens says in [Mens01] that "initial experiments have indicated that metrics can detect different types of evolution, such as restructuring and extension. Nevertheless, it remains an open question which types of software evolution can be identified by which metrics. A related question is whether it is possible to reconstruct the motivation behind an evolution step (e.g., why was a certain change made between two successive releases)?" Therefore although metrics might be easy to use, the issue of actually making sense of the output can be harder.

Gall et al. separate between the use of software metrics before the evolution has occurred (i.e., predictive), and the after the evolution had occurred (i.e., retrospective) [Gall98].

2.6 Researching on open source project

Research has been done on open source before, e.g. [Mont04]. In [Mont04] Monteiro *et al.* uses Gentoo Linux to describe the process of decition making in Open Source based on qualitative study of IRC-log and mailing lists.

The reason for choosing to use open source and the Gentoo Project in this thesis was because it was used in the authors previous work and it was therefore found valuable to draw from this experience and take a closer look it.

Approaching an open source project for analysis is in several ways different from approaching an industrial software project. Important characteristics are openness around development and related information and homogeny in technology used in different open source projects.

Open source projects, at least those involving geographically spread developers, are in need of keeping development related information publicly available. This is useful for attracting developers and coordinate development. For a researcher, this means that information, although possibly incomplete, is available and there is no need for clearance and help from project participants.

Another tendency in open source project is that they tend to use the same tools and often similar practices for development. This includes the use of CVS and Bugzilla to manage the code and IRC, mailing list and blogs for discussions.

Shortcoming may include problems of generalization and lack of stable development and release practice caused by a lot of the work being voluntary. Voluntary involvement also makes the motivation different from most industrial projects and consequently also development focus.

This thesis will use an open source project for data analysis and testing of research questions and hypothesis and will try to document experience and challenges related to the characteristics described above.

2.7 Prior research by the author

In earlier work by the author, Portage from the Gentoo projects (same source as in this thesis) was used to answer and test the following research questions and hypotheses:

RQ1: What evolution trend happens in the lifetime of a software system?

RQ2: What is the share of different change types?

H1: Preventive changes modify more files than corrective and adaptive changes. **[Accepted]**

H2: In absence of restructuring focus, the share of preventive changes increases during the lifecycle of software systems. **[Accepted]**

H3: The effort of changes is stable independently of file size. **[Not accepted]**

Too little conclusive results where found on RQ1 and so this thesis will make an attempt at answering related questions.

Also, analysis was done on data describing changes on files for different months. This thesis will try to analyze using data describing changes on modules, classes and function for different logical interval instead since it can be considered a better measure.

For details on data used and results, see [Hagl04].

Chapter 3

Research context

A lot of data is collected during the development of a software system. In order for developers to cooperate smoothly, information is collected to prevent them from stepping on each others toes and to communicate and monitor activities.

Portage, the project used in this study contains about 16,000 lines of Python code in about 50 files. In addition there are several shell scripts that wraps various system tools. Data for about three year of development between October 2001 and October 2004 have been used.

The development of Portage is similar to the development found in other open source projects and the tools used and described later in this chapter are de facto standards for open source development.

Portage is a central piece of the Gentoo Linux system and has therefore been subject to a lot of changes to accommodate the constant need for new functionality. Therefore not only has the size continually increased, but due to constant change in requirements as well as quick updates, no strict attention has been made on structural maintenance and so the design is more or less the same as is was at the initial stage with a tenth of the size in code lines. Portage is therefore particular suited for testing Lehman's second law of increased complexity is absence of structural maintenance (described in section 2.3).

3.1   Study objects

Portage, the package management system for the Linux distribution Gentoo Linux is used. Characteristic for this system is the target users are developer and it should therefore have more and better feedback from its users. It is based on voluntary work and consequently it is cantered around a few stable core developer assisted by a larger developer base. Portage performs critical tasks for the operating system (Gentoo Linux) and therefore it has a high demand for stability and quick updates in case of faults.

The development history is special in that public development has lasted three year without large restructuring of the code (although a major restructuring is under work separately from development studied here). Also a simple fundamental architecture of the process for the system exists and has been stable since development stated, but there has never been any architecture for the internal part of the system. Neither has there been any formal architecture through documentation, diagram or similar.

3.1.1 Gentoo Linux

Gentoo Linux is a Linux distribution based on voluntary effort. It is a collection of several open source programs packed together to form a fully working operating system. It is unique in that it does not distribute compiled binary-version of the system components, but instead makes each system compile and build the programs itself optimized the hardware and with user-specified parameters.

The vast majority of the development is done by third-party developers while about 200 Gentoo developers makes sure all the programs gets integrated with the system. A large part of this work is done by writing *ebuilds,* which is a script with instruction on how to find the newest version of the software, build it and finally merge (integrate) it with the system. A major part of the development focus on making new and improving the ebuilds.

An important characteristic of the Gentoo project is that it is focused on being highly developer friendly to attract highly skilled developers as users to help improving the system. The system is therefore driven by the open source philosophy of "scratching where it itches", meaning that developers fix and improve the part where they themselves see a need as a user.

Combined with having highly skilled developers, this creates a strong community for improving the system.

3.1.2 Portage

The Gentoo web page states that "Portage is the heart of Gentoo Linux, and performs many key functions." [Gent]. It is where most of the in-house development in Gentoo takes place, excluding ebuilds.

It is the program that takes care of distributing, building and keeping track of all the "packages" of the programs that makes up the Gentoo Linux operating system.

Portage contains about 16kloc of Python code and several utilities written is shell script and consists of about 5-10 main developers. It has a recorded public development history stating August 2001.

The process of merging (integrating) a new program using Portage can be described as follows:

1. Search PortDB for the newest ebuild or a given version for a specified package. An ebuild is a script with guidelines for emerge on how to perform the installation.

2. Parse ebuild to determine dependencies; that is required packages needed before installing this particular ebuils.

3. Download source code need for the package from a remote server over HTTP and if necessary download and apply associated patched.

4. Build the source code in temporary location with the specified values from `make.conf`.

5. Perform post-build actions.

6. Merge the newly build package with the existing system.

7. Create new entry in VarDB with the ebuild, parameters from `make.conf` and list of files added to the system.
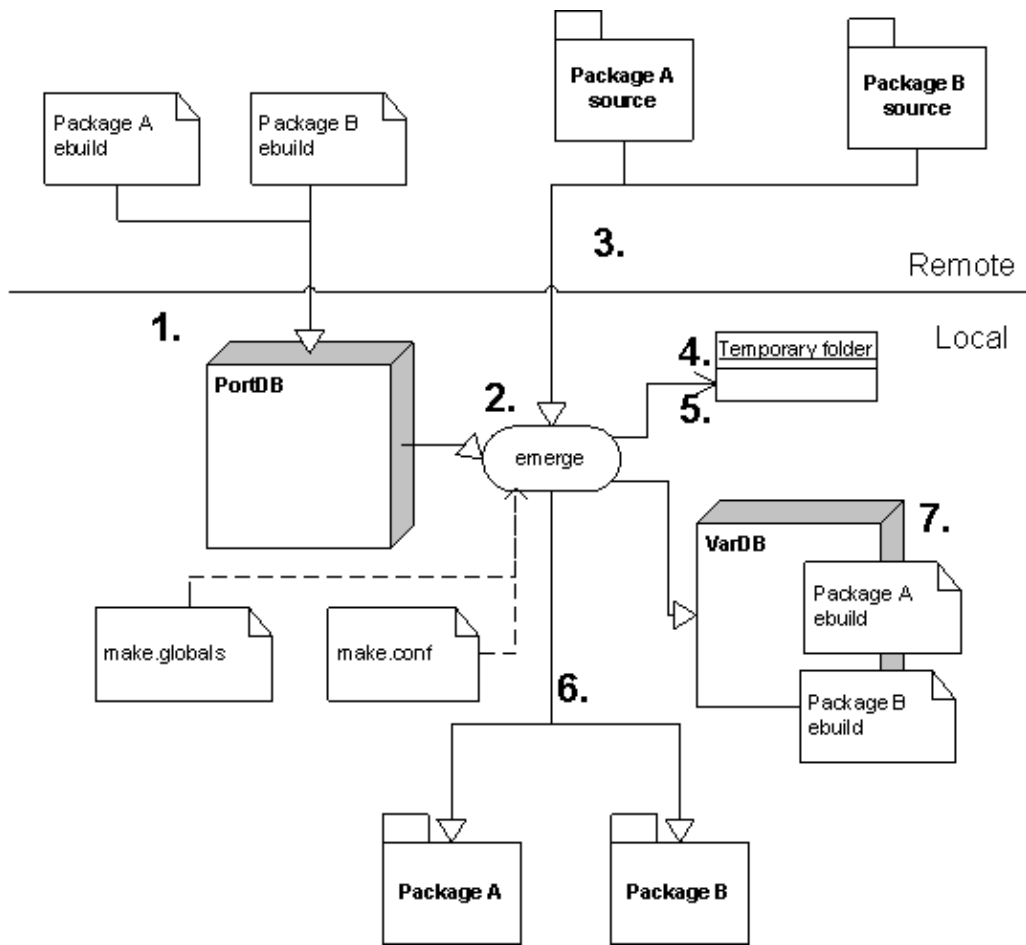
Figure 3.1: Architecture of Portage and the process of installing a package

Ebuilds are central in the Gentoo project. They are ordered in a three level hierarchy of category, package and version. A number of categories are defined, each containing several packages. For each package, there are also specified different versions. Usually the newest version is preferred, but occasionally an older version of a package is needed, so different versions of a package are available to the system administrator.

The PortDB is a database of ebuilds for packages available to the user, either installed or with the potential for been installed or upgraded. In these scripts are instructions for how to build the content of the package, usually a program. The PortDB is updated via a remote database manually be the user to make sure the system know of updates and security fixes.

When a system administrator wants to upgrade a part of the system or install a new package, he or she runs the emerge program that checks PortDB to find and run the necessary ebuild(s). It downloads associated source code for that package from a remote server.

Next emerge compiles the source code for the package in a temporary location based on the configurations in the files make.globals and make.conf and perform any specified post-build task, like adding users, to the system.

Further emerge merges the package with the system by copying the files from the temporary location to the correct location. Finally emerge copy the script and a list of the new files just copied for the package to VarDB to keep track of the state of the system.

3.2 Releases

One improvement for analysis in this thesis over previous work [Hagl04] done on Gentoo Linux by the author is to use compare logical instead of monthly code samples. This makes it safer to balance out periods of bugs fixing and periods of development of new functionality. It also limits the effects of special events skewing the data, e.g. in the development of Portage a module was temporary added to work as backup, thus causing a temporary increase followed by a similar decrease in size.

The possibility of using software releases as a way to compare logical changes is highly dependent on the project. Industrial development projects often have a timeline for when the software should be released for the customer, both for shrink-wrapped or custom-made software. This timeline is affected by technical, business and organizational concerns. For open source projects, business concerns are usually much less important which means more focus on the others. Also, open source projects are often in strong growth in size of the system and amount of developers involved at the beginning and organizational issues are often an adapting process, therefore open source projects can be said to be more likely to change development practices over time and more dependent in technical issues.

Releases of open source are as well suited for logical analysis as releases of other projects, if not better, but only if the development is stable. At a pre-stable level, open source projects are mainly focused on development and makes releases with the aim of using the users as tester for the program. Thus, a new release does not occur when the system is stable enough for end users, but when it is stable enough for testers and early adopters. Open source software might be better at a stable state because it is not as affected of a companies unique business concerns that makes it hard to generalize the findings.

The development of Portage, used for analysis in this thesis, is an example where initial development had frequent releases and where releases was slowed down later. Also

the routine for releases and release number changed during development. This makes it harder to establish a clear understanding of logical releases to be used for comparison. A more careful look at development history is needed, an interpretation of logical releases is shown in figure 3.2. At a more mature stage however, Portage has begun to follow a common release cycle for open source projects described in figure 3.3. Even though this is the formal practices it should be noted that most of the development in Portage is still done in small increments.

The timeline in figure 3.2 shows the overview of the releases recorded in the projects ChangeLog that are considered to be the most important ones. They where considered important mostly based on their release number and the time between them. There several releases between 2.0 and 2.0.47 (2.0.1, 2.0.2, ... ,2.0.46), but they where so frequent that they should not be considered important because they only represents small incremental releases.

Figure 3.2: Timeline of important Portage releases. Red ones not used in this thesis.
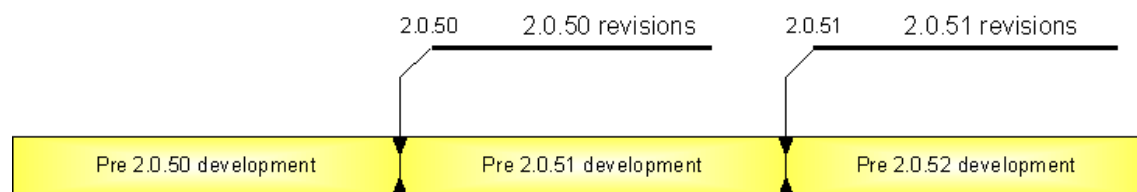

Figure 3.3: Common open source release cycle, currently used by Portage

Open source projects release cycle and release routine is very important if it is necessary to look at difference in logical releases. Several large and mature projects follow a clear release cycle and often has time based releases, e.g. every 6 month.

For this thesis, source for the releases in figure 3.2 was attempted extracted, however only 1.8.0, 2.0.47, 2.0.48, 2.0.49, 2.0.50 and 2.0.51 where found. Although not the complete set of important releases, these do represent regular releases for at three-year time period.

3.3   Data sources in open source projects

The Portage system uses the same development technology as the majority of open source projects. The figure below describes this system. There are two main parts, Bugzilla and CVS. Bugzilla keeps track of the state of the system (like reported faults) while CVS keeps track of changes to the code base.

As shown in figure 3.4 a change request is reported by a user to Bugzilla, a web-based system that keeps track of the state of bugs. If the bug report is accepted by the responsible Bugzilla administrator, it is published as an open task ready to be performed by an unassigned or assigned developer. A developer, either the developer responsible for that part of the system or a voluntary developer, makes the necessary modification to a
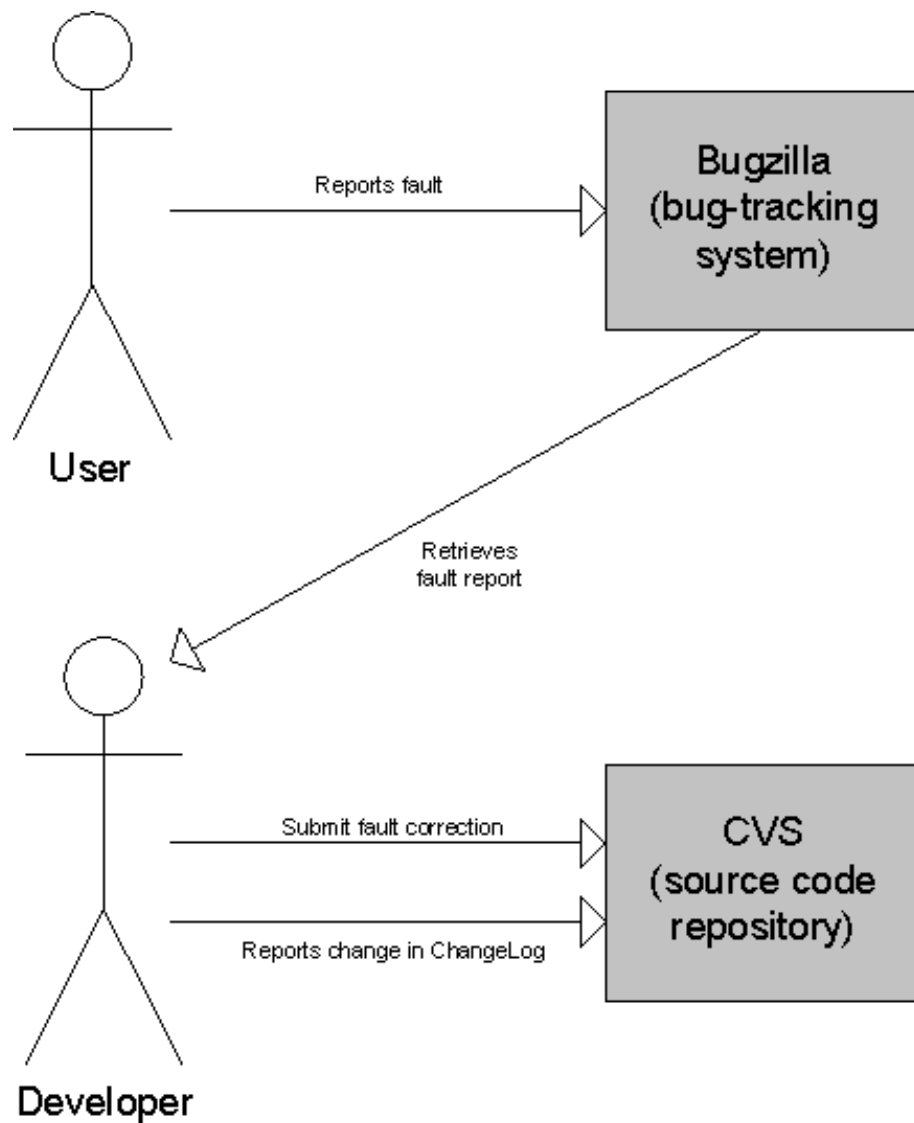
Figure 3.4: The process of a fault being reported and fixed

local version of the code recently updates from CVS. When the developer is done, the code with modification is submitted to CVS, a code repository that keeps track of changes to the code. For any modification, the developer is asked to make a description about the change. Although not forced by the system, the developer also reports the changes made with a description in a file called ChangeLog at the root of the project.

Although Bugzilla is created for the purpose described above, it is also commonly used for tracking request for new features the same way as requests for bug fixing. The process for this is the same as described above.

This gives two sources for data mining of the Portage system, the ChangeLog-file which is a historical log with comments on all changes made to the system and the CVS-logs which logs all changes made to a single file in a more detailed form than the ChangeLog. The Bugzilla system also provides valuable data for empirical studies, but is not used here.

3.3.1 ChangeLog

A changelog is a log of recorded changes made to a project. Commonly used in open source projects with the intention of keeping an overview of changes so that developers can follow what has been done by others.

In Portage, and most open source projects, it comes in the form of a single text file. Although there are no formal standard for the layout of a ChangeLog-file, a change done by a developer to a file in a project is usually supplemented, possibly automatically, by adding a paragraph to the top of the file with the date of the change, the person that made the change, the files effected and a description of the change. When the maintainer makes a release, he or she adds a new line at the top of the file saying that a new release has been made public. Therefore it is also possible to track the version a given change was included in.

Following is a sample text from the ChangeLog-file in the Portage project:

```
* portage-2.0.51 (20 Oct 2004):  Everyone loves stable!

  20 Oct 2004; Jason Stubbs <jstubbs@gentoo.org> repoman:  Added
  check for digest entries that aren't used within the
  corresponding ebuild's SRC_URI.

  20 Oct 2004; Jason Stubbs <jstubbs@gentoo.org> emerge:  Added
  support for EMERGE_WARNING_DELAY defaulting it to 10.  Changed
  all the hardcoded delays to use it.  Needed for the catalyst
  guys as it includes a number of unmerges of system packages.

  19 Oct 2004; Nicholas Jones <carpaski@gentoo.org> portage.5:
  patch included to fix a few typos.
```

This data is provided in a structured way in the ChangeLog-file, therefore, it can be extracted automatically. Date of change, developer, files affected and description can easily be extracted. The description can further be used to classify what kind of change was made and categorize them. Mockus outline a good description on how to label these characteristics automatically [Mock00].

The suitability of data from ChangeLog-files have been questioned, certain research have shown that the correctness of it varies from 5 to 75 %.

3.3.2 CVS

CVS is a commonly used and simple version control system. It tracks the history of files by logging differences (added and removed lines) between the files before and after a change with comments from the developer. It stores information about what lines are added where and what lines are removed, the time at which the change was made, whom made it and comments. It stores information of each file separately and thus does not contain information on which files where changed at the same time as part of a single submitted change to the system. An important strength of using CVS and similar systems for data mining is that it provides consistent data over the duration of development regardless of changes in the development process or active developers.

Data from the CVS can be presented through a web-based interface listing a summary of each change. Following is a sample of a change summary displayed in a web-based CVS- interface:

> Revision **1.530** - (download), view (text) (markup) (annotate) - [select for diffs]
> Mon Oct 25 11:20:46 2004 UTC (4 days ago) by jstubbs
> Branch: **MAIN**
> Changes since **1.529: +16 -8 lines**
> Diff to previous 1.529
> `Converted config.pkeywordsdict from {atom:[keyword]} to`
> `{cp:{atom:[keyword]}} to prevent a lot of unnecessary calculation.`

On the first line show the revision number for a file. This is generated automatically by CVS and has no relevance to any release number. The second line shows the date and time and the username of the developer that committed the change. The third line shows the branch used in case it is necessary to keep track of two separate version of the same file at the same time. The fourth line shows from what revision the changes where committed and the number of lines added and the number of lines removed. At the end of the summary a textual description of the change is shown.

Although CVS does a good job of managing the history of single files, it merely provide data on removing and adding files and removing and adding lines. If a file in moved or renamed it is simply considered removed and a new file is considered added. Also if code is moved around inside a single file or from one file to another, this is also registered as deletion and addition. Therefore, in contrast to some other code repositories, there are no simple ways to find data on moving of code and splitting or merging of files. This is a clear weakness for using CVS in data mining. Another limitation is the lack of data on the project as whole due to the fact that history scope is limited to single files.

3.3.3 Bugzilla

Bugzilla is a bug managing system where users and developers reports bugs which are than publicly viewable. Several properties can be set for the bug, like version number, severity and related component. Also the maintainer can change the status of the bugs, close it or post comments.

Bugzilla provides a web-based user interface for handling and viewing bugs for the system. This interface provides several ways for presenting the data to the user. However, the author found no trivial way to extract the history of bugs for data analysis. Instead a script had to be used in order to make several queries automatically and put together data collected.

Another limitation with using Bugzilla for data mining is that there are no direct connection between Bugzilla and code repositories like CVS. Thus, it is hard to connect data on bugs to data on change. The only possibility the author found is the mentioning of a bug number in the ChangeLog-file the a bug was resolved. However, there are several uncertainties on this connection.

3.3.4 IRC, mailing lists, blogs

The social process is important for all non-trivial software process done by a group of developers. In industrial projects, this is often seen in the form of the well known "water cooler talk" or brainstorming on white boards. This is obviously suited for coworkers working physically collocated. In contrast open source projects have a tradition of working through mailing lists and IRC. Recently the use of blog for discussion has become popular, especially for large projects.

There forms of discussion is public and often archived and makes it much easier to understand the human side of software development than e.g. using an observer at a workplace and can provide valuable qualitative data.

Example of research on the open source project using these sources can be found [Mont04] which looks at the development process behind Gentoo Linux. Such research is not done in this thesis.

3.3.5 Statistics

Since open source projects often use similar tools and the development is public, is rather easy to get data from any give project and to compare them. A good example is the development repository called SourceForge, which hosts about 100,000 projects providing them with services like code repository, mailing lists, web space and so on. It automatically provides statistics over activity and downloads for each project. SourceForge has been used in research to provide data on open source projects like how many projects survives the first month and how many projects has only one or two developers.

3.4 Supporting tools

A number of projects where found when looking for tools to support metrics. They where not all relevant for this thesis because only tools supporting Python was used. Here are some of the best suited tools for different platform and purpose presented.

3.4.1 Maven

Maven [Mave] is "a software project management and comprehension tool" developed by the Apache Foundation. It provides a central way to handle builds, documentation and provide information of the code base. Although it is not directly a tool for metrics, it has a large set of components and plug-ins that provide several metrics of the code with little effort. These metrics are intended to be used for seeing weaknesses in the code base, but it should be possible to use this for analysis. Maven only supports code written in Java.

3.4.2 XRadar

XRadar [Xrad] is a code reporting tool that produces "HTML/SVG reports of the systems current state and the development over time - all presented in sexy tables and graphs". An example is shown is figure 3.5.
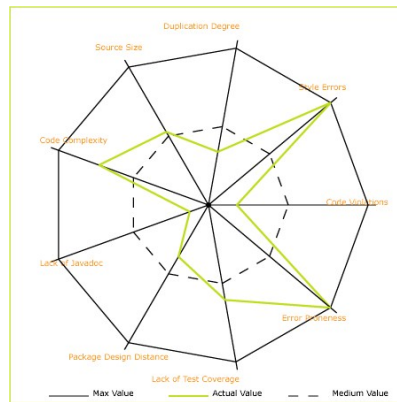

Figure 3.5: Output of XRadar

Originally designed to support the need of large reengineering initiatives at Telenor it gives "measurements on standard software metrics such as package metrics and dependencies, code size and complexity, code duplications, coding violations and code-style violations." Data from unit test metrics and code coverage are also integrated.

It only supports Java-based systems but plans to support other languages.

3.4.3 Pythius

Pythius [Pyth] is a simple open source tool that measures some basic metrics on Python source code.

Pythius provides data for the size (lines of code) for classes and not just files. It also distinguish between commented, blank and ordinary code lines. This made it possible to look for logical changes (changes in class size) and not just physical changes (changes in file size).

Pythius was modified and used in this thesis to store the results in a MySQL-database instead of printing it on the screen. A sample of the source code was analyzed for every month of development since public development stated July 2001.

Most of the data was later replaced by data from Pythonmetric [Pyme] (released 5. May 2005) because it provided more advance data. However data on comment and non-comment lines was extracted by Pythius.

3.4.4 Pythonmetric

Pythonmetric is the result of a student project at NTNU, it "will calculate and output metrics on code written in Python. Reports can be generated in text- or XML-files" [Pyme]. Metrics used includes cyclomatic complexity (see section 4.3.1) and lack of cohesion (see section 4.3.2) in addition to basic metric like lines of code and number of classes. It can be extended with new metrics through a plug-in system. The results can be generated for any number of files inside a folder.

Pythonmetric was used to generate most of the data used in this thesis.

3.4.5 Spreadsheet application

Spreadsheet applications like OpenOffice and Excel are useful for handling the data produced by other tools and make figures for graphically representing the results.

3.4.6 Database

Spreadsheet applications are limited for making selective use of data. It can summaries data and present them, but not suited for sort and restructure them. A database can store data, extract a selection based on a criterion and combine more easily using SQL.

For extra large data store for a general purpose, it might be useful to use a data warehouse to increase speed and get a better overview.

3.4.7 Other

There are several general purpose tools for "washing" data and testing metrics usually through the use of regular expressions and structure analysis. They include Biff/Yacc, lex and perl.

Chapter 4

Research questions and hypotheses

The goal of this thesis is to look at what factors can explain evolution in software architecture and whether it is possible to foresee that evolution will occur.

The research questions and hypotheses described in this chapter is constructed for usage in the context of an open source project called Portage described in chapter 3.

This chapter formulates research questions and hypotheses in order to look further at issues relevant to this report in the context of Portage.

4.1 Software metrics for software evolution

Evolution proneness A term suggested by Mens in [Mens01] is evolution-prone parts, which are parts of the software that are likely to be evolved. The reason is not necessarily because of poor structure or quality, but because the software requirements can change or disappear quickly. A way to detect this in a quantitative manner is to investigate the release history of the software on earlier releases and identify which parts of the software has been most frequently changed. To keep track of the changes, most large software projects use some form of version control system that can be used for analyzing changes between releases.

Metrics can be used to give a measure of the amount of change made to a part of the system at a specified granularity, like module, file, class and methods. When deciding granularity, it is important to know the limitation of the tools being used, measuring changes on a class is probably more demanding that measuring changes on a file. Two factors are important for this measuring. Firstly the size of the parts analyzed matters because the relative amount of change is important. Also, the time at which the change was made is important, recent changes are more important for the current status of the system.

Different visualization techniques might help dealing with scalability issues to adapt to the cognitive limitations of humans. As Mens writes: "to cope with the scalability issue, typical examples of this approach visualise the measurements. For example, Ball and Eick notate code views with colours showing code age [Ball96], and Jazayeri et al. use a three-dimensional visual representation for examining a systems software release history [Jaza99]. Lanza combines software visualisation and software metrics as a simple and effective way to recover the evolution of object-oriented software systems" [Lanz01].

Evolution sensitivity Another useful term used by Mens is evolution-sensitive parts, which are parts of the software that can cause problems upon evolution or where the estimated effort of managing the impact of changes is very high. This typically happens

where the important design goals of loose coupling and high cohesion is not met. When different parts of the software are tightly interwoven, changes to one part might have a high impact on other related parts.

A type of metrics are used to detect evolution-sensitive parts, is coupling metrics. There are some suggestions for coupling metrics, including coupling between object classes (CBO) [Chid94], coupling factor [Brit95] and response set for a class (PFC) [Chid94].

Another possibility is to use cohesion metrics. However, Kabaili et al. investigated whether they could used as changeability indicators, and concluded that this is not the case, at least not with the cohesion metrics present at that time [Kaba01].

More research remains necessary to find out whether other metrics than cohesion and coupling can be used to detect evolution-sensitive parts of the software.

Retrospective analysis of software By comparing two releases of the same software system, Mens states that it is possible to extract information about how the software system has evolved and see what kind of evolution has occurred.

As an example, he mentions an retrospective empirical study by Gall et al. where coupling metrics where used on multiple releases of a large telecommunication switching system [Gall98]. The results could successfully be used to more accurately predict future expected maintenance activities.

Another study performed by Demeyer et al. where various size and inheritance metrics where used on three releases of a medium-sized object-oriented framework [Deme99]. "From the framework documentation one can deduce that the transition from the first release (1.0) to the second release (2.0) was mostly restructuring, while the transition from the second (2.0) to the third release (2.5) was mainly extension. This restructuring and extension was confirmed by the measurements. During the restructuring phase, a substantial number of classes changed their hierarchy nesting level (i.e. the number of super classes) and the number of methods defined. This implies that most of the changes were in the middle of a class hierarchy which is indeed typical for a major restructuring. Yet, during the extension phase none of the classes changed their hierarchy nesting level, but a significant amount increased or decreased the number of children. Thus, all changes were made to the leaves of the inheritance hierarchy which is indeed typical for extensions. Consequently, the 1.0 2.0 restructuring did improve the inheritance structure since the subsequent 2.0 2.5 transition really exploited the inheritance hierarchy" [Mens01].

Comparing two releases in can give information like classes added, classes removed, increase and decrease in number of methods and changes in the class hierarchy. This can provide significant help in under standing the evolution that took place. For example can a stable number of classes and an increase in methods mean that there has been an extension in the form of added functionality. Also, new functionality usually means changes in classes at the leafs of the class hierarchy. On the other hand, if changes happens in the middle or top of the class hierarchy it is more likely that restructuring have been done.

4.2   Origins for data

Before data can be analyzed and hypotheses tested, data has to found. This can be trivial and it can be tricky. It is important to find a software system that is relevant for the research question or hypothesis in mind and it is important to know the context of the data to be analyzed.

With permission, using a commercial software system can be a good source. Commercial development usually has a relatively stable workforce, well-defined development

strategy and clear business goals. Although a good and defined source for answering questions, companies are often reluctant to give away data on the development of their software system since it is critical to their business. It can be hard to find a suitable commercial software system where the company is willing to share its data.

In this case, open source projects can be of great value. There are a large number of open source projects available with a lot of data collected during development to choose from. However they often hold a large number of undocumented and uncontrolled variables. Varying number of developers, lack of schedules and differing or unclear goals are common. Requiring a formal project can limit the suitability of using open source projects, although they do exist.

Whatever data origin chosen, it is important to make sure it is generalizable enough to cover the questions of interest and that relevant data is or have been collected during development. Interesting data from a software system can be extracted from several data origins. There are two important sources of interest for software evolution, change requests and the change reports. Also various attributes of the source code is usually very useful.

There is a large number different system, standards and routines used in the process of creating the data. Change requests can be reported using a physical paper form or through a separate software system design to manage a vast number of requests. For archiving the changes, there are several different configuration management and version control systems. Some provide easy extraction of data, other require manual work.

4.3 Metrics used in this report

4.3.1 Cyclomatic Complexity

Cyclomatic complexity (CC) was introduced by McCabe in 1976 [McCa76], and is often reffered to simply as program complexity or McCabe's complexity. It was construct to meet the challenge of how to modularize a software system to be more testable and maintainable. At the time this was usually done by limiting the programs physical size. However McCabe states that several properties of the graph-theoretic complexity shows that complexity is independent on physical size [McCa76].

It is one of the more widely accepted software metrics and can be considered a board measure for soundness and confidence [VanD00], it also indicates the psychological complexity of a program and the number of necessary test. It measures the number of linear-independent paths through a programs source code and results in a single ordinal number that can be used for comparison.

Cyclomatic complexity is computed using a graph that describes the control flow of the program. The nodes of the graph correspond to a block of code where flow in sequential and the arcs corresponds to branches taken in the program. By definition,

$$CC = E - N + P$$

where

CC = cyclomatic complexity
E = the number of edges of the graph
N = the number of nodes of the graph
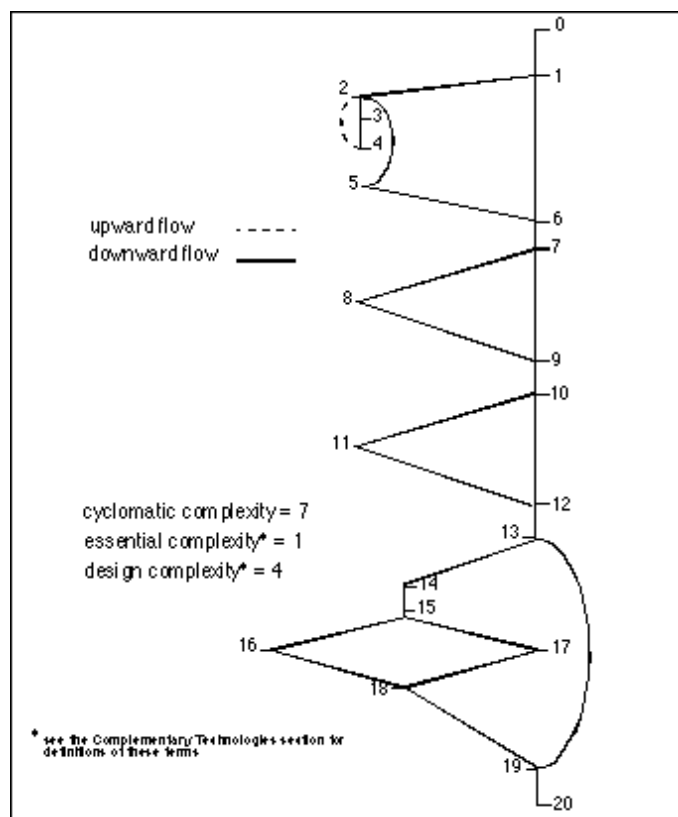P = the number of connected components

Figure 4.1: Connected Graph of a Simple Program [VanD00]

The cyclomatic complexity of a section of a source code is the number of paths through it. A method with no branches has a cyclomatic complexity of one. The complexity is increased by one for every occurrence of a branch, i.e statements like 'if', 'for', 'while', 'case' and 'catch' or operators like 'and', 'or', '&&' and '||' or other variants used by various languages. For instance a section with a single if-statement would have a cyclomatic complexity of two since there are two paths through the code, one path for true and one for false. Inserting or deleting functional statements does not affect the codes cyclomatic complexity. Cyclomatic complexity is a procedural rather than an object-oriented metric but still gives meaning in object-oriented development at method. It is independent of language of platform.

A problem with the reliability of cyclomatic complexity might be that certain parts of a program would expect to have high complexity. This would include UI modules, certain code for data validation and error recovery. In these cases comparing them to other classes might give a wrong picture of where complexity problems are the biggest.

4.3.2 Lack of Cohesion in Methods

Cohesion is central concept in object-oriented programming and can loosely be described as the "togetherness" of a class. It gives an indication on whether a class represents a single abstraction or multiple. Classes that represent more than one abstraction should be refactored to multiple classes representing multiple abstractions.

High cohesion means that strong separation of different abstractions, thus lack of cohesion usually indicates poor code where abstractions are unclear. Low cohesion of

methods implies a large likelihood of errors during the development process, because of the increasing complexity.

Although a relatively easy concept to understand by humans, it is difficult to define a clear concrete definition that can be measured by a metric. Even so, some measures exists e.g. that of Chidamber and Kemerer and that of Henderson and Sellers. These two measure lack of cohesion in methods, so a low score is considered better than a higher score.

There are certain problems in measuring cohesion for some languages. E.g. in Java classes with *getters* and *setters* (like getProperty() and setProperty()) get high lack of cohesion using the methods described bellow although this is not an indication of a problem.

Chidamber-Kemerer

Chidamber and Kemerer define Lack of Cohesion in Methods as the number of pairs of methods in a class that don't have at least one field in common minus the number of pairs of methods in the class that do share at least one field. When this value is negative, the metric value is set to 0 [Chid93].

It is defined as follows:

Let C be a class with n methods M1, M2, ..., Mn
Let Ik = the set of instance variables used by method Mk
Let $\mathbf{P}$ = (Ii,Ij) | Ii joined with Ij = 0
Let $\mathbf{Q}$ = (Ii,Ij) | Ii joined with Ij <> 0

then,

Lack of cohesion = $|\mathbf{P}|$-$|\mathbf{Q}|$ (or 0 if $|\mathbf{P}|$>$|\mathbf{Q}|$)

Henderson-Sellers

Henderson-Sellers [Hend95] defines Lack of Cohesion in Methods as follows:

Lack of Cohesion in Methods = $\frac{<r>-|\mathbf{M}|}{1-|\mathbf{M}|}$

where

$\mathbf{M}$ is the set of methods defined by the class
$\mathbf{F}$ is the set of fields defined by the class
$r(f)$ is the number of methods that access field f, where f is a member of $\mathbf{F}$
$<r>$ is the mean of $r(f)$ over $\mathbf{F}$

For some programming languages, it can be useful to make small modifications to the metric to prevent undesirable results. For instance excluding methods that do not access any fields since these are often used in polymorphic languages to call a method on the super class or exclude fields that are not accessed by any methods, since most compilers will inform the developer of this.

4.3.3 Other metrics used

The following metrics where also used in the research questions or for the hypotheses:

Size: Non-comments lines of code (NLOC) are used for size measure. Alternatively bytes or all lines of code (LOC) could be use, but NLOC is considered better since it relates to the amount of statements used.

Critical development: Changes in status for bugs in Bugzilla marked as *blocker* or *critical*. (Compare with general development defined as changes in status for any bug.)

Fault density: Faults are defined as the amount of bugs reported to and accepted in Bugzilla minus the amount of closed bugs. Fault densitiy is the amount of fault per thousand lines of non-comment code (KNLOC).

4.4 Research questions

RQ1 What is the size and cyclomatic complexity of the largest modules?

Motivation for RQ1: Motivation for RQ1: The intention of this question is to get a better understanding of the data and trends for the system. The three largest modules are measured to enable comparison and show local growth.

Metric for RQ1: Metric for RQ1: Size (lines of non-comment code) and cyclomatic complexity will be measured for the three largest modules (`portage`, `emerge` and `repoman`) and the results for each release will be compared.

RQ2: Does the file structure change and if so what are those changes and reasons for them?

Motivation for RQ2: In theory, there is no need for attention on the file structure for application written in Python. Changes to the file structure are only done to get a better overview and better coordination of work. Therefore looking at how it evolves over time, we can get an idea of the situation of a project. It's expected that to gradually see an increase in files as part of a strategy for improving maintainability.

Metric for RQ2: Samples of the file structure for each release will be analyzed. The amount of folders and files and the distribution of files will be measured and compared.

RQ3: How does the number of files, classes, functions and code lines change over time?

Motivation for RQ3: Previous research by the author showed a linear increase of the number of code lines of the two largest files for the project. Does the same apply across the all files in the project? Also does the number of files, classes and functions increase in similar fashion?

Metric for RQ3: Data from every month will be collected. The number of files, classes, functions and non-comment code lines will be measured for each release and compared.

RQ4: How does the distribution of class sizes change between releases?

Motivation for RQ4: In previous research, it became clear that the size of the two major files increased linearly. However no data was analyzed on the size classes. This research question tries to answer the question on whether all classes increase in size like the two major files did or whether only certain classes increase while others do not.

Metric for RQ4: The size of different classes will be measured for each release and seperated into three categories. Classes with 30 or less non-commented lines of code will be classified as small classes, classes with more than 30 and less than 100 will be classified as medium and thus with 100 or more as large. The result from different releases will be compared.

4.5 Hypotheses

H1: The share of commented lines of code is independent of file size.

Motivation for H1: When measuring the size of source code, the number of code lines is usually used. However, there is no clear consent on whether this should include commented lines or not. This hypothesis looks at the difference in these two measures. If the hypothesis is true, the difference is only relative, but if false, including comments or not when measuring size might give different results.

Metric for H1: Data on total lines of code and lines of commented code will be extracted for every month and compared. It should not vary more than 10 % from average during the three year development of Portage.

H2: Critical development is more stable than general development in open source projects.

Motivation for H2: Previous research [Hagl04] showed that development activity in the Gentoo project decreased in May and November and increased in August and February, a pattern assumed to be affected by school and vacations. The intention of this hypothesis is to see how open source development might be affected by fluctuation in development activity. The assumption is that security and severely important bugs have a high priority and are fixed first, thus making critical development more stable than development in general.

Metric for H2: Data on resolved bugs will be collected from Bugzilla containing bugs from the whole Gentoo Linux operating system. Activity on bugs marked with severity *blocker* or *critical* will be considered critical while those marked with severity *major*, *normal*, *minor*, *trivial* or *enhancement* will be considered non-critical. It is expected that the number of resolved critical issues will be more stable than the resolved non-critical issues.

H3: Cyclomatic complexity increase in an evolving system increases its complexity unless work is done to reduce it.

Motivation for H3: The intention of this hypothesis is to prove hypothesis H5.

Metric for H3: A calculation of McCabe's cyclomatic complexity will be done for different releases of Portage. The result for each release will be compared to look for a trend of increasement.

H4: Cohesion decrease in an evolving system unless work is done to reduce it.

Motivation for H4: The intention of this hypothesis is to prove hypothesis H5.

Metric for H4: A calculation of lack of cohesion using metrics by Chidamber-Kemerer og by Henderson-Sellers will be done for different releases of Portage and divided by the total lines of non-commented code. The result for each release will be compared. There should be a clear increase for both metrics for lack of cohesion.

H5: An evolving system increases its complexity unless work is done to reduce it.

Motivation for H5: A widely debated statement. The goal here is to see whether this can be considered true in the context of Portage.

Metric for H5: The hypothesis will be accepted if H3 and H4 are accepted.

H6: Fault density increase with complexity.

Motivation for H6: Intuitively, the number of error should increase when the size of a program increases. But increasement in the density of errors is not given. This hypothesis will see whether there is a connection between complexity and the density of errors.

Metric for H6: Fault density will be defined as described earlier in this chapter and increase in complexity will be considered true if hypothesis H5 is accepted.

28

Chapter 5

Results

5.1 Research question and hypotheses

RQ1 What is the size and cyclomatic complexity of the largest modules?

Figure 5.1 shows how the size (non-comment lines of code) of the three largest modules (`portage`, `emerge` and `repoman`) for different releases. This corresponds the observation in previous work by the author [Hagl04].
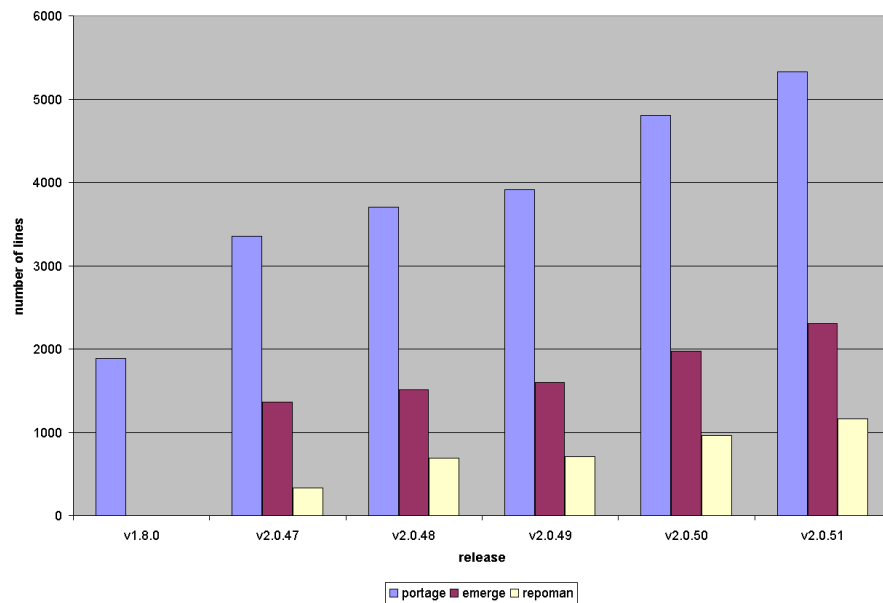

Figure 5.1: The size of the three largest modules

Figure 5.2 shows difference in size between releases. We see that although there is a clear trend of increase in size shown in figure 5.1, local changes are more erratic.

Figure 5.3 shows the total cyclomatic complexity for all function in the largest three modules. We see a general increase similar to the growth in lines of code, except for `emerge` in release 2.0.50 that has a significant temporary growth.

From these figures we see that the relative difference in lines of code and cyclomatic complexity seems to be larger for the larger modules. There are several possible
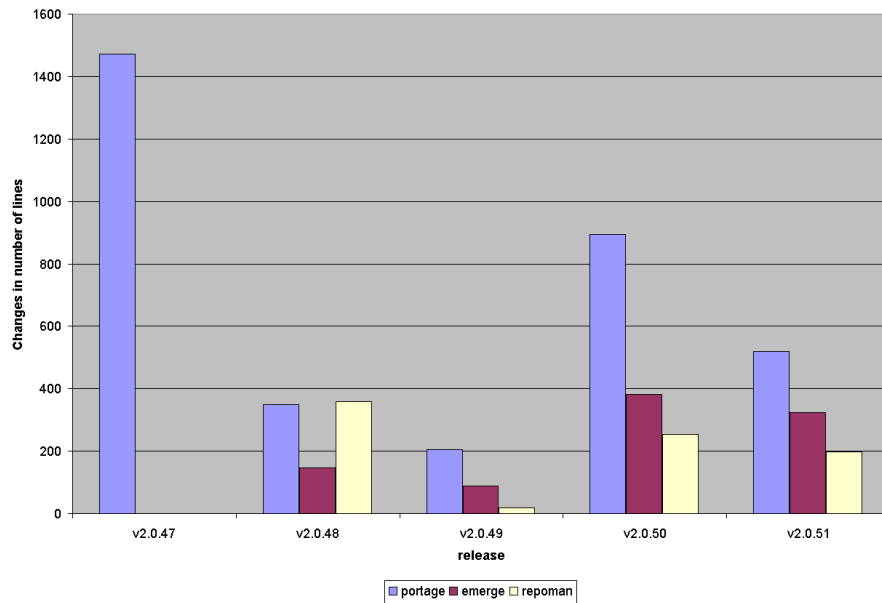
Figure 5.2: Changes in size from previous release for three largest modules

explanations; one might be that the modules receiving the most development activity tend to increase cyclomatic complexity per lines of code.

RQ2: Does the file structure change and if so what are those changes and reasons for them?

Figure 5.4 shows in which releases different source files exist. It does not seem to be any large change in structure. Instead changes are focused around the addition of new tools and the extension of functionality from existing files to a new. The `pym` folder is an example of the latter where functionality from `portage.py` is extracted to new files and `portagedb.py` is removed and split to several new files. The `bin` folder is an example where new files are added when new tools are added. In both examples, changes are largely limited to the introduction of new files, in the case of the `pym` folder the new files contains functionality used by existing files and in the `bin` folder new files are new functionality used multiple places.

Changes in file structure seems to be largely a matter of organizing functionality in an increasing number of files.

RQ3: How does the number of files, classes, functions and code lines change over time?

In figure 5.5, the increase in the number of files, classes, functions and code lines is shown, and can largely be considered linear over time. A notable exception is the big increase in classes for the last release (2.0.51). This is due to the introduction of exception handling in Portage which resulted in 21 new trivial classes.

Figure 5.6 show the same increase as above, but weighted so that it can be compared relatively. Intuitively, in a system with no structural changes, we might expect to see the highest increase in lines of code and less increase in measures of higher conceptual level, with the lowest increase on modular level. The reason is that
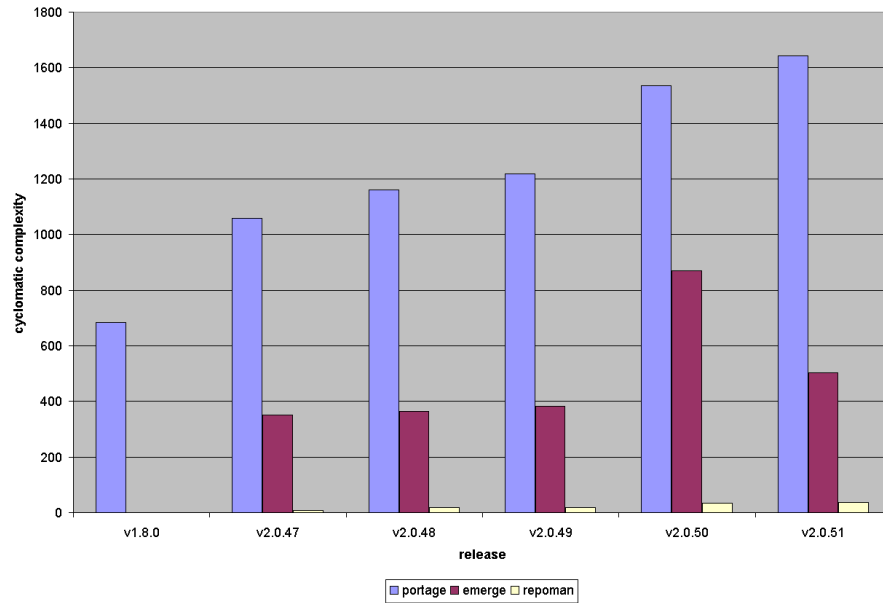
Figure 5.3: The cyclomatic complexity of the three largest modules

we might expect the average size of function to increase (which results in higher increase in code lines than functions), the average number of functions in classes to increase (which results in higher increase in functions than classes) and the average number of classes in modules to increase (which results in higher increase in classes than modules). Although this is almost true, there is an exception for modules. Instead of having the lowest increase, it has the highest.

The reasons for the high increase in modules is possibly that new functionality, like using database, help or exception handling, is added as a new module or extracted from an excisting one. The structuring of this is mostly a matter of maintainability and separation is more encuraged here than in e.g. classes.

The conclusion is that the number of files, classes, functions and code lines increase linearly with exception when certain functionality is introduced that skew the results (like the introduction of 21 trivial classes when exception handling was added). Also, the relative increase in share is smaller for "higher level" measures with the exception of modules.

RQ4: How does the distribution of class sizes change between releases?

With the exception of the last Portage release (2.0.51), figure 5.7 shows how the number of large classes clearly increase linearly while the number of medium classes is relatively stable. The number of small classes is unstable between two releases, but fairly stable over time. The notably increase in the number of small classes in the last Portage release (2.0.51) is due to the introduction on the module `portage_exception` containing 21 5-line classes skewing the data. Ignoring this data the number of small classes would be five for release 2.0.51.

Figure 5.8 shows how the share of the different class sizes change between releases. We see (with the exception of the skewed data from the introduction of `portage_exception` explained above) that the share of large classes clearly increase

		v1.8.0	v2.0.47	v2.0.48	v2.0.49	v2.0.50	v2.0.51
pym	portage.py	✳	✳	✳	✳	✳	✳
	portagedb.py	✳	✳	✳	✳		
	portage_core.py	✳	✳	✳	✳		
	portage_core2.py	✳	✳	✳	✳		
	xpak.py	✳	✳	✳	✳	✳	✳
	output.py		✳	✳	✳	✳	✳
	cvstree.py			✳	✳	✳	✳
	dispatch_conf.py				✳	✳	✳
	getbinpkg.py				✳	✳	✳
	getpkg.py				✳		
	portage_utils.py				✳		
	portage_db_anydbm.py					✳	✳
	portage_db_cpickle.py					✳	✳
	portage_db_flat.py					✳	✳
	portage_db_template.py					✳	✳
	portage_db_test.py					✳	✳
	dcdialog.py						✳
	emergehelp.py						✳
	portage_checksum.py						✳
	portage_const.py						✳
	portage_contents.py						✳
	portage_data.py						✳
	portage_dep.py						✳
	portage_exception.py						✳
	portage_exec.py						✳
	portage_file.py						✳
	portage_gpg.py						✳
	portage_localization.py						✳
	portage_locks.py						✳
	portage_util.py						✳
bin	chkcontents		✳	✳	✳	✳	✳
	db-update.py		✳	✳	✳	✳	✳
	dispatch-conf		✳	✳	✳	✳	✳
	dohtml		✳	✳	✳	✳	✳
	dopython		✳	✳	✳	✳	✳
	ebuild		✳	✳	✳	✳	✳
	emerge		✳	✳	✳	✳	✳
	emergehelp.py		✳	✳	✳	✳	
	env-update		✳	✳	✳	✳	✳
	pdb		✳	✳	✳		
	pkglist		✳	✳	✳		
	pkgmerge		✳	✳	✳	✳	✳
	pkgmerge.new		✳	✳	✳	✳	✳
	pkgname		✳	✳	✳	✳	✳
	regenworld		✳	✳	✳	✳	✳
	repoman		✳	✳	✳	✳	✳
	xpak		✳	✳	✳	✳	✳
	fixpackages			✳	✳	✳	✳
	portageq				✳	✳	✳
	mirror.py				✳	✳	✳
	dispatch-conf-dialog					✳	✳
	emerge.orig					✳	
	fix-db.py					✳	✳
	fixvirtuals					✳	✳
	md5check.py					✳	✳
	archive-conf						✳
	clean_locks						✳
	fixvardbentries						✳
	pemerge.py						✳

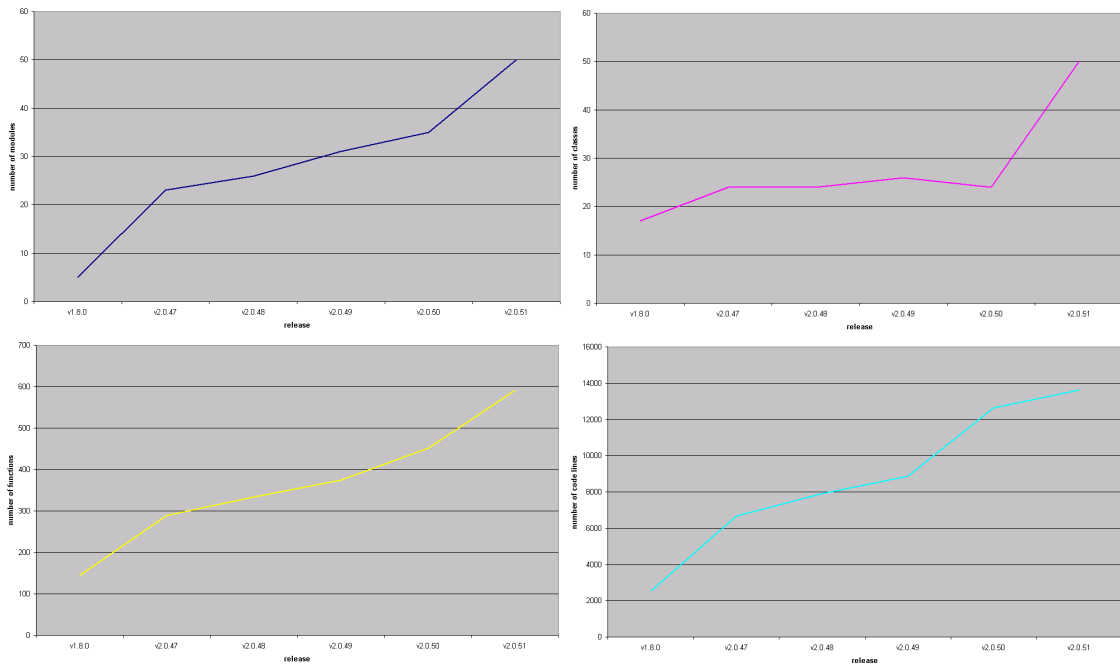Figure 5.4: Changes in file structure

32

Figure 5.5: Increase in modules, classes, functions and lines of code between releases
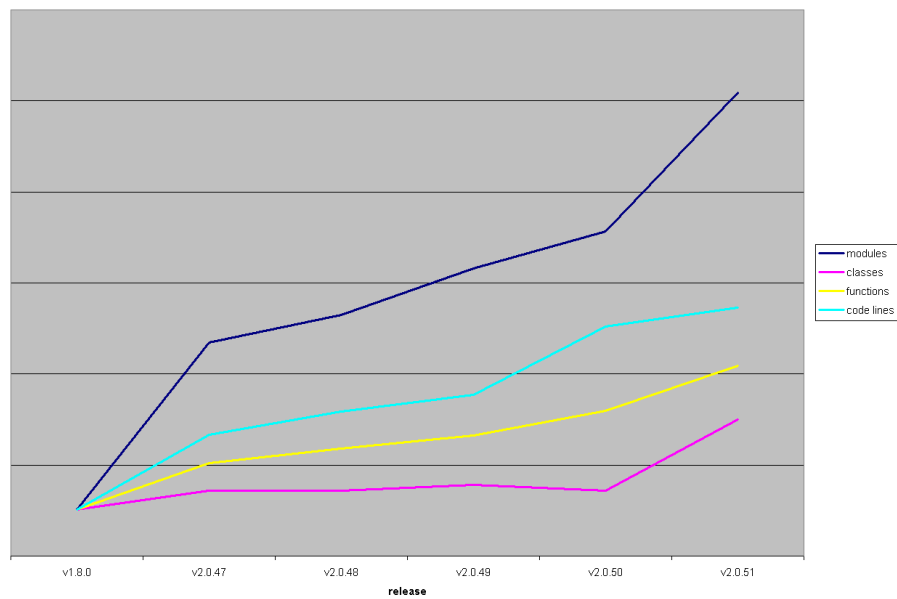

Figure 5.6: Increase in modules, classes, functions and lines of code between releases weighted and compared
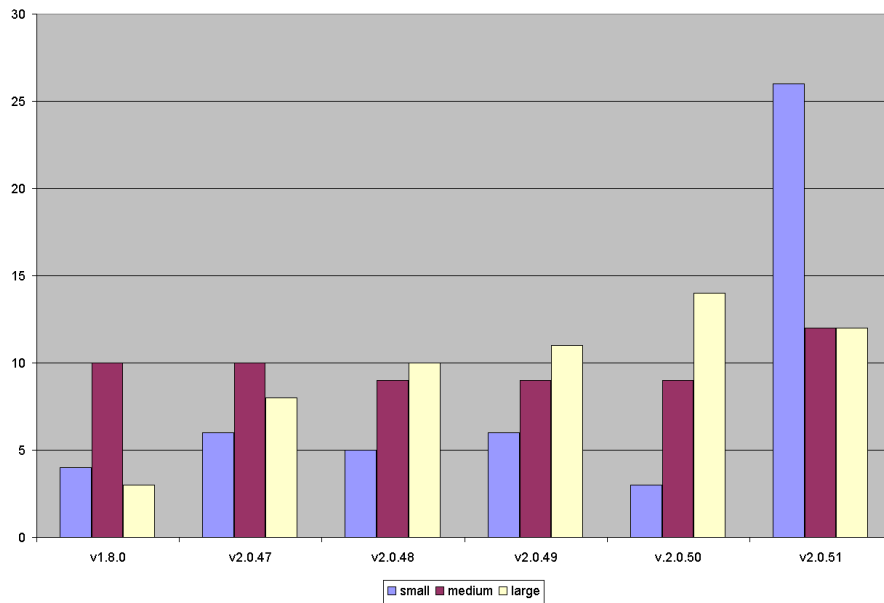
Figure 5.7: Distribution of small, medium and large classes

to more than 50 % for release 2.0.50.

The data seems to show two things. Firstly, not only do the largest classes increase in size, but the amount and share of large classes increases too.

Secondly, special events (in this case the introduction of the `portage_exception` module) has significant effect on the share of class sizes.

The conclusion should be that there seems to be an increase in large classes over time, either by splitting one large class into two large classes or by increase in size of a non-large class. Also, occasionally the introduction of new functionality (in this case exception handling) introduce several new classes that can skew results.

H1: The share of commented lines of code is independent of file size. **[Rejected]**

Figure 5.9 shows how the share of commented lines in `portage.py` changes over time. It does not appear to be stable and the variation is more than 10 %. The hypothesis is therefore rejected.

Therefore the question of including comments or not when measuring lines of code can have an affect on the results.

H2: Critical development is more stable than general development in open source projects. **[Rejected]**

Figure 5.10 shows how critical development (critical bugs resolved) compare to total development (bugs resolved) throughout 2004. No conclusion can be drawn from this.

A closer look at the data by measuring the standard deviation against the average number of bugs resolved is needed. We then get that the standard deviation is 10,74 % of the average for critical development and 10,90 % for total development. These numbers are so similar that the hypothesis is rejected based on this case.
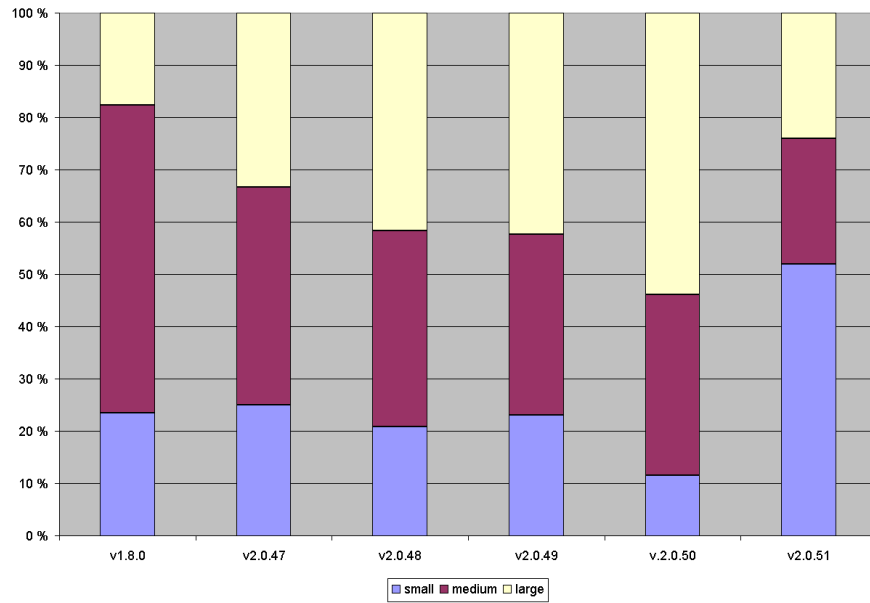
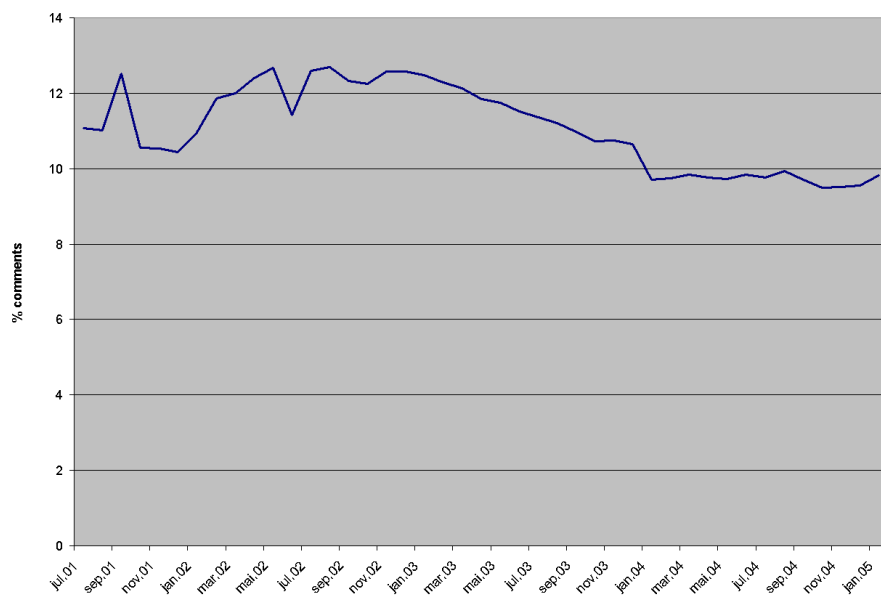Figure 5.8: Share of distribution of small, medium and large classes

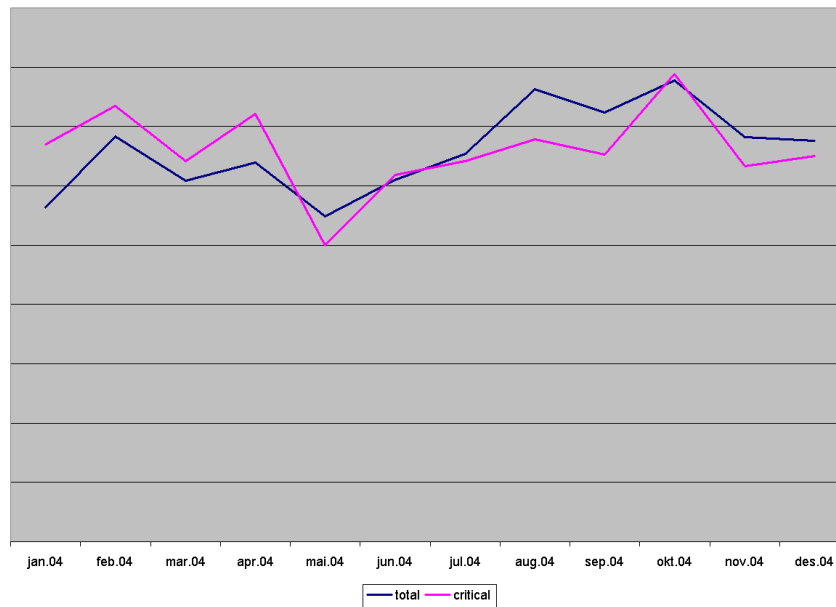Figure 5.9: Share of comment lines in portage.py

35

Figure 5.10: Critical development compared to total development

H3: Cyclomatic complexity increase in an evolving system increases its complexity unless work is done to reduce it. **[Accepted]**

Figure 5.11 shows the accumulated cyclomatic complexity for different releases of Portage. It clearly increases for each release.

Since Gentoo Portage is considered a case where work to reduce complexity is not done and cyclomatic complexity increase, H3 can be accepted.

H4: Cohesion decrease in an evolving system increases its complexity unless work is done to reduce it. **[Accepted]**

Figure 5.12 shows the accumulated lack of cohesion (using metric by Chidamber-Kemerer) for different releases of Portage. It clearly increases for each release.

Figure 5.13 shows the accumulated lack of cohesion (using metric by Henderson-Sellers) for different releases of Portage. It clearly increases for each release.

Since Gentoo Portage is considered a case where work to reduce complexity is not done and lack of cohesion using both Chidamber-Kemerer and Henderson-Sellers increase, H4 can be accepted.

H5: An evolving system increases its complexity unless work is done to reduce it (Lehman's 2nd law). **[Accepted]**

By considering lack of cohesion and cyclomatic complexity as measures of complexity, H3 and H4 concludes that the complexity in Portage increases and H5 can be accepted.

H6: Fault density increase with complexity. **[Accepted]**
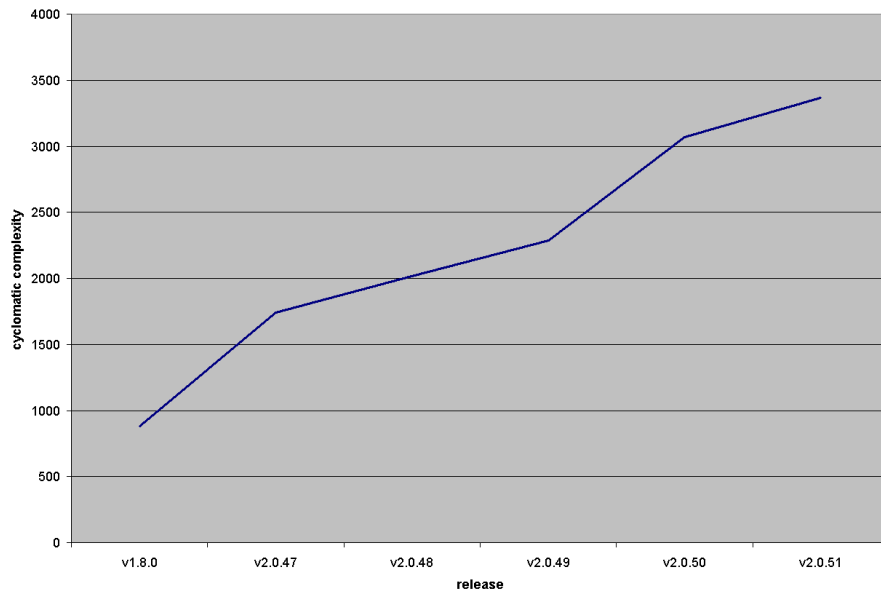
36

Figure 5.11: Cyclomatic complexity in Portage releases
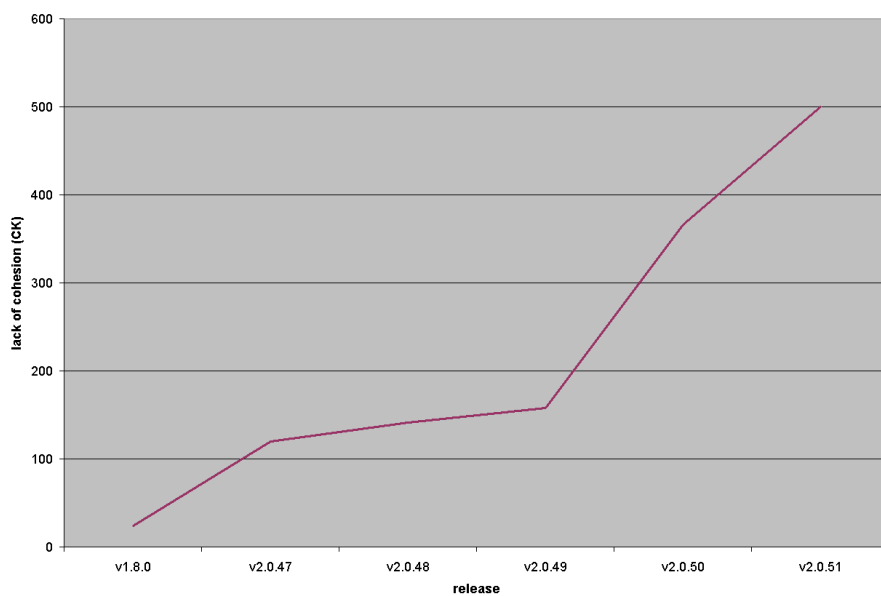

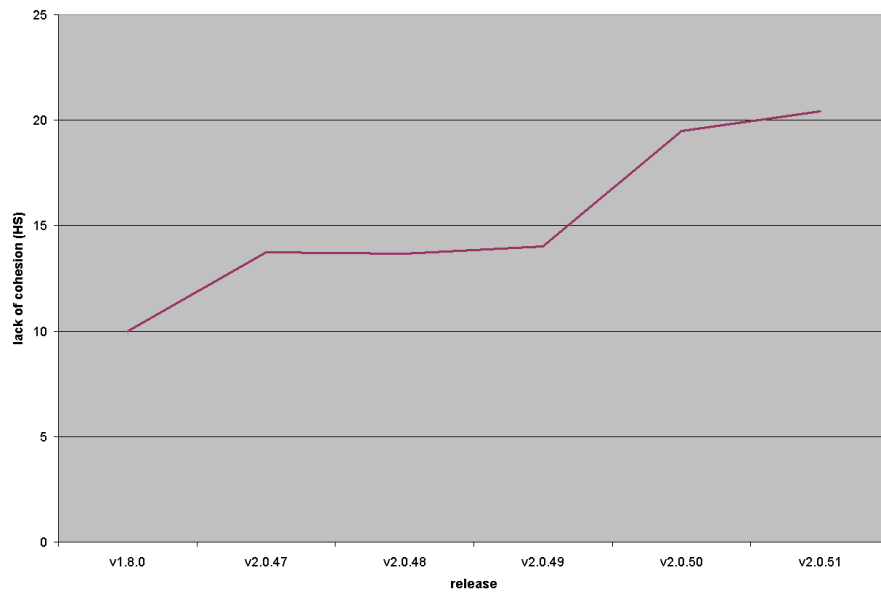Figure 5.12: Lack of Cohesion in Portage releases (Chidamber-Kemerer)

Figure 5.13: Lack of Cohesion in Portage releases (Henderson-Sellers)

Figure 5.14 shows that the fault density clearly increases between releases. From H5, we know that complexity increases and we can therefore accept the hypothesis. (This is however no proof that the complexity is the cause of increasement in fault density.)

5.2 Laws of Program Evolution Dynamics

We see how the same laws of trends and software decay found by Belady and Lehman [Bela76] apply to a system developed about 30 years later. This section attempts to describe how the findings in this thesis relates to them.

5.2.1 Law of continuing change

Research question RQ3 and RQ4 showed the inevitability of growth. RQ3 showed the system growth for different levels (files, classes, functions and code lines) and how they all increased over time in order for the system to meet constant requirement for functionality. RQ4 showed how this growth also results in an increasing share of larger classes.

This is assumed to be a result of constant need for new functionality. This is supported by the results in RQ2 which shows that new files/modules are added as a result of introducing new functionality.

5.2.2 Law of increasing entropy

Hypotheses H3, H4 and H5 showed how the systems complexity increased by different complexity measures. This relates to the second law described by Belady and Lehman in section 2.3 that deals the consequence of growth, namely increasement in complexity unless effort is made to prevent it.
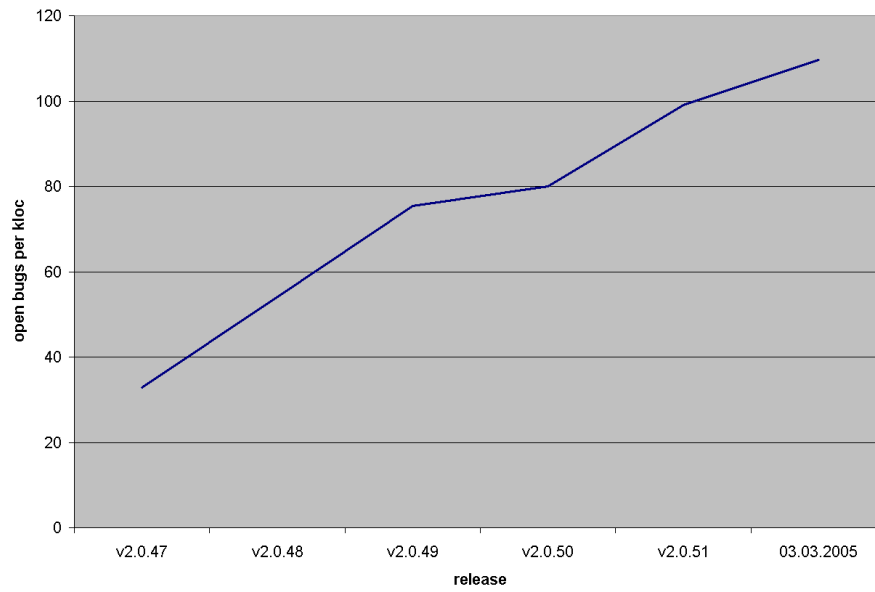
38

Figure 5.14: Fault density of Portage

5.2.3 Law of statistically smooth growth

Research question RQ1 and partly RQ2 supplements the findings in RQ3 and RQ4 to show that even though there are local non-linear growth, there are smooth long-term trends.

Chapter 6

Discussion

6.1 Validity

The four threats to validity - conclusion, internal, construct and external - outlined in [Cook79] is used in this section to describe the validity of the study.

6.1.1 Conclusion validity

Conclusion validity concerns issues that affect the ability to draw the correct conclusion about relations between the treatment and the outcome. Whether there is a strong enough statistical significance.

No statistical tests on significance where used for the questions and hypotheses in this thesis. However, the data is collected from almost three year of Portage development between December 2001 and October 2004. There are about 50 classes and over 16 000 lines of code in the last release.

Also, most of the trends found have clear visual proof.

6.1.2 Internal validity

Internal validity concerns issues that may indicate a casual relationship, although there is none. How factors unaccounted for can have affected the results.

In RQ3 and RQ4, where lines, functions, classes and modules where counted and measured, relatively small adjustments can skew the results. For instance, in RQ4 for release 2.0.51 exception handling was introduced and caused the share of small classes to increase dramatically. Similar events might affect other results.

In hypothesis H2, critical development activity was defined as the number of bugs resolved that are marked *blocker* or *critical* (the two most sever categories) in Bugzilla. This data might be skewed by inaccurateness when classifying a bug. Different users and developers might have different opinions on the criteria for a bug being critical. Also a rather large portion of the bugs (more that 50 %) are marked normal, probably since this is "safe" and since it is the default option when reporting a bug. Also, depending on the practices used in the project, some bugs might not be public but instead just fixed directly with a patch. This might cause the data to show either more or less critical development. On the other hand in hypothesis H2, it was the stability of the development and not amount than was compared. For this case the results is not biased.

In hypothesis H6, the number of bugs where counted and each bugs was considered an error. However in addition to managing bugs, Bugzilla is used for managing feature requests. These feature requests are usually represented as a "bug" with severity *enhancement*. Although they do not represent a large portion of the "bugs" these feature requests are included as faults is H6 which skew the data somewhat.

6.1.3 Construct validity

Construct validity refers to the extent to which the setting actually reflects the construct under study. Whether poor measures have been used.

The research questions mostly use simple concrete measures and so has strong construct validity.

Measuring critical development, complexity and cohesion is harder. Measuring critical development by looking at the severity in Bugzilla might at best measure development perceived as critical by the developers.

Evaluation on using cyclomatic complexity as a measure for complexity is widely debated and has certain weaknesses as described in section 4.3.1.

For cohesion, metrics for lack of cohesion by both Henderson and Sellers and Chidamber and Kemerer where used. Evaluation of there are described in section 4.3.2.

This thesis tried to compare logical changes and samples and not just physical, it used compared releases instead of a time-based measure. Since the development of the system used is largely incremental, the actual difference in these two measures might not be significant.

A larger concern for the construct validity is the fact that even though the results showed increase in a number of measures, it is not certain to what extend this was a simple result of the fact that the system increased in number of lines or not. For instance the complexity increase, but this is intuitively expected when code lines increase. It can be debated whether this increase is due to more lines of code or due to consequences of continuous change.

6.1.4 External validity

External validity concerns the ability to generalize results outside the settings used. The correct subject, environment and timing is necessary.

The research questions are largely not generalizable, although similar projects might give the same results as in RQ2. Also one might expect to see the same trends on increasement other places.

Hypothesis H1 might be generalizable for open source project with the same complexity and number of developers.

In hypothesis H2 should be generalizable for all open source project, since it collected bugs from the whole Gentoo Linux operating system consisting of a large amount of diverse open source projects. Usually what happens is that the bug is reported for Gentoo Linux and sent to the specific external project and fixed there before an update is sent back.

Hypotheses H3, H4 and H5 concerning complexity can be generalized to other project where complexity and structure have not been considered.

Hypothesis H6 should be generalizable for all software projects, although the results from this thesis do not support this assumption and further study is needed.

6.2 Experience with open source

Part of the goal of this thesis was to look at how open source projects can be used when doing empirical research on software system. It should be noted that the author has no experience with using non-open source project for data analysis and so this is not a comparison.

6.2.1 Understanding development history

For reasons of data interpretation and generalization, it is not enough to consider only the data. To be able to make reasonable interpretation of data collected from a project, it is necessary to see it in context. What development method was used? What concerns where in focus at a certain time? Are there any significant events? And so forth.

Characteristic for open source development is public archived discussion and more adaptable development (rarely follows a "textbook"). Common for open source projects is the use of mailing lists to coordinate activities and discuss solutions. An archive of these mailing lists is usually publicly available on the web. Although it is usually too big and contain too much irrelevant email for getting a decent overview, it can give an idea of what happened around a specific event and can be an easily accessible source for understanding development history. Alternatively IRC, blogs or other sources are used.

Open source development is less of a planed effort and more opportunistic than industrial project. The development process is largely formed around the developers and naturally evolves, either to adapt to new circumstance or end with development stop. A certain event might attract a several new contributors and change how things get done.

One might say that open source projects do not have much reflection, risk consideration and requirement for strict business consideration and lack milestones and other events to explain development history.

6.2.2 Dialog with developers

In addition to just analyzing data, it is useful to get in a dialog with the developers to clear up the understanding of development, like social interaction and motivation.

An obvious difference here between open source and other projects is that open source projects most open source projects consists of voluntary developers. This simplifies the barrier by not having to go through a superior or other formal channels to the developers. Also open source developers are usually more open to questions and, unless to busy, usually answer questions from the users. Also project discussion is public and it is easy for an outside person to participate, like on a mailing list. Chances are there is a person capable of answering a question.

6.2.3 Generalisation

When doing empirical study on open source project the question of relevance to other software project arises. A lot of the research on software development is done around different development practices like unit testing, uml, agile development, pair programming and release frequency. It is important to note that open source is not a

development practices like the once mentioned, but can include all of them. In fact, open source project has a lot in common with agile development since agile development is inspired by open source.

Even tough agile development might be a good starting point for generalisation, it might be hard to conclude on details and general understanding of development process for individual project. Often there is no authority or a single person with complete understanding. And the development process itself is evolving.

Differences exist however on the issue of motivation. Open source project is largely driven by to things; fun and search for knowledge. Although this is a common motivation in non-open source projects, for many open source projects it is the driving force.

6.2.4 Data sources and tools

Section 3.3 describes common data sources used in open source development and how they can by utilized for data mining. Section 3.4 describes some of the tools used and other tools evaluated.

Experience with data mining was positive. A main strength was the fact that most of the data was publicly available and easy to access. The tools where essential, since the repositories themselves provided little useful data themselves. For instance, the data from the source used to compare different releases where all extracted by Pythonmetrics analysis. The data from Bugzilla had to be extracted by a script. The data was analyzed in a database and in spreadsheet applications before presented.

The conclusion is that open source provides good repositories for data mining, but some effort and the right tools are necessary to get useful data. Combining some of the data found might be difficult since common repositories like CVS and Bugzilla (managing files and bugs respectively) have no direct connection and there are therefore no trivial way to combine bug activity and source code activity.

6.3 Thesis process

GQM, as explained in section 2.4, starts by establishing a set of goal, then a set of questions whose answers will help achieving the goal and lastly a set of metrics to answer those questions. This rational process works when there is a clear goal and the means to answer this goal is available. This is not the case for this thesis. The goal is not concrete and the means to answer the most interesting ones is not necessary available because there is no control over data used, since it is already collected during development. This is in contrast to common GQM usage where wanted data is decided before development.

A modified approach was therefore needed; a bottom up-approach was used in addition to a top down-approach. For practical reasons, instead of just looking at what should be answered, it was also considered what could be answered. The author therefore had to construct research questions and hypothesis and at the same time construct measurable general data to see what was possible and whether there where data to answer these questions.

This approach requires focus on prioritizing; it is a wasted effort to formulate questions and hypothesis that can't be answered and to generate data that do not answer any question of interest. Also, there should be a focus on prioritizing issues that gives the most results for the least effort; traditional cost and benefit evaluation.

There is also the issue of capacity. Any hypothesis that proves to be too hard to test fully should be excluded. As should any data that answers uninteresting or incomplete questions.

Chapter 7

Conclusion

The goal of this thesis was two folded. Firstly this thesis performed and empirical study to answer research question and test hypothesis related to the laws on software evolution outlines by Belady and Lehman, described in section 2.3. Secondarily this thesis used an open source project as data source and tried to take a closer look at tools and data sources used and experience from this.

The research questions and hypothesis showed that the same laws formulated by Belady and Lehman in 1976 still apply to software systems today. Portage was used as an example of a system where there has been constant pressure for new functionality and little has been done on structural maintenance. The results showed how there was an trend of constant increasement in size as a results of constant change to adapt for changes in requirements (Lehman's first law). It showed how the systems complexity increases over time where no effort is made to prevent it (Lehman's second law). Lastly it showed how the system has a long-ranged statistical smooth growth even though there are local stochastic growth, e.g. when certain functionality is added (Lehman's third law).

Finally, we can also saw how the increasement in complexity has lead to and increasement in error density and decrease in maintainability making it more difficult to reach a stable release.

This thesis also looked at how open source can be used for data mining. It showed how common tools and practices used in open source development (CVS, Bugzilla and ChangeLog) can be used as data sources for different purposes and how public repositories and development makes the data easily available. It also show limitations related to how CVS does not provide data on moving of code and files, practical limitation on extracting data from Bugzilla and the difficulties in combining data from different sources.

7.1   Further work

Several paths can be taken to extend on this work.

The hypotheses can be tried on other projects to see whether the same results are found. Also it would be interesting to look at a project where effort has been made to control the complexity and see whether there are notable differences from the findings here. Lack of notable difference would not disprove Lehman's second law, but undermine the rational behind it. A notable difference would strengthen the case for focus on structural maintenance in addition to functional maintenance.

In would also be interesting to use a case where the size of the project (in terms of code lines) has been relatively stable and see how complexity increase or decrease over time for other environmental factors.

Additional metrics can be used, especially for a metric for measuring coupling would be useful since it, together with cohesion, is a central way to evaluate good structural design. Also other object-oriented metric that looks at the class hierarchy could be considered.

Additional tools (see section 3.4) could be considered Maven and XRadar provide several out-of-the-box metrics with graphical outputs for java-based programs. A closer look at Bugzilla might also provide useful, trying to extract better data. Also it is currently popular for open source projects to move away from CVS as code repository and over to more advanced alternative. Some of these code repositories might provide data one changes across multiple files.

The work done in this thesis is largely qualitative and could be followed up with a qualitatively study. I could be interesting to see what factors forced an increase in size and consequently and increase in complexity and what the consequence of the complexity on the development process and on the usage of the system.

Bibliography

[Ball96] Ball, T. and S. G. Eiek. Software visualization in the large. *IEEE Computer*, 29(4), April 1996.

[Basi94] Basili, V.R., Calidiera, G., Rombach, H.D., *Goal Question Metric Paradigm*, In: Maraciniak, J.J. (ed.): Eccyclopaedia of Software Engineering. New York Wiley 1994, pp. 528-532.

[Bela76] Belady, B. and M. Lehman (1976), *"A Model of Large Program Development,"* IBM Systems Journal 15 , 3, 225–252.

[Brit95] Brito, F. Abreu, M. Goulao, and R. Esteves. Toward the design quality evaluation of object-oriented software systems. In *Proc. 5th lnt 7 Conf. Software Quality*, pages 44-57, October 1995.

[Chid93] Chidamber and Kemerer, 1993, *A Metric Suite for Object-Oriented Design*, Working Paper #249, MIT Center for Information Systems, Cambridge, MA, 40 pp

[Chid94] Chidamber, S. R. and C. E Kemerer. A metrics suite for object-oriented design. *IEEE Trans. Software Engineering*, 20(6):476-493, June 1994.

[Cook79] Cook, T.D. and Campbell, D.T., *Quasi-Experimentation Design and Analysis Issues for Field Settings*, Houghton Mifflin Company, 1979.

[Deme99] Demeyer, S. and S. Dueasse. Mettles: Do they really help? In Proc. *Languages et ModUles d Objets*, pages 69-82. Hermes Science Publications, 1999.

[Deme01] Demeyer, S., Mens, T., Wermelinget, M., *Towards a Software Evolution Benchmark*, Proceedings of the 4th international workshop on Principles of software evolution, September 10-11, 2001, Vienna, Austria

[Eick01] Eick, S. G., Graves, T. L., Karr, A. F., Marron, J. S., Mockus, A., *Does Code Decay? Assessing the evidence from Change Management Data* (.pdf). See does code decay. http://csdl2.computer.org/dl/trans/ts/2001/01/e0001.pdf. IEEE Trans. on Software Engineering, 27(1):1-12, Jan. 2001.

[Evan01] Evans, H., *Why Is Distributed System Evolution Not Better Supported?*, Proceedings of the 4th international workshop on Principles of software evolution, September 10-11, 2001, Vienna, Austria

[Fent97] Fenton, N. and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. International Thomson Computer Press, London, UK, second edition, 1997.

[Gall98] Gall, H., K. Hajek, and M. Jazayeri. Detection of logical coupling based on product release history. In *International Conference on Software Maintenance (1CSM '98)*. IEEE Computer Society Press, November 1998.

[Gent] Gentoo Foundation, 2005. *Gentoo Project Homepage*, http://www.gentoo.org.

[Hagl04] Hagli, Andreas Tørå, 2004. *Observed Changes in Software Evolution*.

[Hend95] Henderson-Sellers, 1995, *A BOOK of Object-Oriented Knowledge*, 2nd Ed., Prentice Hall

[Jaza99] Jazayeri, M., C. Riva, and H. Gall. Visualizing software release histories: The use of color and third dimension. In H. Yang and L. White, editors, *Proc. lnt'l Conf. Software Maintenance (ICSM '99)*. IEEE Computer Society, 1999.

[Kaba01] Kabaili, H., R. K. Keller, and E Lustman. Cohesion as changeability indicator in object-oriented systems. In P. Sousa and J. Ebert, editors, *Proc. 5th European Conf. SoJtware Maintenance and Reengineering*, pages 39-46. IEEE Computer Society Press, 2001.

[Lanz01] Lanza, M.. The evolution matrix: Recovering software evolution using software visualization techniques. *In Proc. lnt'l Workshop on Principles of Software Evolution (1WPSE2OO1)*, 2001.

[Lehm74] Lehman, M. M., *Programs, Cities, Students–Limits to Growth*, Imp. Col. 1974, Inaug. Lect. Series, Vol.9, 1970-1974, pp. 211 - 229; also in Gries, 1978

[Lehm80] Lehman, M.M. *Programs, Life Cycles, and Laws of Software Evolution*. Proceedings of the IEEE, vol 68, no 9, 1980.

[Lehm01a] Lehman, M. M. , J. F. Ramil, *Evolution in software and related areas*, Proceedings of the 4th international workshop on Principles of software evolution, September 10-11, 2001, Vienna, Austria

[Lehm01b] Lehman, M. M., *An Approach to a Theory of Software Evolution*, Proceedings of the 4th international workshop on Principles of software evolution, September 10-11, 2001, Vienna, Austria

[Lien80] Lientz, B. P., Swanson, E. B., *Software Maintenance Management*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, 1980

[Mave] Maven Homepage, 2005. http://maven.apache.org/.

[McCa76] McCabe, T. J., "*A Complexity Measure,*" IEEE Transactions on Software Engineering, vol. SE-2, Dec. 1976.

[McKe84] McKee, J. R.. *Maintenance as a function of design*. In Proc. 1984 AIPS national Computer Conference, pages 187-93, 1984.

[Mens01] Mens, T. and Demeyer, S., *Evolution Metrics*, Proceedings of the 4th international workshop on Principles of software evolution, September 10-11, 2001, Vienna, Austria

[Mock00] Mockus, A. and Votta, L. G., *Identifying Reasons for Software Changes using Historic Databases*, From International Conference on Software Maintenance, pages 120-130, San Jose, California, October 11-14 2000

[Moha04] Mohagheghi, Parastoo and Conradi, Reidar: "Exploring Industrial Data Repositories: Where Software Development Approaches Meet", Proc. of the 8th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE'04), 15 June 2004, Olso, Norway, Coral Calero, Fernando Brito e Abreu, Geert Poels and Houari A. Sahraoui (Eds.), pp. 61-77. Affiliated with 18th European Conference on Object-Oriented Programming (ECOOP 2004), 14-18 June 2004, Oslo.

[Mont04] Monteiro, Eric; Østerlie, Thomas; Rolland, Knut and Røyrvik, Emil. *Keeping it going: The everyday practices of open source software*, 2004, submitted for reviewing.

[Pfle91] Pfleeger, S. L.. *Software Engineering: The Production of Quality Software*. Macmillan Publishing Company, 2 edition, 1991.

[Pigo96] Pigoski, T. M., Practical Software Maintenance, Wiley, 1996, pp. 384

[Pyme] Pythonmetric Homepage, 2005. http://sourceforge.net/projects/pythonmetric/.

[Pyth] Pythius Homepage, 2005. http://www.gentoo.org.

[Rajl01] Rajlich, V., *Role of Concepts in Software Evolution*, Proceedings of the 4th international workshop on Principles of software evolution, September 10-11, 2001, Vienna, Austria

[VanD00] VanDoren, E., Sciences, K. and Springs, C. *Cyclomatic Complexity*, http://www.sei.cmu.edu/str/descriptions/cyclomatic.html, Carnegie Mellom University, 2000.

[Venn03] Venners, B., *Don't Live with Broken Windows*, Interview with Andy Hunt and Dave Thomas, http://www.artima.com/intv/fixitP.html, 2003.

[Xrad] XRadar Homepage, 2005. http://xradar.sourceforge.net/.