Understanding Software Evolution

Michael W. Godfrey

Software Architecture Group
University of Waterloo

Background and interests

- Research
  - Software evolution
  - Versioning, configuration management
  - Software architecture, reverse engineering, program visualization
  - Interchange formats for rev. eng. tools
- Software engineering education
  - SIGCSE (J ), CCCEE, IST
  - Professional M.Eng. program (Cornell)
  - SE option (Wloo), SE ugrad program (Wloo)

Overview

- What is software evolution?
  - Why should we care?
- Previous research
- A case study: The Linux OS kernel
- Observations, hypotheses, and future research

What is software evolution?

"Evolution is what happens while you're busy making other plans."

- Usually, we consider evolution to begin once the first version has been delivered:
  - *Maintenance* is the planned set of tasks to effect changes.
  - *Evolution* is what actually happens to the software.

So we can consider maintenance as what we have traditionally considered when looking at how systems change. But what's missing is a more "scientific" consideration of what, exactly, software is and how and why it changes. I'm trying to be a good scientist here and observe what actually happens to the software rather than what was planned. It's a fundamental change in perspective from most studies of long-lived software.

Common "maintenance" tasks

- *Adaptive*
  - Add new features
  - Add support for new platforms
- *Corrective*
  - Fix bugs, misunderstood requirements
- *Perfective*
  - Performance tuning
- *Preventive*
  - Restructure code, "refactoring", legacy wrapping, build interfaces

Many have criticized the use of the term maintenance as it's used for software development. Software doesn't decay in the same sense as physical systems, and the kinds of "maintenance" tasks performed are not the same. Yet we are stuck with the term.

Why should we care?

- Much of the commercial software world operates in perpetual crisis mode.
  - "Fix it, don't try to understand it."
  - Just-in-time program comprehension [Lethbridge]
- … but … large software systems are *major* assets of many businesses
  - Getting it right more important than getting it done fast.
  - Budget and time for preventive maintenance, navel gazing.
- Relatively little research on trying to understand *how* and *why* programs evolve.

One of the core Perl developers, Chip Salzenberg, has said of the current version: "You really need indoctrination in all the mysteries and magic structures and so on before you can really hope to make significant changes to the Perl core without breaking more things than you're adding."

Previous research

- Lehman's laws
- Parnas on software geriatrics
- Eick *et al.* on code decay (10 MLOC telecom)
- Gall *et al.* (10 MLOC telecom)
- Munro, Burd *et al.* (2 MLOC `gcc`)

Lehman's Laws of Software Evolution

- Based on measurement of a few (commercially-developed) systems, most notably IBM's OS 360
  - Originally three laws, now there are eight.
- Controversial as "laws"
  - Has been criticized for strong claims based on limited data.
  - However, it's pioneering work on software evolution and software engineering.

Lehman was one of the first to claim that a large software system is more like an ugly beast than a beautiful theorem. This was quite revolutionary thinking for a British academic in the 1970s.

Lehman's Laws of Software Evolution

1. *Continuing change* — An E-type program that is used must be continually adapted else it becomes progressively less satisfactory.
2. *Increasing complexity* — As a program is evolved, its complexity increases unless work is done to maintain or reduce it.
3. *Self regulation* — The program evolution process is self-regulating with close to normal distribution of measures of product and process attributes.

I won't discuss the laws in detail as there difficult to understand without explanation. Instead, I'll leave them on the slides for reference, and give a nutshell summary.

Lehman's Laws of Software Evolution

4. *Invariant work rate* — The average effective global activity rate on an evolving system is invariant over the product lifetime.
5. *Conservation of familiarity* — During the active life of an evolving program, the content of successive releases is statistically invariant.
6. *Continuing growth* — Functional content of a program must be continually increased to maintain user satisfaction over its lifetime.

Lehman's Laws of Software Evolution

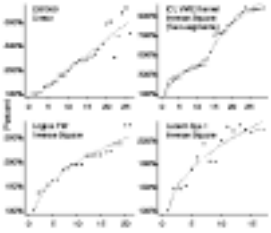
7. *Declining quality* — E-type programs will be perceived as of declining quality unless rigorously maintained and adapted to a changing operation environment.
8. *Feedback system* — E-type programming processes constitute multi-loop, multi-level feedback systems and must be treated as such to be successfully modified or improved.

Lehman's Laws in a nutshell

- Observations:
  - (Most) useful software must evolve or die.
  - As a software system gets bigger, its resulting complexity tends to limit its ability to grow.
  - Development progress/effort is (more or less) constant.
- Advice:
  - Need to manage complexity.
  - Do periodic redesigns.
  - Treat software and its development process as a feedback system (and not as a passive theorem).

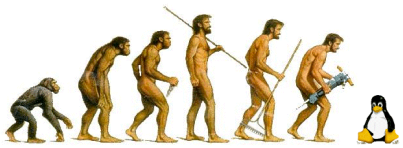
Constant effort/progress might have been reasonable for 1960s OS development, but it's clearly not true for the dot-com or what we used to call the shrinkwrap market. This is also clearly false for open source development projects too. This piqued my interest in examining the evolution of Linux (especially after chatting with Lehman at a workshop in May 1999).

Lehman's examples



In the systems he and his group have examined over time, most seem to obey an inverse-square growth model (y=x^.5). This is consistent with his "laws" that state that growth tends to slow over time. The top left pattern is interesting, and you can see this as several mini-patters in the bottom two. This consists of growth followed by slowing down. His hypothesis is that restructuring is being performed, but I think it's just careful planning. Make some changes, and then see what breaks and fix it before embarking on a major new release. The one outlier here is the OS360 which seems to be linear until the end when it varies wildly. Lehman still does not have a good explanation for either its linear growth or its late variability even years later.

A case study in evolution: The Linux OS kernel



A case study in evolution: The Linux OS kernel

"Evolution in Open Source Software: A Case Study"
[Godfrey and Tu, ICSM 2000]

- It's Linux!
  - Large system, very stable, many releases over several years, many developers
  - Growing mainstream adoption
- Open source development model
  - Interesting phenomenon in itself
  - Easy to track, can publish results, many experts
  - Not much previous study

Evolution of Linux: Questions

- How has Linux evolved over time?
  - Does it obey Lehman's laws?
  - What is the best way to characterize growth?
- How has its (open source) process model affected its development?
- How has the (high-level) architecture
  - changed over time?
  - affected the system's evolution?

Open source development

- Open source development *vs.* open source software
  GNU, Linux, Apache, `vim`, `gcc`, FreeBSD
  vs.
  Mozilla, JDK, Jikes, NetBeans
- "The Cathedral and the Bazaar" [Raymond]
  - Usual goal: scratching an interesting itch, not filling a commercial void.
  - Anyone may contribute, tho owner(s) have final say.
- Usually, developers work part-time and for free.
  - Motivation is peer recognition and personal satisfaction, not money.
  - However, industrial participation also increasing (*e.g.,* Cygnus, IBM)

Open source development

- Largely immune from time-to-market pressures
  - Can release when it's really ready
- Can be hard to control/direct developers
  - Big egos, can't be "fired"
  - What's cool *vs.* what's needed
  - Less "sexy" development tasks often suffer
    e.g., planned testing, preventive maintenance
- Code quality varies widely
  - Some projects have coding standards
  - Unstable/experimental code common (and even encouraged)
  - Quality maintained via "massively parallel debugging", not rigorous testing.

Linux background

- Linux kernel v1.0 released March 1994
  - 487 source files, 165 KLOC, i386 only
- Linux kernel v2.3.39 released January 2000
  - 4854 source files, 2.2 MLOC, 10 hardware architectures supported, over 300 developers credited
- Maintained along two parallel paths:
  - *development* and *stable*

Development path: New and experimental features, relatively untested.

Stable path: Mostly bugfixes since last stable release, but also some fast-tracked new features.

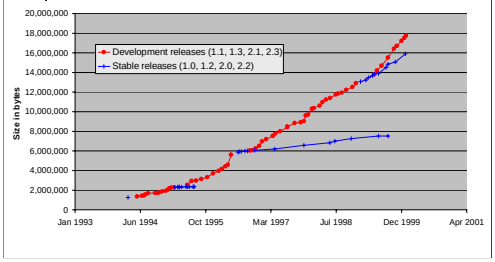
Methodology

- Examined 96 versions of Linux kernel
  - 34 of the 67 stable releases
  - 62 of the 369 development releases
- All measures considered only `.c/.h` files contained in the tarball
  - Counted LOC using "`wc -l`" and an `awk` script that ignored comments and blank lines
  - Counted # of fcns/vars/macros using `ctags`
  - Architectural model (SSs hierarchy) based on default directory structure
- We plotted growth against calendar time
  - Lehman suggests plotting growth against release number

Could ask, why not do a more elaborate SS hierarchy (based on ICSE paper) and look at change that way. (1) Lotta work without clear benefit and (2) it would be our peculiar view of Linux, and not the most common one.

Plotting against calendar time seemed to make sense to use, especially with two and sometimes three concurrent development paths. Lehman's argument for using release number is that it's the same as time according to his law of constant progress (which doesn't hold here!)

Growth of compressed tar file



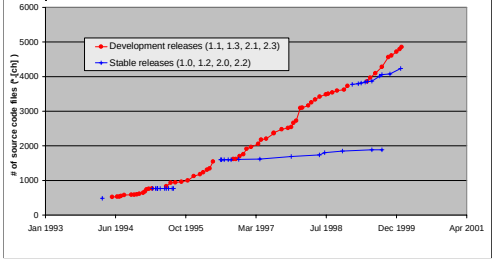
OK, this is the only measure that includes non-source files. A simple way of measuring size.

First problem is how to measure growth. There is no agreed-upon standard. Lehman says to use # modules, others use KLOC, or FPs, … so I measured several trends to see how they compare.

Strong growth of 2.2 stable release, mostly due to fast tracked contributed code. This is an example of how market forces and acts-of-God (IBM contributed code deus-ex-machina) can influence growth.

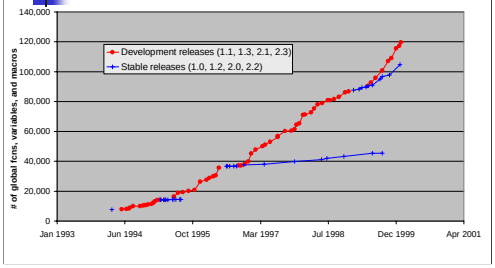
Note overlap of stable release paths 2.0 and 2.2

Growth of # of source files



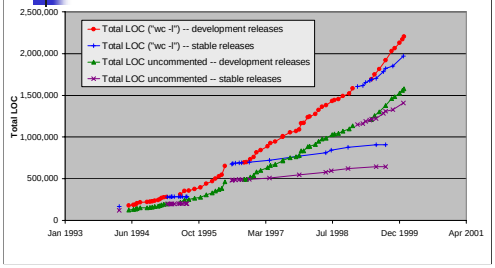
# of modules (source files in this case) is often used as a measure of growth, although you lose info such as avg/median file size, percent comments/blanks.

Growth of # of global fcns, variables, and macros

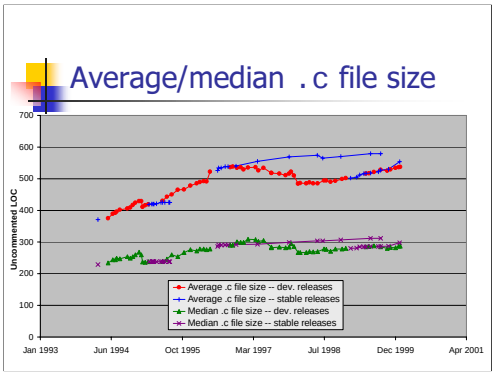


Measured using ctags; note that we are not counting local variables, just externally visible entities. In C, this is a pretty good measure of complexity since C has a single flat namespace (unlike C++ and Java).

Growth of Lines of Code (LOC)



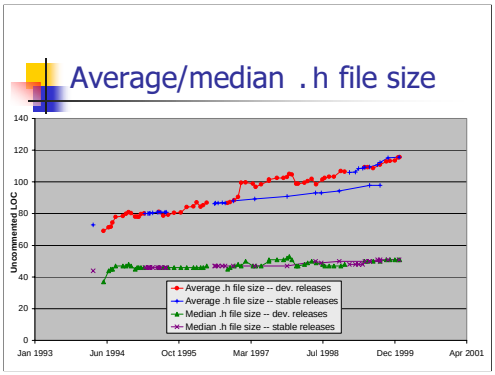
Percentage of comments and white space very stable throughout entire lifespan at 28-30%



Since growth trends seem similar to the naked eye, we tried dividing one by the other to see if we got a straight line. The measures that made the most sense were average and median file size for both .c and .h files.

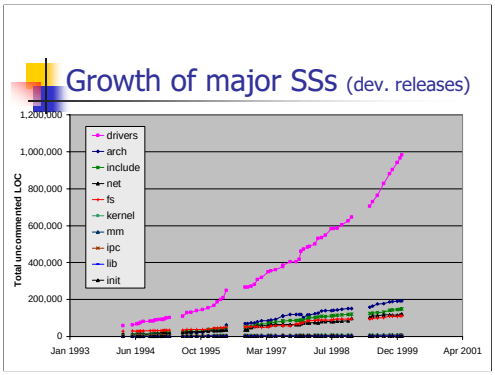
So what's happening here? First we note that the median .c file size has grown only slightly over time, whereas the average file size has grown non-trivially. This means that there are a growing number of large files rather than an overall growth in all files. We also note that over parallel paths 2.0 and 2.1, the average .c file size is growing in the stable release and actually shrinking in the development release. What seems to be happening is that the stable releases are encountering growth in (some of) the implementation files; that is, code is being added to some existing .c files, as you would expect for a stable release path.

With the development release 2.1, the average actually drops as many smaller files are added initially and then fleshed out. It should be noted that a lot of the growth is in a relatively small number of files: six of the eight largest .c files were SCSI drivers, which are KLOC++.



Here we again notice that median .h file size is remarkably flat. Here we notice the opposite growth pattern for average file size: development .h files increase in size more rapidly than stable releases along 2.0 and 2.1. This is what you would expect. .h files effectively are the interfaces, and you would hope that the interfaces would not change much along a stable path. The development release at 2.1 has a jump as infrastructure is laid out, then a more stable period as the structure stabilizes and the implementations are filled in.

Except that this is not quite accurate. Some of the growth is, as far as I can tell, just this kind of phenomenon. However, investigation showed that four of the six largest .h files were for network card devices. So, you hypothesize, SCSI card have complicated logic while network cards have complicated interfaces. Well, the former is accurate, but the bulk of the network card .h files was actually embedded data (hex). This is yet another place where I had to paused and wonder about how exactly I should be measuring growth.



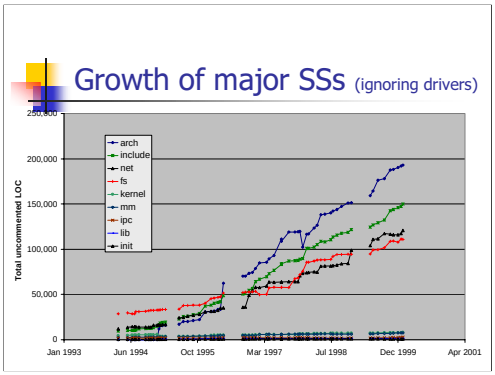
Now that we've seen growth at the system level, we delved into the major subsystems.

Also, from this point on, we consider only the development releases, as this is where the real action is. Each new stable release family starts at the end of a development release anyway (and then further evolution is mostly bugfixes).

Wow, that's interesting! And it was my first clue into why Linux can grow: it's mostly drivers.

We can think of there being three equivalence classes of major SSs: drivers, "big", and "small core".

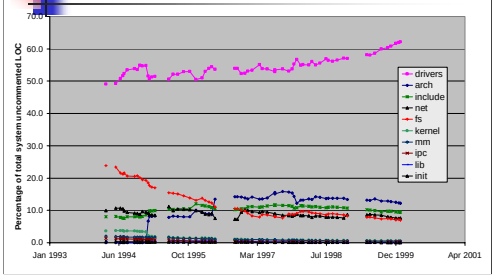
The strong growth of drivers masks the growth of the other SSs, so let's remove it from our picture.



We can see that the growth of the drivers does mask the significant growth in the other major SSs.

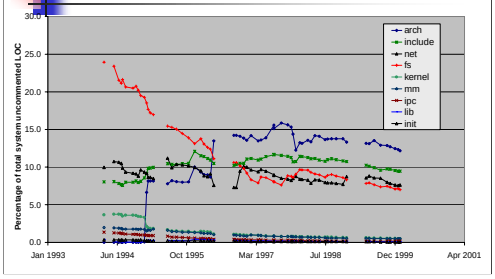
Include contains the .h files for all SSs except drivers, net, and fs (ie the core kernel .h files).

SS LOC as percentage of total system

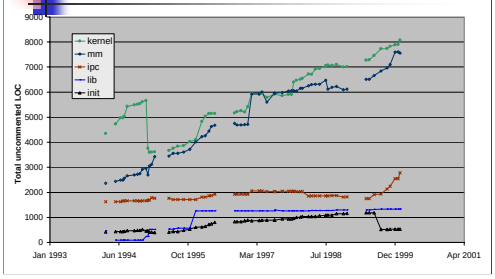


A horizontal line here indicates a SS that is growing at the same rate as the system as a whole.

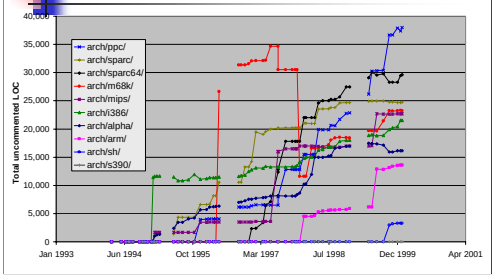
SS LOC as percentage of total system (ignoring drivers)



Growth of small core SSs



Growth of arch SSs

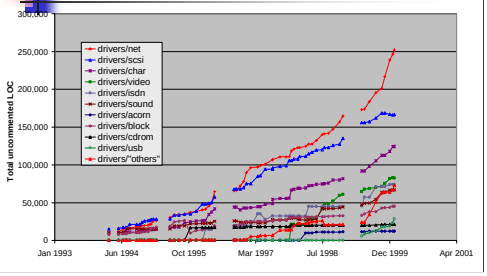


What's interesting here is the sudden jumps. This is because most of the parallel arch support is done by separate working groups (sometimes a corporation) and contributed in chunks. This is by far the most "bursty" growth of the major subsystems.

"sh" is a port to the SuperH family of microprocessors from Hitachi.

ARM is a chip used in a variety of (mainly British) computers including NCs and PDAs.

Growth of drivers SSs



Acorn is a British make of computer that uses the ARM chips (also found in a variety of other computers, including some handhelds).

My observation is that the growth in network and SCSI cards in particular is a sign of Linux's growth. Many of these drivers were developed by the device manufacturers and donated to the open source community. It should be noted that SCSI and network cards have fairly complicated logic.

Character devices are those devices that can be accessed simply as files: modems, sound cards, ttys. Block devices have read/writes chunked into blocks, which are managed as such; SCSI and IDE devices are accessed in this way. CDROM drivers are for devices that use proprietary interfaces and are relatively uncommon as most use IDE or SCSI connections now; note that this SS hasn't grown much.

Observations and hypotheses

- Growth along development path is super-linear

  y = .21*x^2 + 252*x + 90,055 r2=.997
  y = size in LOC x = days since v1.0
  r2 is "coefficient of determination" using least squares

  - Strong growth is continuing.
  - This is stronger growth than observed by others (Lehman, Gall), even for other OSs.

My hypothesis is that there is still a lot of room for growth: new drivers, file systems, architectures.

Also, the bandwagon is very popular right now. IBM, Dell and other heavyweights are strongly committed to Linux (including support).

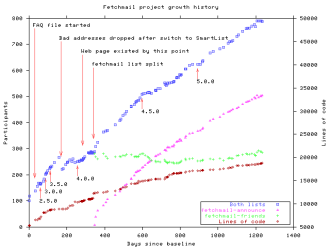
Why has Linux been able to continue its geometric growth?

- Core code quality is carefully maintained
- Architecture/problem domain
  - It's largely drivers
  - Much of the code is "parallel"
  - It's not as big as you might think
    - Vanilla configuration used only 15% of files
- Development model (OSD) and its sociology
  - Popularity and visibility has encouraged outsiders (both hackers and industry) to contribute

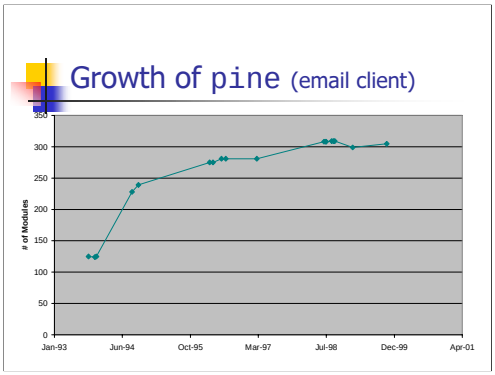
1. Drivers are independent of each other and the rest of the system (hard abstract interface).
2. Lots of "legacy drivers"

Hypothesis: significant cloning has occurred

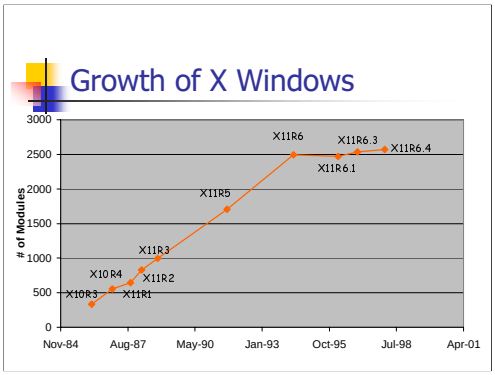
Growth of `fetchmail` [Raymond]



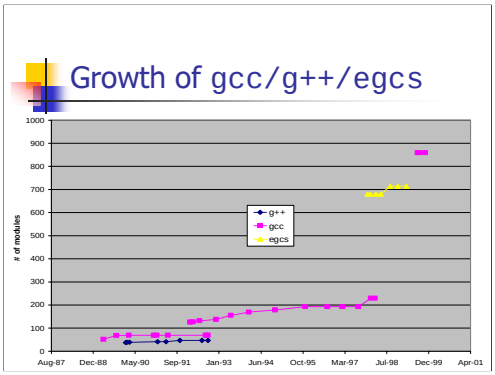
This comes from home page of Eric Raymond [Cathedral and the Bazaar]. The brown diamond tracks the code size. Raymond said that he expects growth to be slow as he considers the main development to be finished (ie only bug fixes from now on).



This was part of the work done by Qiang Tu. This is the raw data for the evolution of the pine email client. It does show the slowed growth. The rationale for the slowed growth here is that the main functionality is finished. However, I have not examined pine in any detail, so I am loathe to draw many conclusions.

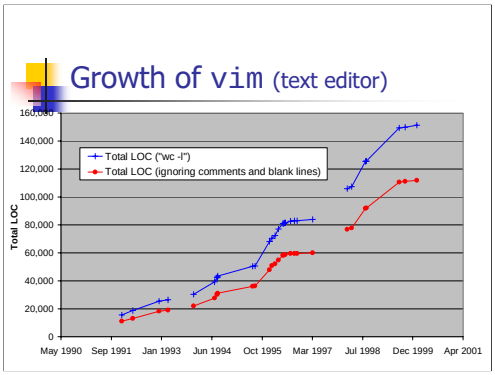


Here is X11. Since R6 was released, it's been in maintenance mode.



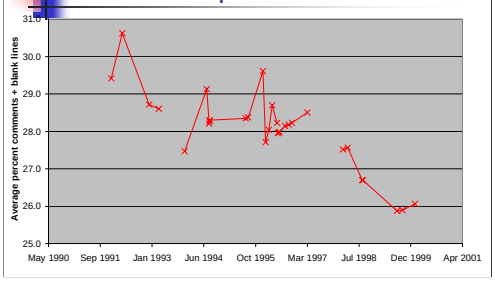
The gcc history is pretty interesting, and I only know some of it. The basic lesson is that non-technical issues have dominated here.

My Master's student Qiang Tu is examining the gcc history. Interesting to note that (a) gcc steering committee is distinct from developers and (b) most of the gcc developers are full-time employees of Cygnus. Gcc is the jewel in the crown of the GNU/OSD movement. It supports multiple front ends (different programming languages) and back ends (hw/OS platforms).



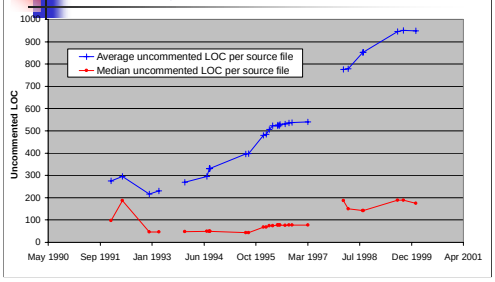
The vim text editor is another tool I have looked at in some detail. Like Linux I am a long-time user of vim going back to 1993 or so.

vim avg % comments and blank lines per file



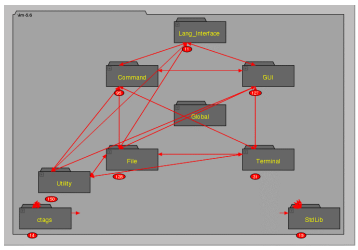
While this looks dramatic, it's not clear to me how significant the drop is (30% to 26%).

vim avg/median file size



This is more significant: The median size is staying fairly small, but the average size is increasing. The two largest source files are called <drum roll> misc1.c and misc2.c. Yikes! This leads me to wonder how long vim can continue to evolve. However, (a) it's an order of magnitude smaller than Linux and (b) it's mostly a central data structure with a few distinct subsystems/large scale features.

vim's architecture



Architecture model created by Eric Lee. VIM is basically a central data structure with a handful of major subsystems.

Hypotheses

Factors affecting evolution include

- Size and age of system
- Use of traditional sw. eng. principles during development

PLUS

- Problem domain
  - Problem complexity, multi-platform, multi-features
- Software architecture
- Process model
- Sociology, market forces, and acts-of-God

Gall said have to examine top level subsystems.

Software evolution research: What next?

- So far, have examined only growth of various aspects of code.
- We need:
  1. more detailed case studies
  2. supporting tools
  3. codified knowledge

Case studies (future work)

- Need to look at more systems:
  - Qualitative and quantitative studies
  - Industrial and open source systems
  - Different architectures, problem domains
    - OSs, telecom systems, compilers, …
- Examples:
  - More detailed analysis of Linux [Davor Svetinovic]
  - Linux *vs.* FreeBSD, Solaris
  - `gcc` *vs.* commercial compilers [Qiang Tu]

Reqs for a program evolution comprehension tool

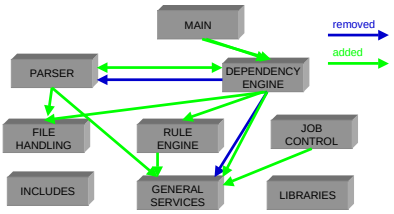
- Usual prog. comp. / reverse eng. tool requirements:
  - fast, reliable fact extractors
  - practical repository
  - visualization tools
  - interoperability (!)

Reqs for a program evolution comprehension tool

- Fast, incremental ocean boiling:
  - take advantage of "mostly the same"
  - precompute, use relational calculator (`grok`) when possible
- Usability analysis
  - Support for disposable views, experimentation, flexible usage, system "slicing"

Example: KAC and `gmake`

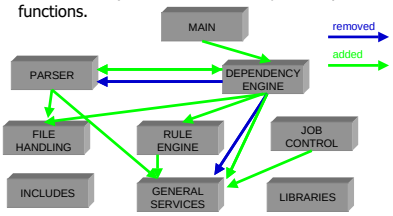
New feature — Support for determining `make` goals specified on command line.



KAC is a student project from my last seminar course (I gave them the idea and told them to go build it and experiment with it).

Example: KAC and `gmake`

Refactoring — Various functions (within Parser and General Services) were renamed or replaced by similar functions.



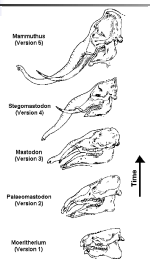
Tool future work

- So far, have assumed nodes are the same between graphs, only relationships change
  - not realistic
- Need to account for:
  - added / removed / preserved nodes
  - *changed* nodes and relationships
  - rearranged (different) containment trees
  - several versions at once
    - linear evolution *vs.* variants

Codified knowledge

- Mature engineering disciplines codify knowledge and experience.
- Arguably, this is lacking in software engineering.
  - Software architecture styles [Shaw]
  - Design patterns [GoF]
- Codified knowledge of how and why programs evolve:
  - *Evolutionary narratives* [Godfrey]
    - Long term, coarse granularity
  - *Change patterns*
    - Short term, fine granularity

Evolutionary narratives




```
% webster elephantine
1a: having enormous size
  or strength: MASSIVE
1b: CLUMSY, PONDEROUS
```


Change patterns and evolutionary narratives

- Cathedral style [Raymond]
  - careful control and management
  - debugging done before committing code
  - evolution is slow, planned, rarely undone
- Bazaar style (OSD)
  - lots of low-level changes, frequent fixes
  - lots of "building around" rather than wholesale changing, occasional redesigns
  - creeping feature-itis, "complete" dependency graph

Change patterns and evolutionary narratives

- Band-aid evolution (just add a layer)
  - quick & dirty way to add new functionality, esp. if system is not well understood
    e.g., Y2K fixing, adding portability, new features
- "Vestigial features"
  - design artifact persists after rationale dies
    e.g., whale fin bone structure resembles hand

Change patterns and evolutionary narratives

- "Adaptive radiation" [Lehman]
  - when conditions permit, encourage wild variation for a while.
  - later, evaluate and let "best" ideas live on.
    e.g., Linux kernel evolution
- "Convergent evolution"
  - compare similar systems to reference arch. (or to each other)
    e.g., everyone grows an XML generator in response to market pressure

Change patterns and evolutionary narratives

- Radical redesigns (localized and global)
  - aka "refactoring"
  - little new functionality added, but structure changes significantly, legacy cruft dissipates
  - likely "goodness" (design metrics) improves
- Migration patterns
  - look out for known translation idioms, especially if migration is not one big bang
    - *e.g.,* procedural-to-OO idioms

Change patterns and evolutionary narratives

- OO evolutionary patterns
  - one recognizable design pattern transformed into another (or a variation of the original)
    - requires good OO extraction tools (dynamic binding, polymorphism, reflection, *etc.*)
- Reuse patterns
  - components are (re)used in different systems
    - *e.g.,* build COTS interface, throw out homebrew DB

Change patterns and evolutionary narratives

- Phenomena observed in Linux evolution
  - Bandwagon effect
  - Contributed third party code
  - "Mostly parallel" enables sustained growth
  - Clone and hack
  - Careful control of core code; more flexibility on contributed drivers, experimental features

Summary of future research

- More case studies needed
  - Qualitative and quantitative
  - Industrial and open source systems
  - Different problem domains, architectures
- Supporting tools to aid analysing, visualizing, and querying program evolution
  - More than just `RCS` and `perl`
  - Support for architecture repair
- Why and how does software change?
  - Build catalogue of *change patterns* and *evolutionary narratives*

May not be of immediate use to practitioners …

… but some good foundational study may lead to better understanding of what software *is* and how it evolves.

(Fin)