Learning from history

How studying software evolution can make us wiser

Michael W. Godfrey
University of Waterloo

Joint work with

- Daniel Germán
- Cory Kapser
- Abram Hindle
- Ric Holt
- Qiang Tu

Overview

- What, exactly, is software evolution?
- Evolution in open source software
  – The Linux kernel
- Copy/paste as a principled engineering tool
- Learning from history

What, exactly, is software evolution?

And how does it differ from software maintenance?

The bluegill sunfish

- Female
- “Paternal” male
- “Cuckolder” male
  - Sneaker (age 2-3)
  - Satellite (age 4-5)

- An evolutionarily-stable strategy (ESS) … decided on *at run-time **

So …

- … to understand how a “thing” evolves, you must understand:
  - the thing and its programming,
  - its environment, and
  - how they can influence each other.

- … and “hard coding” can still lead to flexible, interesting run-time behaviours

Evolution vs. Maintenance

Maintenance

- “Keep it running”
- Active, engineering view:
  What ought be done and how?
- Study planned activities

Evolution

- Essential, design change
- Passive, scientific view:
  What happened and why?
- Study “whatever happens”
  e.g., unplanned phenomena such as interface bloat, emergent uses

Responding to evolutionary pressures

- Software is *expected* to evolve
  - Lehman’s first law: Adapt or die
  - Software doesn’t decay physically
    - Rather, the environment and our expectations change

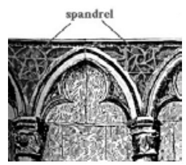
- “Intelligent design”
  - Parnas: Design for change
    - Info hiding, virtualize likely hotspots, design reviews
  - OO dev, frameworks, AOSD

… but you can’t anticipate everything
… and flexibility has a cost

Responding to evolutionary pressures

- Selection and adaptation
  - The deployment environment (users) "selects" individuals and features for success
  - Tho, unlike in biology, this can also be planned + evaluated

- Software systems often exhibit *emergent* properties
  (cf. "spandrels")
  e.g., vmware as farm management + malware tool
  XML as a DB
  IM as a debugger
  WWW as externalized memory



Why study software evolution?

- To improve understanding
  - Why is your system is designed as it is?
    c.f. the "temporal layers" architectural pattern
  - Quality assessment of third-party software
  - Challenge perceived truths

- To better anticipate change and reduce risk
  - Spot recurring problems, development bottlenecks
  - Better informed decision making by management

- Because we can :-)

Evolution in open source software

A case study of the Linux kernel

Lehman's Laws of Evolution

1. Continuing change — A system will become progressively less satisfying to its users over time, unless it is continually adapted to meet new needs.
2. Increasing complexity — A system will become progressively more complex, unless work is done to explicitly reduce the complexity.
3. Self-regulation — The process of software evolution is self regulating with respect to the distributions of the products and process artifacts that are produced.
4. Conservation of organizational stability — The average amount of work that goes into each release is about the same.
5. Conservation of familiarity — The amount of new content in each successive release of a system tends to stay constant or decrease over time.
6. Continuing growth — The amount of functionality in a system will increase over time, in order to please its users.
7. Declining quality — A system will be perceived as losing quality over time, unless its design is carefully maintained and adapted to new
8. Feedback system — Successfully evolving a software system requires recognition that the development process is a multi-loop, multi-agent, multi-level feedback system.

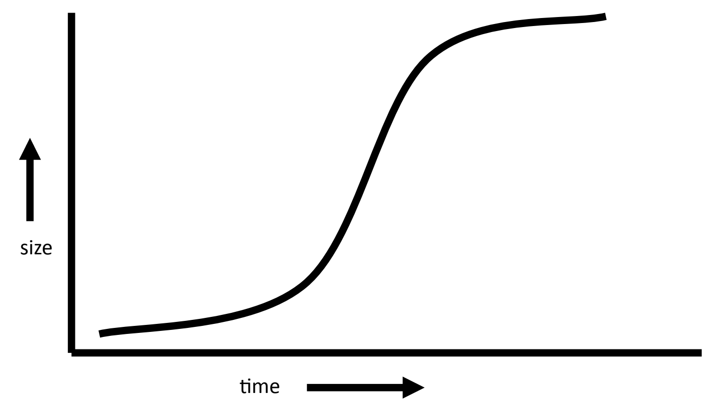
Lehman's Laws [in a nutshell]

- 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; growth is at best constant.

    Lehman/Turski's model: y’= y + E/y2 ~ (3Ex)1/3

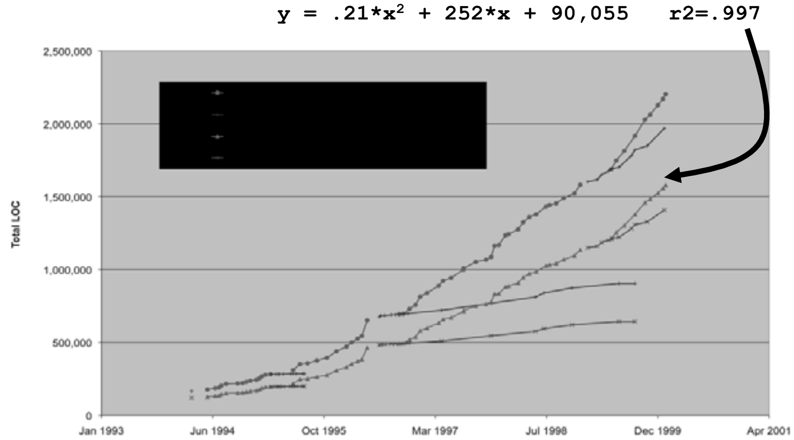
    where y= # of modules, x = release number
- Advice
  - Need to manage complexity.
  - Do periodic redesigns.
  - Treat software and its development process as a feedback system (and not as a passive theorem).

The S curve

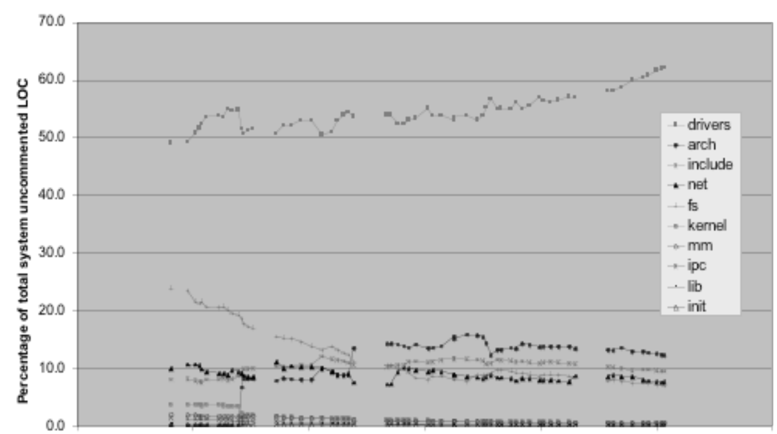


Growth of Lines of Code (LOC)

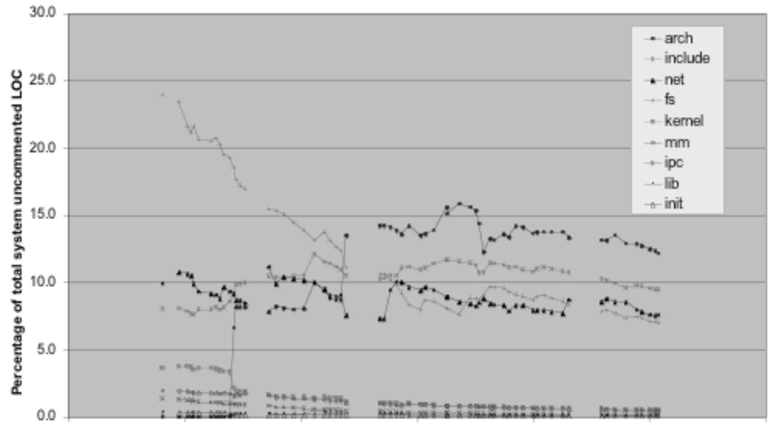
$y = .21*x^2 + 252*x + 90,055$ r2=.997



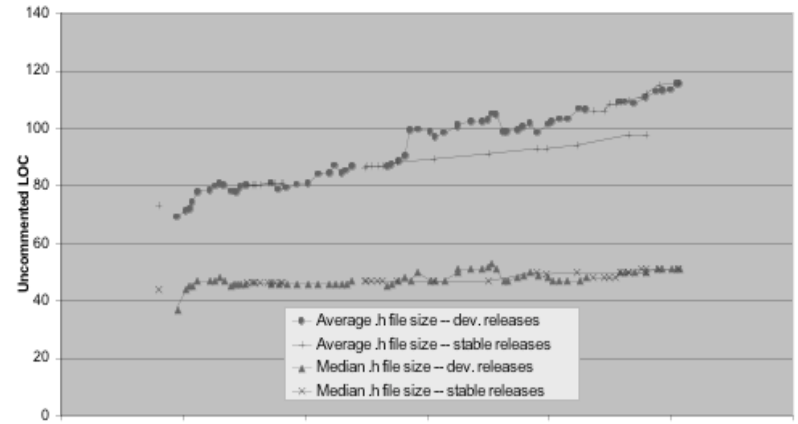
SS LOC as %age of total system



SS LOC as %age of total system



Average / median `.h` file size



Change patterns and evolutionary narratives

- Phenomena observed in Linux evolution
  - "Open" encourages participation, from industry too
  - Careful control of core code; more flexibility on contributed drivers, experimental features
  - "Mostly parallel" enables sustained growth
    - "Hard interfaces" make good neighbours.
    - Loadable modules makes feature development easier
  - "Clone and hack" makes sense!

Change patterns and evolutionary narratives

- "Band-aid evolution"
  - just add a layer, temporal architecture
- "Vestigial features"
- "Convergent evolution"
- "Adaptive radiation" [Lehman]
  - When conditions permit, encourage wild variation
  - Later: evaluate, prune, and let "best" ideas live on

Copy/paste as a principled engineering tool

Consider this code…

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
   return err;
}
ap_threads_per_child = atoi(arg);
if (ap_threads_per_child > thread_limit) {
   ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
            "WARNING: ThreadsPerChild of %d exceeds ThreadLimit "
            "value of %d", ap_threads_per_child,
            thread_limit);
   ….
   ap_threads_per_child = thread_limit;
}
else if (ap_threads_per_child < 1) {
   ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
            "WARNING: Require ThreadsPerChild > 0, setting to 1");
   ap_threads_per_child = 1;
}
return NULL;
```

and this code …

```
const char *err = ap_check_cmd_context(cmd, GLOBAL_ONLY);
if (err != NULL) {
   return err;
}
ap_threads_per_child = atoi(arg);
if (ap_threads_per_child > thread_limit) {
   ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
            "WARNING: ThreadsPerChild of %d exceeds ThreadLimit "
            "value of %d threads,", ap_threads_per_child,
            thread_limit);
   ….
   ap_threads_per_child = thread_limit;
}
else if (ap_threads_per_child < 1) {
          ap_log_error(APLOG_MARK, APLOG_STARTUP, 0, NULL,
            "WARNING: Require ThreadsPerChild > 0, setting to 1");
          ap_threads_per_child = 1;
}
return NULL;
```

… or these two functions

```
gnumeric_oct2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
    return val_to_base (ei, argv[0], argv[1],
      8, 2,
      0, GNM_const(7777777777.0),
      V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}

gnumeric_hex2bin (FunctionEvalInfo *ei, GnmValue const * const *argv)
{
    return val_to_base (ei, argv[0], argv[1],
      16, 2,
      0, GNM_const(9999999999.0),
      V2B_STRINGS_MAXLEN | V2B_STRINGS_BLANK_ZERO);
}
```

Or this …

```
static PyObject *
py_new_RangeRef_object (const GnmRangeRef *range_ref){
  py_RangeRef_object *self;
  self = PyObject_NEW py_RangeRef_object,
   &py_RangeRef_object_type);
  if (self == NULL) {
   return NULL;
  }
  self->range_ref = *range_ref;
  return (PyObject *) self;
}
```

… and this

```
static PyObject *
py_new_Range_object (GnmRange const *range) {
  py_Range_object *self;
  self = PyObject_NEW (py_Range_object,
  &py_Range_object_type);
  if (self == NULL) {
   return NULL;
  }
  self->range = *range;
  return (PyObject *) self;
}
```

What's in a clone?

- Cloning versus similarity
  - *"Software clones are segments of code that are similar according to some definition of similarity. "*
    – Ira Baxter, 2002
  - Hard to compare results!
- Bellon's taxonomy:
  Type 1: Program text identical; white space / comments may differ
  Type 2: … also literals + identifiers may be different
  Type 3: … gaps allowed (can add / delete sections)
  Type 4: Two code segments have same semantics

Code clone detection methods

- Strings
- Tokens
- ASTs
- PDGs

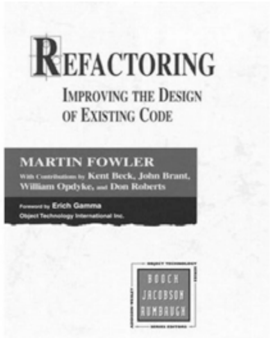
Time and complexity
/ prog lang dependence

- Metrics
- "Lightweight" semantics

Similar but different

- Problems related to software clone detection
  - Plagiarism detection, IP theft
  - DNA sequence analysis
  - Software compression
  - SPAM analysis, malware detection

Quotes on source code cloning



"Number one in the stink parade is duplicated code.

If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them."

– "Bad Smells" [Beck/Fowler in *Refactoring*]

Why cloning is supposed to be bad

- Code bloat
  - Design becomes harder to understand, less "essential"
- Inconsistent maintenance likely
- Ossified design, poor extensibility
  - Cruft accrues as developers fear changing working code
  - Need to keep doing same kinds of things, but there's no easy way to automate it

What you are supposed to do instead

- Identify commonalities across code base
- Refactor duplicate functionality to one place in the code:
  - Functions with parameters
  - Base class encapsulates commonalities, derived classes specialize peculiarities
  - Generics / templates for classes / functions

'Cloning considered harmful' … considered harmful*

1. Forking
   - Hardware variation
   - Platform variation
   - Experimental variation
2. Templating
   - Boilerplating
   - API / library protocols
   - Generalized programming idioms
   - Parameterized code
3. Customizing
   - Bug workarounds
   - Replicate + specialize

**Best paper at 2006 Working Conference on Reverse Engineering*

1. Forking

- Often used to "springboard" new or experimental development
  - Clones will need to evolve independently
  - Big chunks are copied!
- Works well when the commonalities and difference of the end solutions are unclear.

1. Forking: Platform variation

- Motivation:
  - Different platforms ⇒ very different low level details
  - Interleaving the platform-specific code in one place may be very complex
- Advantages of cloning:
  - Each (cloned) variant is simpler to maintain
  - No risk to stability of older variants
  - Platforms are likely to evolve independently, so maintenance is likely to be "mostly independent"

1. Forking: Platform variation

- Disadvantages:
  - Evolution in two dimensions: the user requirements and the support of the platform.
  - Change to the interface level means changes to many files
- Management and long-term issues:
  - Factor out platform independent functionality as much as possible
  - Document the variation points and platform peculiarities
  - As number of platforms grows, the interface to the system effectively hardens

1. Forking: Platform variation

- Structural manifestations:
  - Cloning usually happens at the file level.
    - Clones are often stored as files (or dirs) in the same source directory
- Well known examples:
  - Linux kernel “arch” subsystem
  - Apache Portable Runtime (APR)
    - Portable impl of functionality that is typically platform dependent, such as file and network access
    - **`fileio -> {netware, os2, unix, win32}`**
    - Typical changes: insertions of extra error checking or API calls.
    - Cloning is clearly obvious and is documented

2. Templating

- Code embodying the desired behavior already exists
  - … but the impl. language does not provide strong support for the desired abstraction
- Linked editing or source auto-generation can be used
- Examples
  - COBOL boilerplate code
  - C routines that treat floats and ints analogously
  - (old) Java code that could have used generics
  - API usages for common tasks (eg GUI creation)
  - Language / platform idioms, such as safe pointer handling

3. Customization

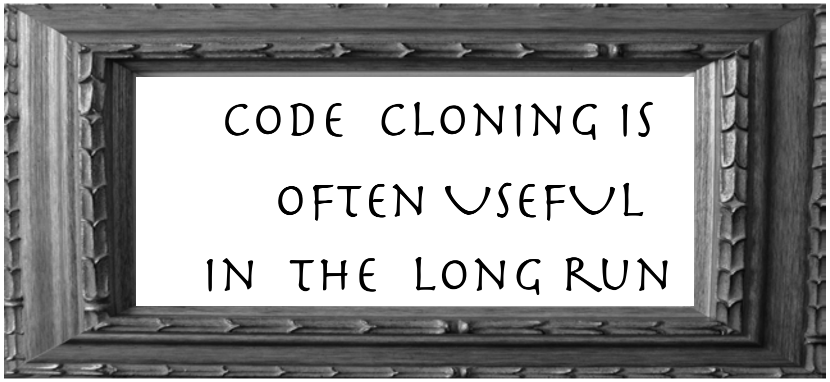
- Existing code solves a similar problem but you can’t or won’t change it
  - May not own the code [Microsoft: “Clone and own”]
  - May not want to risk change there
  - Changing may be too complex
- Examples:
  - Replicate and specialize
  - Bug workarounds

Two case studies

Group	Pattern	Good	Harmful	Good	Harmful
Forking	Hardware variation	0	0	0	0
Forking	Platform variation	10	0	0	0
Forking	Experimental variation	4	0	0	0
Templating	Boiler-plating	5	0	6	7
Templating	API	0	0	0	9
Templating	Idioms	0	12	1	1
Templating	Parameterized code	5	12	10	34
Customizing	Replicate + specialize	12	4	15	16
Customizing	Bug workarounds	0	0	0	0
Total		**36**	**28**	**32**	**67**

Apache httpd 2.2.4 - 60 Tokens
Gnumeric 1.6.3 - 60 Tokens

~~Myth~~
Motto



Learning from history

Summing up

The nature of software evolution

- Change is essential to software development
  [Brooks, Lehman]
- "Maintenance" + "evolution" connote different ideas
  - Maintenance: What should we do and how? (engineering)
  - Evolution: What happened and why? (science)
  - We need both views!
- To understand the whole picture of how software evolves, we need to study systems in context of use

What history taught me

- Study what you already have and understand
  - Take it apart and see how it works (e.g., Linux study)
- Challenge pre-conceived notions
  - Create testable hypotheses + evaluate them (e.g., cloning)
- Software archives contain lots of rich data
  - But need to process, link, mine the artifacts
- Need to continually re-examine reasonableness of assumptions
  - Don't blindly trust the numbers; dig and validate!

References

"The past, present, and future of software evolution"
by Michael W. Godfrey and Daniel Germán
Proc. of the Frontiers of Software Maintenance track at the 2008 IEEE Intl. Conf. on Software Maintenance, Beijing, China.

"`Cloning considered harmful' considered harmful"
by Cory J. Kapser and Michael W. Godfrey
Proc. of 2006 Working Conference on Reverse Engineering, Benevento, Italy. (Best paper award)

"Evolution in open source software: A case study"
by Michael W. Godfrey and Qiang Tu
Proc. of the 2000 IEEE Intl. Conf. on Software Maintenance, San Jose, CA.

Learning from history

How studying software evolution can make us wiser

Michael W. Godfrey
University of Waterloo