CSE584: Software Engineering

Lecture 6: Evolution (A)

David Notkin
Computer Science & Engineering
University of Washington
http://www.cs.washington.edu/education/courses/584/

Outline

- Software change
  - Basic background
  - Approaches to change
- Alternative approaches to maintenance
  - Introduction
  - Source models
  - Visualizing source models

Software evolution

- Software changes
  - Software maintenance
  - Software evolution
  - Incremental development
- The objective is to use an existing code base as an *asset*
  - Cheaper and better to get there from here, rather than starting from scratch
  - Anyway, where would you aim for with a new system? (We've discussed this a bit)

A legacy

- Merriam-Webster on-line dictionary
  - "a gift by will especially of money or other personal property"
  - "something transmitted by or received from an ancestor or predecessor or from the past"
- The usual joke is that in anything but software, you'd love to receive a legacy
  - Maybe we feel the same way about inheritance, too, especially multiple inheritance

Change

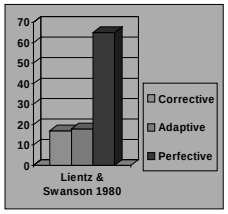
- "There is in the worst of fortune the best of chances for a happy change" —Euripides
- He who cannot dance will say, "The drum is bad" —Ashanti proverb
- "The ruling power within, when it is in its natural state, is so related to outer circumstances that it easily changes to accord with what can be done and what is given it to do" —Marcus Aurelius
- "Change in all things is sweet" —Aristotle

Why does it change?

- Software changes does not change primarily because it doesn't work right
  - Maintenance in software is different than maintenance for automobiles
- But rather because the technological, economic, and societal environment in which it is embedded changes
- This provides a feedback loop to the software
  - The software is usually the most malleable link in the chain, hence it tends to change
    - Counterexample: Space shuttle astronauts have thousands of extra responsibilities because it's safer than changing code

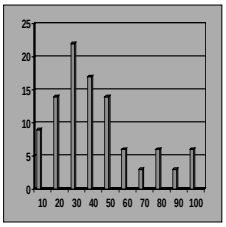
Kinds of change

- Corrective maintenance
  - Fixing bugs in released code
- Adaptive maintenance
  - Porting to new hardware or software platform
- Perfective maintenance
  - Providing new functions
- Old data, focused on IT systems…now?



High cost, long time

- Gold's 1973 study showed the fraction of programming effort spent in maintenance
- For example, 22% of the organizations spent 30% of their effort in maintenance



Total life cycle cost

- Lientz and Swanson determined that at least 50% of the total life cycle cost is in maintenance
- There are several other studies that are reasonably consistent
- General belief is that maintenance accounts for somewhere between 50-75% of total life cycle costs

Open question

- How much maintenance cost is "reasonable?"
  - Corrective maintenance costs are ostensibly not "reasonable"
  - How much adaptive maintenance cost is "reasonable"?
  - How much perfective maintenance cost is "reasonable"?
- Measuring "reasonable" costs in terms of percentage of life cycle costs doesn't make sense

High-level answer

- For perfective maintenance, the objective should be for the cost of the change in the implementation to be proportional to the cost of the change in the specification (design)
  - Ex: Allowing dates for the year 2000 is (at most) a small specification change
  - Ex: Adding call forwarding is a more complicated specification change
  - Ex: Converting a compiler into an ATM machine is …

Aside: reuse

- There's been some discussion of reuse on the mailing list
- I. Jacobson, M. Griss, P. Jonsson. *Software Reuse : Architecture Process and Organization for Business Success* (1997)
  - Among other things, they argue that reuse cannot be done effectively bottom-up: it requires a change in culture, reward structures, and goals
- Patterns vs. reuse: my view is that they are both attempting to increase productivity, but in different (albeit somewhat related) ways
  - Reuse: taking advantage of assets
  - Patterns: raising the notational/thought level

(Common) Observations

- Maintainers often get less respect than developers
- Maintenance is generally assigned to the least experienced programmers
- Software structure degrades over time
- Documentation is often poor and is often inconsistent with the code
- Is there any relationship between these?

Laws of Program Evolution

Program Evolution: Processes of Software Change
(Lehman & Belady)

- Law of continuing change
- "A large program that is used undergoes continuing change or becomes progressively less useful."
  - Analogies to biological evolution have been made; the rate of change in software is far faster
- P-type programs
  - Well-defined, precisely specified
  - The challenge is efficient implementation
  - Ex: sort
- E-type programs
  - Ill-defined, fit into an ever-changing environment
  - The challenge is managing change
- Also, S-type programs
  - Ex: chess

Law of increasing complexity

- "As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it."
  - Complexity, in part, is relative to a programmer's knowledge of a system
    - Novices vs. experts doing maintenance
  - Cleaning up structure is done relatively infrequently
    - Even with the recent interest in refactoring, this seems true. Why?

Reprise

- The claim is that if you measure any reasonable metric of the system
  - Modules modified, modules created, modules handled, subsystems modified, …
- and then plot those against time (or releases)
- Then you get highly similar curves regardless of the actual software system
- A zillion graphs on http://www.doc.ic.ac.uk/~mml/feast1/

Statistically regular growth

- "Measures of [growth] are cyclically self-regulating with statistically determinable trends and invariances."
  - (You can run but you can't hide)
    - There's a feedback loop
  - Based on data from OS/360 and some other systems
  - Ex: Content in releases decreases, or time between releases increases
- Is this related to Brooks' observation that adding people to a late project makes it later?

And two others

- "The global activity rate in a large programming project is invariant."
- "For reliable, planned evolution, a large program undergoing change must be made available for regular user execution at maximum intervals determined by its net growth."
  - This is related to "daily builds"

Open question

- Are these "laws" of Belady and Lehman actually inviolable laws?
- Could they be overcome with tools, education, discipline, etc.?
- Could their constants be fundamentally improved to give significant improvements in productivity?
  - Recently Greenspan and others have claimed that IT has fundamentally changed the productivity of the economy: "The synergistic effect of new technology is an important factor underlying improvements in productivity."

Approaches to reducing cost

- Design for change (proactive)
  - Information hiding, layering, open implementation, aspect-oriented programming, etc.
- Tools to support change (reactive)
  - grep, etc.
  - Reverse engineering, program

Approaches to reducing cost

- Improved documentation (proactive)
  - Discipline, stylized approaches
  - Parnas is pushing this *very* hard, using a tabular form of specifications
  - Literate programming
- Reducing bugs (proactive)
  - Many techniques, some covered later in the quarter
- Increasing correctness of specifications (proactive)
- Others?

Program understand & comprehension

- Definition: The task of building *mental models* of the underlying software at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for maintenance, evolution, and re-engineering purposes [H. Müller]

Various strategies

- Top-down
  - Try to map from the application domain to the code
- Bottom-up
  - Try to map from the code to the application domain
- Opportunistic: mix of top-down and bottom-up
- I'm not a fan of these distinctions, since it has to be opportunistic in practice
  - Perhaps with a *really* rare exception

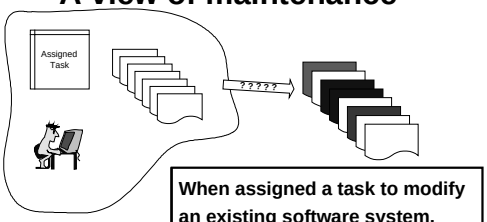
Did you try to understand?

- "The ultimate goal of research in program understanding is to improve the process of comprehending programs, whether by improving documentation, designing better programming languages, or building automated support tools." —Clayton, Rugaber, Wills
- To me, this definition (and many, many similar ones) miss a key point: What is the programmer's task?
- Furthermore, most good programmers seem to be good at knowing what they need to know *and what they don't need to know*

A scenario

- I'm about to walk through a simple scenario or two
- The goal isn't to show you "how" to evolve software
- Rather, the goal is to try to increase some of the ways in which you think during software evolution

A view of maintenance



When assigned a task to modify an existing software system, how does a software engineer choose to proceed?

Sample (simple) task

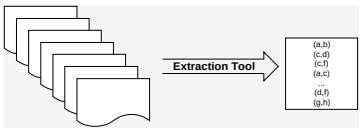
- You are asked to update an application in response to a change in a library function
- The original library function is
  - `assign(char* to, char* from, int cnt = NCNT)`
  - Copy `cnt` characters from `to` into `from`
- The new library function is
  - `assign(char* to, char* from, int pos, int cnt = NCNT)`
  - Copy `cnt` characters starting at `pos` from `to` into `from`
- How would you make this change? (In groups)

Recap: example

- What information did you need?
- What information was available?
- What tools produced the information?
  - Did you think about other pertinent tools?
- How accurate was the information?
  - Any false information? Any missing true information?
- How did you view and use the information?
- Can you imagine other useful tools?

Source models

- Reasoning about a maintenance task is often done in terms of a model of the source code
  - Smaller than the source, more focused than the source
- Such a *source model* captures one or more relations found in the system's artifacts



Example source models

- A calls graph
  - Which functions call which other functions?
- An inheritance hierarchy
  - Which classes inherit from which other classes?
- A global variable cross-reference
  - Which functions reference which globals?
- A lexical-match model
  - Which source lines contain a given string?
- A def-use model
  - Which variable definitions are used at which use sites?

Combining source models

- Source models may be produced by combining other source models using simple relational operations; for example,
  - Extract a source model indicating which functions reference which global variables
  - Extract a source model indicating which functions appear in which modules
  - Join these two source models to produce a source model of modules referencing globals

Extracting source models

- Source models are extracted using tools
- Any source model can be extracted in multiple ways
  - That is, more than one tool can produce a given kind of source model
- The tools are sometimes off-the-shelf, sometimes hand-crafted, sometimes customized

Program databases

- There are many projects in which a program database is built, representing source models of a program
- They vary in many significant ways
  - The data model used (relational, object-oriented)
  - The granularity of information
    - Per procedure, per statement, etc.
  - Support for creating new source models
    - Operations on the database, entirely new ones
  - Programming languages supported

Three classic examples

- CIA/CIA++, ATT Research (Chen et al.)
  - Relational, C/C++
  - http://www.research.att.com/sw/tools/reuse/
  - CIAO, a web-based front-end for program database access
- Desert, Brown University (Reiss)
  - Uses Fragment integration
    - Preserves original files, with references into them
  - http://www.cs.brown.edu/software/desert/
  - Uses FrameMaker as the editing/viewing engine
- Rigi (support for reverse engineering)
  - http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml

Information characteristics

	no false positives	*false positives*
no false negatives	ideal	conservative
false negatives	optimistic	approximate

Ideal source models

- It would be best if every source model extracted was perfect
  - All entries are true and no true entries are omitted
- For some source models, this is possible
  - Inheritance, defined functions, `#include` structure, etc.
- For some source models, achieving the ideal may be difficult in practice
  - Ex: computational time is prohibitive in practice
- For many other interesting source models, this is not possible
  - Ideal call graphs, for example, are uncomputable

Conservative source models

- These include all true information and maybe some false information, too
- Frequently used in compiler optimization, parallelization, in programming language type inference, etc.
  - Ex: never misidentify a call that can be made or else a compiler may translate improperly
  - Ex: never misidentify an expression in a statically typed programming language

Optimistic source models

- These include only truth but may omit some true information
- Often come from dynamic extraction
- Ex: In white-box code coverage in testing
  - Indicating which statements have been executed by the selected test cases
  - Others statements may be executable with other test cases

Approximate source models

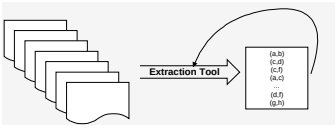
- May include some false information and may omit some true information
- These source models can be useful for maintenance tasks
  - Especially useful when a human engineer is using the source model, since humans deal well with approximation
  - It's "just like the web!"
- Turns out many tools produce approximate source models (more on this later)

Static vs. dynamic

- Source model extractors can work
  - *statically*, directly on the system's artifacts, or
  - *dynamically*, on the execution of the system, or
  - a combination of both
- Ex:
  - A call graph can be extracted statically by analyzing the system's source code or can be extracted dynamically by profiling the system's execution

Must iterate

- Usually, the engineer must iterate to get a source model that is "good enough" for the assigned task
- Often done by inspecting extracted source models and refining extraction tools
- May add and combine source models, too



Another maintenance task

- Given a software system, rename a given variable throughout the system
  - Ex: `angle` should become `diffraction`
  - Probably in preparation for a larger task
- Semantics must be preserved
- This is a task that is done infrequently
  - Without it, the software structure degrades more and more

What source model?

- Our preferred source model for the task would be a list of lines (probably organized by file) that reference the variable `angle`
- A static extraction tool makes the most sense
  - Dynamic references aren't especially pertinent for this task

Start by searching

- Let's start with grep, the most likely tool for extracting the desired source model
- The most obvious thing to do is to search for the old identifier in all of the system's files
  - `grep angle *`

What files to search?

- It's hard to determine which files to search
  - Multiple and recursive directory structures
  - Many types of files
    - Object code? Documentation? (ASCII vs. non-ASCII?) Files generated by other programs (such as yacc)? Makefiles?
  - Conditional compilation? Other problems?
- Care must be taken to avoid false negatives arising from files that are missing

False positives

- `grep angle` *`[system's files]`*
- There are likely to be a number of spurious matches
  - `…triangle…, …quadrangle…`
  - `/* I could strangle this programmer! */`
  - `/* Supports the small planetary rovers presented by Angle & Brooks (IROS '90) */`
  - `printf("Now play the Star Spangled Banner");`
- Be careful about using `agrep`!

More false negatives

- Some languages allow identifiers to be split across line boundaries
  - Cobol, Fortran, PL/I, etc.
  - This leads to potential false negatives
- Preprocessing can hurt, too

```
#define deflection angle
...
deflection = sin(theta);
```

It's not just syntax

- It is also important to check, before applying the change, that the new variable name (`degree`) is not in conflict anywhere in the program
  - The problems in searching apply here, too
  - Nested scopes introduce additional complications

Tools vs. task

- In this case, `grep` is a lexical tool but the renaming task is a semantic one
  - Mismatch with syntactic tools, too
- Mismatches are common and not at all unreasonable
  - But it does introduce added obligations on the maintenance engineer
  - Must be especially careful in extracting and then using the approximate source model

Finding vs. updating

- Even after you have extracted a source model that identifies all of (or most of) the lines that need to be changed, you have to change them
- Global replacement of strings is at best dangerous
- Manually walking through each site is time-consuming, tedious, and error-prone

Downstream consequences

- After extracting a good source model by iterating, the engineer can apply the renaming to the identified lines of code
- However, since the source model is approximate, regression testing (and/or other testing regimens) should be applied

An alternative approach

- Griswold developed a meaning-preserving program restructuring tool that can help
- For a limited set of transformations, the engineer applies a local change and the tool applies global compensating changes that maintain the program's meaning
  - Or else the change is not applied
  - Reduces errors and tedium when successful

But

- The tool requires significant infrastructure
  - Abstract syntax trees, control flow graphs, program dependence graphs, etc.
- The technology OK for small programs
  - Downstream testing isn't needed
  - No searching is needed
- But it does not scale directly in terms of either computation size or space

Recap

- "There is more than one way to skin a cat"
  - Even when it's a tiger
- The engineer must decide on a source model needed to support a selected approach
- The engineer must be aware of the kind of source model extracted by the tools at hand
- The engineer must iterate the source model as needed for the given task
- Even if this is not conscious nor explicit

Build up idioms

- Handling each task independently is hard
- You can build up some more common idiomatic approaches
  - Some tasks, perhaps renaming, are often part of larger tasks and may apply frequently
  - Also internalize source models, tools, etc. and what they are (and are not) good at
- But don't constrain yourself to only what your usual tools are good for

Source model accuracy

- This is important for programmers to understand
- Little focus is given to the issue

Call graph extraction tools (C)

- Two basic categories: lexical or syntactic
  - lexical
    - e.g., awk, mkfunctmap, lexical source model extraction (LSME)
    - \- likely produce an approximate source model
    - \+ extract calls across configurations
    - \+ can extract even if we can't compile
    - \+ typically fast

... tools (C)

- Two basic categories: lexical or syntactic...
  - syntactic
    - e.g., CIA, Field, cflow, rigiparse, etc.
    - \+ more likely to produce conservative information than a lexically-based tool
    - \- have to pick a configuration
    - \- need to get the source to a parseable state

Apply a syntactic CGE tool

- C Information Abstractor (CIA)
  - extracts references between functions
- Constraints:
  - specific configuration, libraries, etc.
- Queries:
  - `cref func - func socket`

```
HTFTP.c  get_listen_socket -> <libc.a> socket
HTTCP.c  HTDoConnect       -> <libc.a> socket
accept.c NetServerInit     -> <libc.a> socket
```

- Can dump the entire source model

How precise?

- Are the source models extracted by CIA conservative?
- It is typically difficult to determine the answer to this kind of question
- But, to perform a task confidently, you need to get a handle on the precision
  - maybe by reading the tool's documentation
  - maybe by comparison to other tools
  - maybe by...?

A CGE experiment

- To investigate several call graph extractors for C, we ran a simple experiment
  - For several applications, extract call graphs using several extractors
  - Applications: mapmaker, mosaic, gcc
  - Extractors: CIA, rigiparse, Field, cflow, mkfunctmap

Experimental results

- Quantitative
  - pairwise comparisons between the extracted call graphs
- Qualitative
  - sampling of discrepancies
- Analysis
  - what can we learn about call graph extractors (especially, the design space)?

Pairwise comparison (example)

- CIA vs. Field for Mosaic (4258 calls reported)
  - CIA found about 89% of the calls that Field found
  - Field did not find about 5% of the references CIA found
  - CIA did not find about 12% of the calls Field found

Quantitative Results

- No two tools extracted the same calls for any of the three programs
- In several cases, tools extracted large sets of non-overlapping calls
- For each program, the extractor that found the most calls varied (but remember, more isn't necessarily better)
- Can't determine the relationship to the ideal

Qualitative results

- Sampled elements to identify false positives and false negatives
- Mapped the tuples back to the source code and performed manual analysis by inspection
- Every extractor produced some false positives and some false negatives

Call graph characterization

	no false positives	*false positives*
no false negatives	ideal *none*	conservative *compilers*
false negatives	optimistic *profilers*	approximate *software engineering tools*

Next week

- **Software reflexion models**
  - **Software summarization, task-based approach**
- **Rigi (and perhaps some other clustering approaches)**
- **Miscellaneous**