Four “interesting” ways in which history can teach us about software

Michael W. Godfrey *
Xinyi Dong
Cory Kapser
Lijie Zou

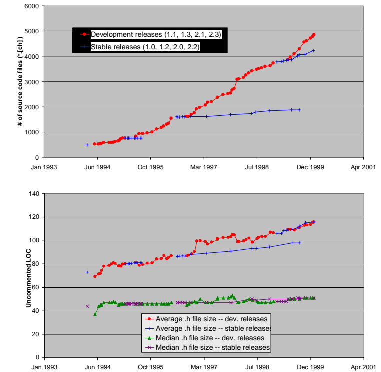
University of Waterloo

Software Architecture Group (SWAG)
University of Waterloo

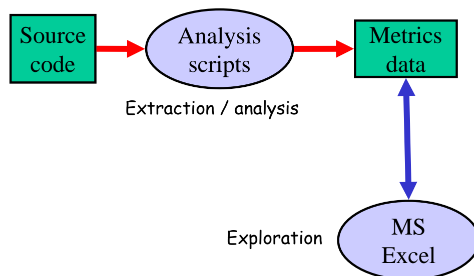
****Currently on sabbatical at Sun Microsystems***

1. Longitudinal case studies of growth and evolution

- Studied several OSSs, esp. Linux kernel:
  - Looked for “evolutionary narratives” to explain observable historical phenomena
- Methodology:
  - Analyze individual tarball versions
  - Build hierarchical metrics data model
  - Generate graphs, look for interesting lumps under the carpet, try to answer why

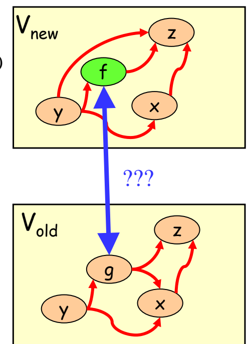


1. Longitudinal case studies of growth and evolution

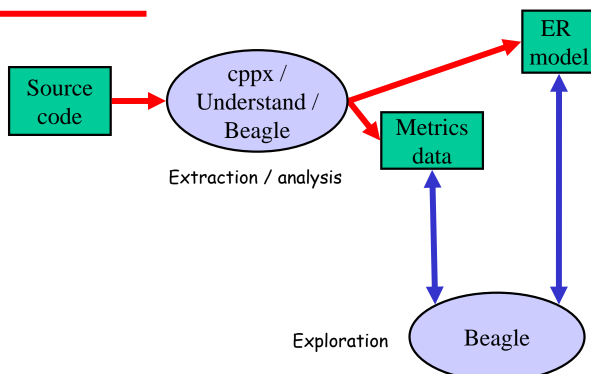


2. Case studies of *origin analysis*

- Reasoning about structural change
  - (moving, renaming, merging, splitting, *etc.*)
  - Try to reconstruct what happened
  - Formalized several “change patterns”
    - *e.g.,* service consolidation
- Methodology:
  - Consider consecutive pairs of versions:
    - *Entity analysis* – metrics-based clone detection
    - *Relationship analysis* – compare relational images (calls, called-by, uses, extends, *etc*)
  - Create evolutionary record of what happened
    - what evolved from what, and how/why



2. Case studies of *origin analysis*



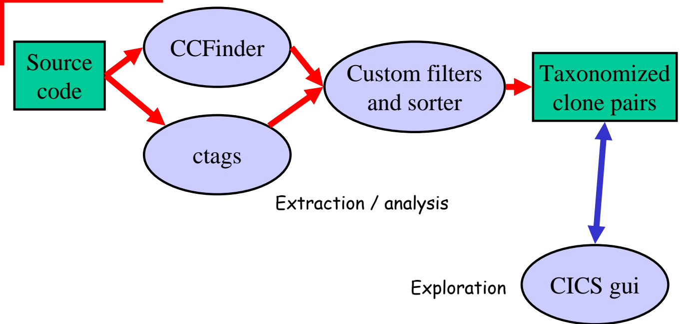
3. Case studies of code cloning

- Motivation:
  - Lots of research in clone detection, but more on algorithms and tools than on case studies and comprehension
    - What kinds of cloning are there? Why does cloning happen? What kinds are the most/least harmful? Do different clone kinds have different precision / recall numbers? Different algorithms?
  - Future work: track clone evolution
    - Do related bugs get fixed? Does cloned code have more bugs?
- Methodology:
  1. Use `CCFinder` on source to find initial clone pairs.
  2. Use `ctags` to map out source files into “entity regions”
     - *Consecutive* typedefs, fcn prototypes, var defs
     - *Individual* macros, structs, unions, enums, fcn defs
  3. Map (abstract up) clone pairs to the source code regions

3. Case studies of code cloning

- Methodology:
  4. Filter different region kinds according to observed heuristics
     - C `struct`s often look alike; parameterized string matching returns many more false positives without these filters than, say, between functions.
  5. Sort clones by location:
     - Same region, same file, same directory, or different directory
  6. … and entity kind:
     - Fcn to fcn
     - structures (`enum`, `union`, `struct`)
     - macro
     - heterogeneous (different region kinds)
     - misc. clones
  7. … and even more detailed criteria:
     - Function initialization / finalization clones, …
  8. Navigate and investigate using CICS gui, look for patterns
     - Cross subsystem clones seems to vary more over time
     - Intra subsystem clones are usually function clones

3. Case studies of code cloning



4. Longitudinal case studies of software manufacturing-related artifacts

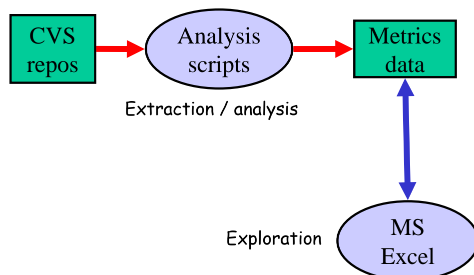
Q: How much maintenance effort is put into SM artifacts, relative to the system as a whole?

- Studying six OSSs:
  - **`GCC, PostgreSQL, kepler, ant, mycore, midworld`**
    - All used CVS; we examined their logs
    - We look for SM artifacts (`Makefile`, `build.xml`, `SConscript`) and compared them to non-SM artifacts

4. Longitudinal case studies of software manufacturing-related artifacts

- Some results:
  - Between 58 and 81 % of the core developers contributed changes to SM artifacts
  - SM artifacts were responsible for
    - 3-10% of the number of changes made
    - Up to 20% of the total LOC changed (GCC)
- Open questions:
  - How difficult is it to maintain these artifacts?
  - Do different SM tools require different amounts of effort?

4. Longitudinal case studies of software manufacturing-related artifacts



Dimensions of studies

- Single version *vs*. consecutive version pairs *vs*. longitudinal study
- Coarsely *vs*. finely grained detail
- Intermediate representation of artifacts:
  - Raw code *vs*. metrics *vs*. ER-like semantic model
  - Navigable representation of system architecture; auto-abstraction of info at arbitrary levels

Challenges in this field

1. Dealing with scale
   - "Big system analysis" times "many versions"
   - Research tools often live at bleeding edge, slow and produce voluminous detail

2. Automation
   - Research tools often buggy, require handholding
     - Often, hard to get automated multiple analyses.

Challenges in this field

3. Artifact linkage and analysis granularity
   - Repositories (CVS, Unix fs) often store only source code, with no special understanding of, say, where a particular method resides.
   - (How) should we make them smarter?
     - *e.g.,* **`ctags`** and **`CCfinder`**

4. [Your thoughts?]