CSE P503: Principles of Software Engineering

David Notkin
Autumn 2007

Miscellaneous & Mining Software Repositories

Experience is that which enables us to recognize our mistakes when we make them again. --AM51, 1973

[A]ny fool can make history, but it takes a genius to write it. --Oscar Wilde

Technology is dominated by two types of people: those who understand what they do not manage, and those who manage what they do not understand. --Putt's Law

Agenda

- Some left over material (not all, but some), all quickly
  - Restructuring and star diagrams
  - SeeSoft
- Interludes
  - Education
  - Testing configurations?
- Mining Software Repositories

Restructuring

- Why don’t people restructure as much as we’d like…?
  - Doesn’t make money now
  - Introduces new bugs
  - Decreases understanding
  - Political pressures
  - Who wants to do it?
  - Hard to predict lifetime costs & benefits

Griswold's 1st approach

- Griswold developed an approach to meaning-preserving restructuring
- Make a local change
  - The tool finds global, compensating changes that ensure that the meaning of the program is preserved
    - What does it mean for two programs to have the same meaning?
  - If it cannot find these, it aborts the local change

Simple example

- Swap order of formal parameters

```
procedure push(s, v)
   insert(v, s.head)
   return s
end
  .
  .
  .
push(myStack,1)
  .
  .
  .
push(myStack,h(myStack))
```

- It's not a local change nor a syntactic change
- It requires semantic knowledge about the programming language
- Griswold uses a variant of the sequence-congruence theorem [Yang] for equivalence
  - Based on PDGs (program dependence graphs)
- It's an O(1) tool

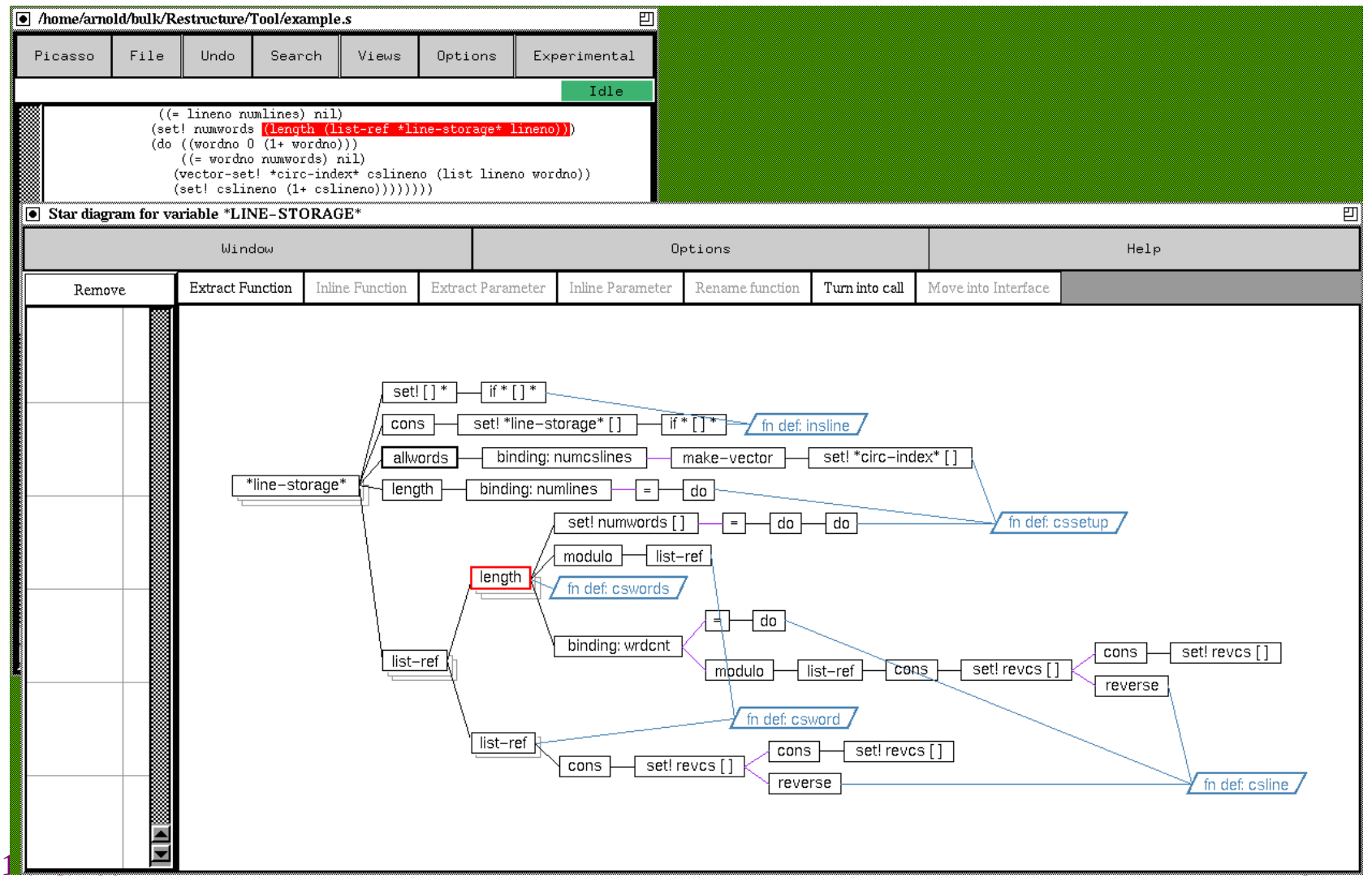
Limited power

- The actual tool and approach has limited power
- Too limited to be useful in practice
  - PDGs are limiting
    - Big and expensive to manipulate
    - Difficult to handle in the face of multiple files, etc.
- May encourage systematic restructuring in some cases

Star diagrams [Griswold et al.]

- Meaning-preserving restructuring isn't going to work on a large scale
- But sometimes significant restructuring is still desirable
- Instead provide a tool (star diagrams) to
  - record restructuring plans
  - hide unnecessary details
- Some modest studies on programs of 20-70KLOC

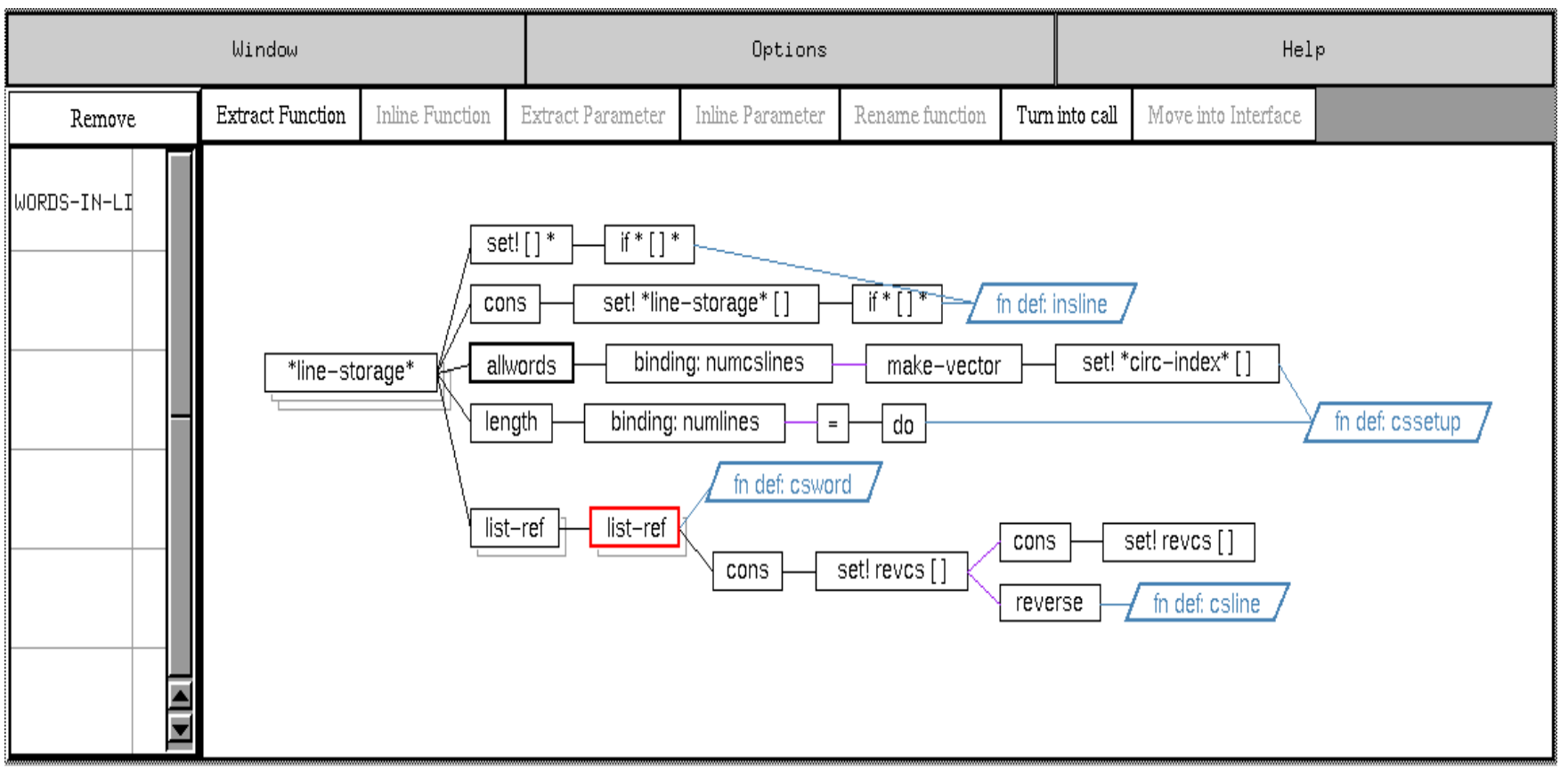
A star diagram



Interpreting a star diagram

- The root (far left) represents all the instances of the variable to be encapsulated
- The children of a node represent the operations and declarations directly referencing that variable
- Stacked nodes indicate that two or more pieces of code correspond to (perhaps) the same computation
- The children in the last level (parallelograms) represent the functions that contain these computations

After some changes



Evaluation

- Compared small teams of programmers on small programs
  - Used a variety of techniques, including videotape
  - Compared to vi/grep/etc.
- Nothing conclusive, but some interesting observations including
  - The teams with standard tools adopted more complicated strategies for handling completeness and consistency

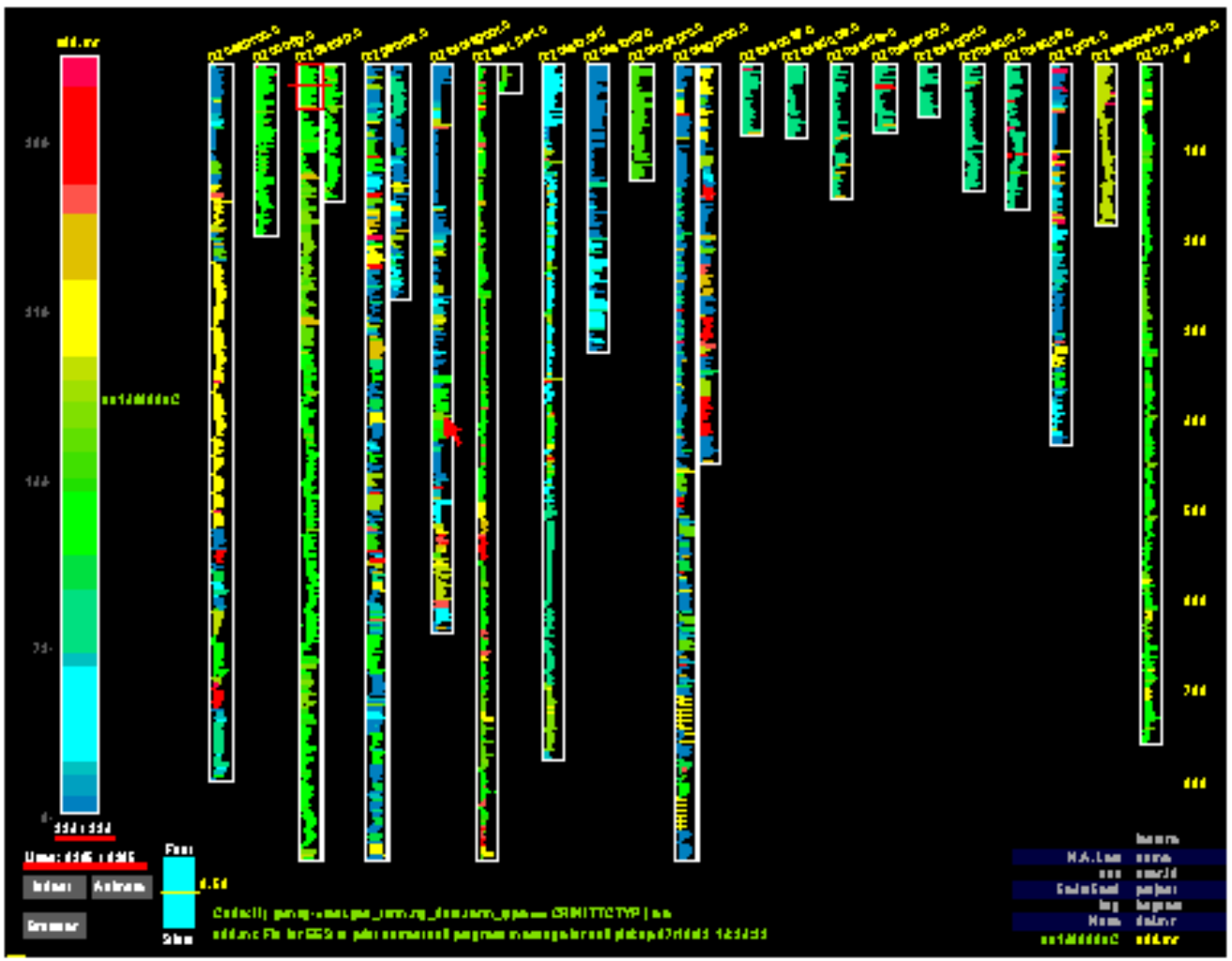
My view

- Star diagrams may not be “the” answer
- But I like the idea that they encourage people
  - To think clearly about a maintenance task, reducing the chances of an ad hoc approach
  - They help track mundane aspects of the task, freeing the programmer to work on more complex issues
  - To focus on the source code
- Murphy/Kersten and Mylyn and tasktop.com are of the same flavor….

SeeSoft: Eick et al.

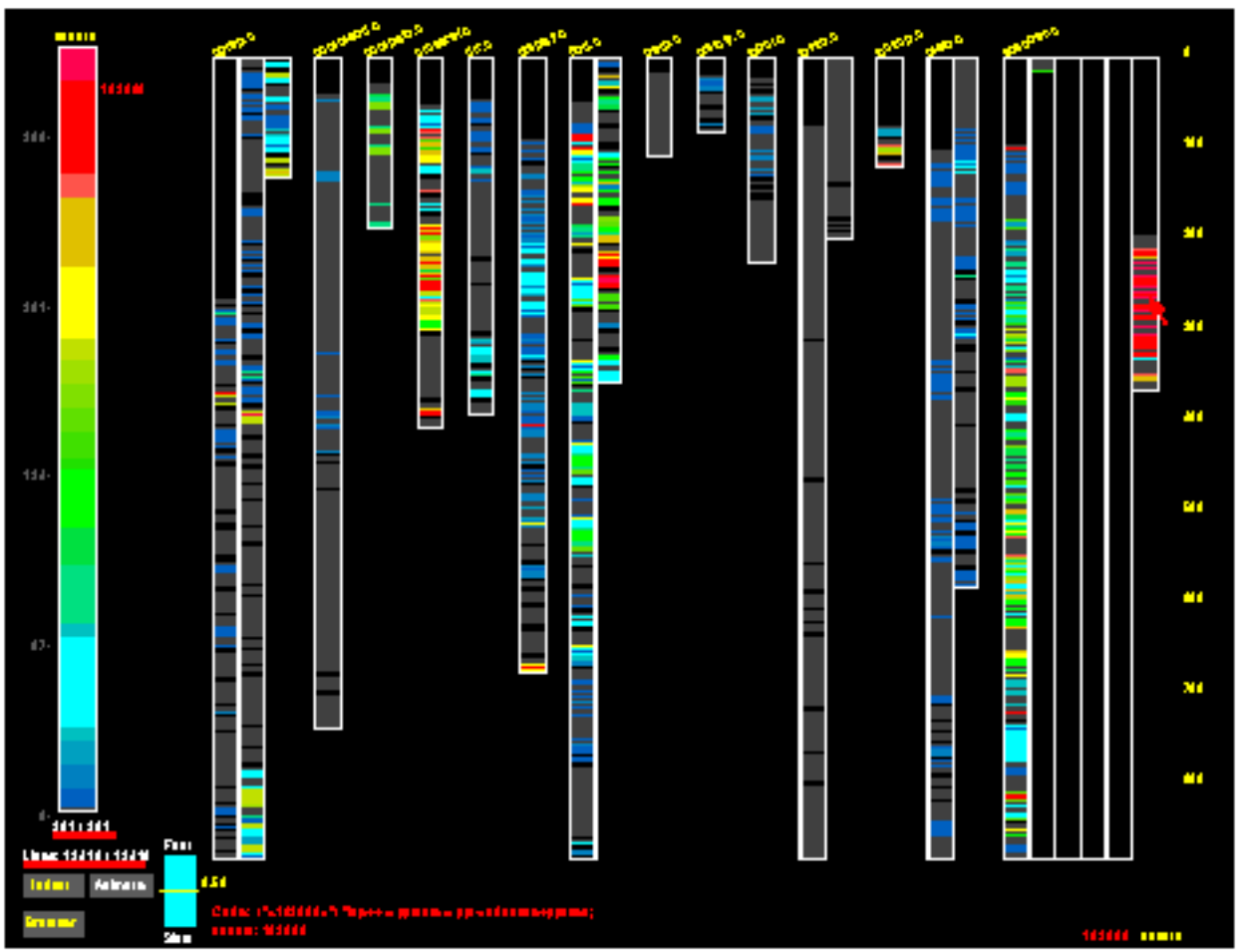
- Visualize text files by
  - mapping each line into a thin row
  - colored according to a statistic of interest
- Focus on source code, with sample statistics including
  - age, programmer, or functionality of each line
  - Data extracted from version control systems, static analysis and profiling
- User can manipulate this representation to find interesting patterns in software
- Applications include data discovery, project management, code tuning and analysis of development methodologies

Code age:
newest code in red, oldest in blue



Execution profile:

red shows hot spots, non-executed lines are gray/black



SeeSoft

- SeeSoft seems excellent for building important, qualitative understanding of some aspects of source code
- It also links in effectively with the underlying source code
- It is flexible in terms of what statistics are viewed
  - It’s not entirely clear how much work is needed to add a new statistic

Interlude: education

- OK, what should (should not) go in an undergraduate education leading to jobs like yours?
- A couple of rules
  - It's a zero-sum game (something goes in, something comes out)
  - You cannot assume every student is precisely like yourself

A relatively new, “hot” approach: Mining Software Repositories

- “Research is now proceeding to uncover the ways in which mining [software] repositories can help to understand software development, to support predictions about software development, and to plan various aspects of software projects.” [MSR 2007 web page]

- Repositories are broadly defined to include code, defect databases, version control information, programmer communications, etc.

Note: distinct from in-field testing

- …gathering data from actual usage in the field that can be used to improve the product
- …more in a later lecture

What has enabled this approach?

- The Internet
- Open source
- More repositories
- More complex repositories
- Fast/cheap processors
- Big/cheap memories
- Big/cheap disks
- Data mining/machine learning results
- New analyses
- …and surely more

Underlying premise

- We believe there is something – actually, a lot of things – that can be learned from studying these repositories
- But it presents a paradox – if we think most software is low quality, how can we learn by studying the repositories?

Four ways to resolve this paradox

- The premise is false – most (or even all) software is good
- We can learn about good practices from bad software
- We can distinguish good from bad software and only study the good ones
- Mining software repositories cannot succeed

A pertinent tangent: science vs. engineering

- Science focuses on learning about the structure and behavior of the real world
- Engineering focuses on designing useful things
- “Computer science” as a research field tends to do both in an unusually intermingled way
  - At times, the distinction is still instructive

Mining software repositories: science, engineering, both?

- Mining software repositories is largely a scientific venture, albeit it with respect to human-engineered artifacts
- That is, software is a part of our reality, and there is enough of it to study it
- There is no question (to me) that this is valuable and that we can learn a lot from this
  - Belady and Lehman showed this, among others – statistical results that deepened our understanding about the relationships among users, program change and program structure

An important question remains

- Can we learn things that can then be used to improve the engineering side?
  - That is, can we learn specific, concrete things that lead to better software, better software practices, better software tools, better …?
- Unless we can provide useful feedback to the engineering side, I believe the long-term value of mining software repositories will be limited
  - Belady and Lehman's work has not, overall, let us to "better" software, but rather to a better understanding of software
  - Although I am largely uneducated in software metrics, I believe that this is also a limitation of that approach

Mining software repositories vs. reverse engineering etc.

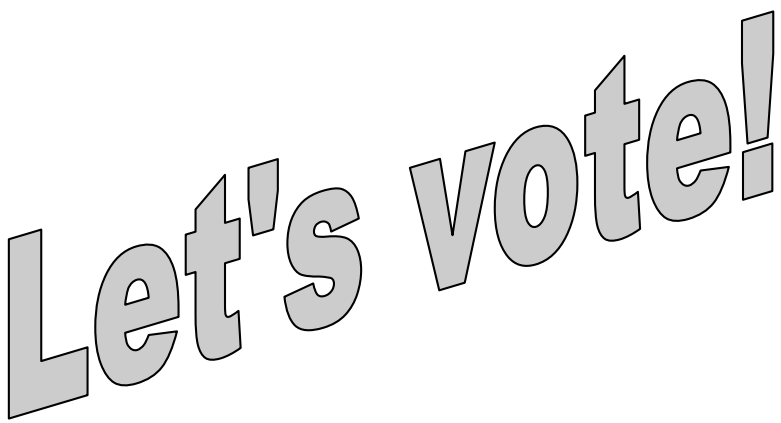
- In reverse engineering, reengineering, program comprehension, etc. approaches, the information from a given software system is used to help software engineers improve that system
- Mining software repositories feedback must provide information that is more broadly applicable – probably not to all software systems, but to some (many) that have not been analyzed

Field of Dreams: "If you build it, they will come"

- Separate reality from fantasy – just mining software repositories will not by itself cause significant advances in software engineering

Four ways to resolve this paradox

1. The premise is false – most (or even all) software is good
2. We can learn about good practices from bad software
3. We can distinguish good from bad software and only study the good ones
4. Mining software repositories cannot succeed



Two ways to resolve the paradox

- The premise is false – most (or even all) software is good
- We can learn about good practices from bad software
- We can distinguish good from bad software and only study the good ones
- MSR cannot succeed

Programming language design

- Many of the advances come from observations that distinguish “good” programs from “bad” ones
- Classic examples include control constructs, abstract data types, …
- A related, but non-language example is design patterns
- These advances come from studying the “good” programs, not the “bad” ones

Change support: largely *ad hoc*

- In contrast, support for software change has been much less disciplined
- Relatively little has been done to make it easier to make good changes and harder to make bad changes
- We are seeing some movement in making this more systematic: refactoring is perhaps the clearest example

Unjustified claim

- The key to the (engineering) success of mining software repositories is identifying “good” changes in a specific and concrete way
- This appears to be harder than improving languages, for several reasons
  - Doing it automatically is almost surely harder
  - Looking at change is harder than looking at programs, at least at present
  - Even with success, we have fewer ways to encode “good” changes than “good” programs, at least at present
  - …

The way to resolve the paradox

- The premise is false – most (or even all) software is good
- We can learn about good practices from bad software
- We can distinguish good from bad software and only study the good ones … and hope we can learn from them!
- MSR cannot succeed

Change: Now passing the assembly language phase

- In general, software changes are applied by stringing together a set of low-level operations (keystrokes, macros, operations, etc.)
- Just as people saw useful patterns in assembly language – leading to, for example, high-level control constructs – we are beginning to see analogous patterns in change
  - refactoring
  - simultaneous text editing/linked editing
  - co-evolving entitites
  - …

This positions us

- …to move from low-level and statistical models of change to a higher-level, specific and concrete model of change
- A key piece of this shift to a higher-level model is making change a first-class notion

Co-evolution

- Ying et al. (and several others)
  - "To augment existing analyses and to help developers identify relevant source code during a modification task, we have developed an approach that applies data mining techniques to determine change patterns -- sets of files that were changed together frequently in the past -- from the change history of the code base. Our hypothesis is that the change patterns can be used to recommend potentially relevant source code to a developer performing a modification task."
- Or, "other people who changed this file were also interested in the following files"

Team Tracks: DeLine et al. Microsoft Research

- Team Tracks guides code exploration
  - Records the team's code navigation during development
  - Mines that data to prune the working set and guide navigation
- Does navigation frequency indicate importance?
  - An empirical study suggests "yes"
- Does Team Tracks help with task completion rates?
  - An empirical study suggests "yes"

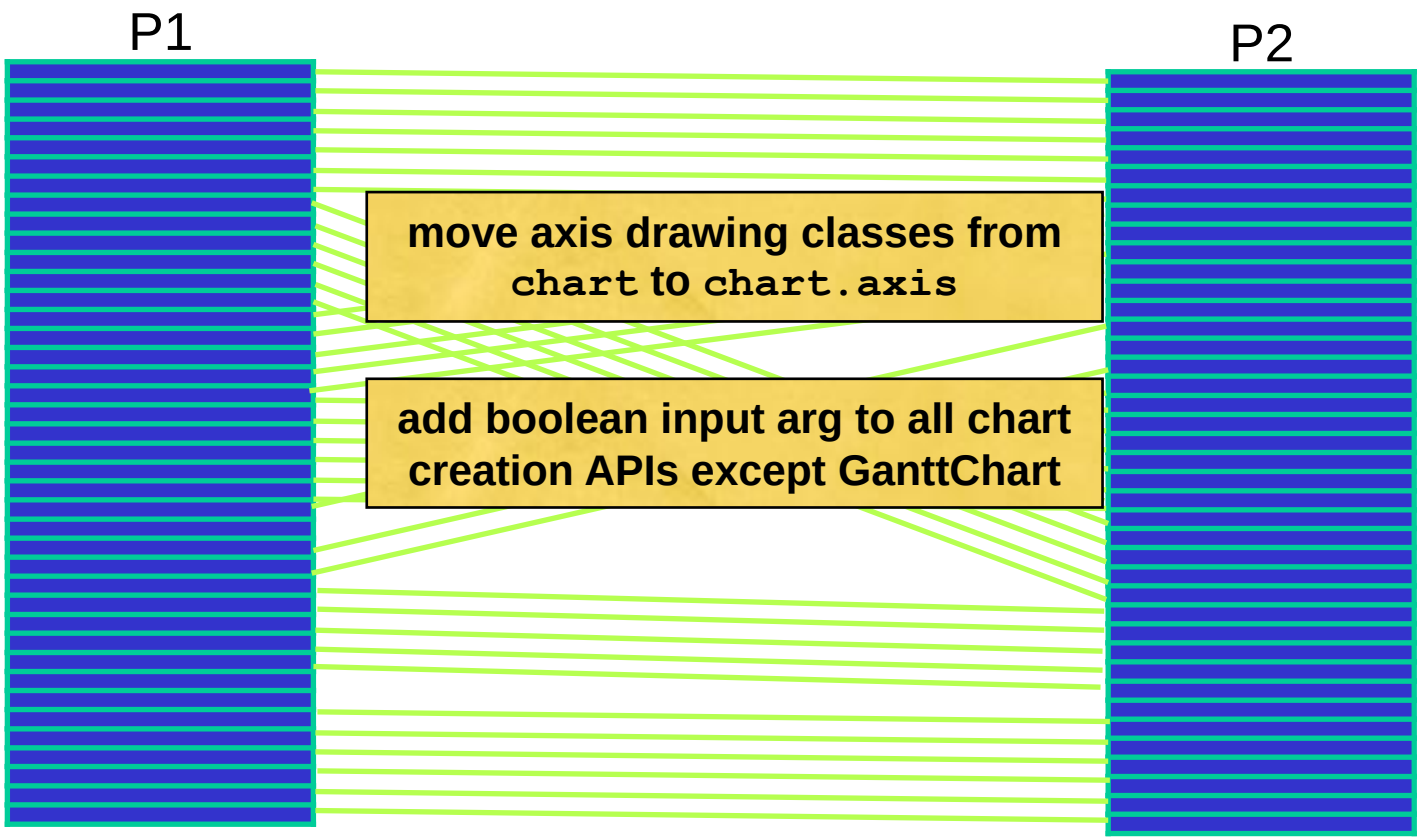
Other topics (discussed at MSR 2005 etc.)

- Approaches to study the quality of the mined data along with guidelines to ensure the quality of the recovered data
- Proposals for exchange formats, meta-models, and infrastructure tools to facilitate the sharing of extracted data and to encourage reuse and repeatability
- Models for social and development processes that occurin large software development projects
- Search techniques to assist developers in finding suitable components for reuse
- Techniques to model reliability and defect occurrences
- Analysis of change patterns to assist in future development
- Case studies on extracting data from repositories of large long lived projects
- Suggestions for benchmarks, consisting of large software repositories, to be shared among the community

Another example approach:
Miryung Kim [UW]

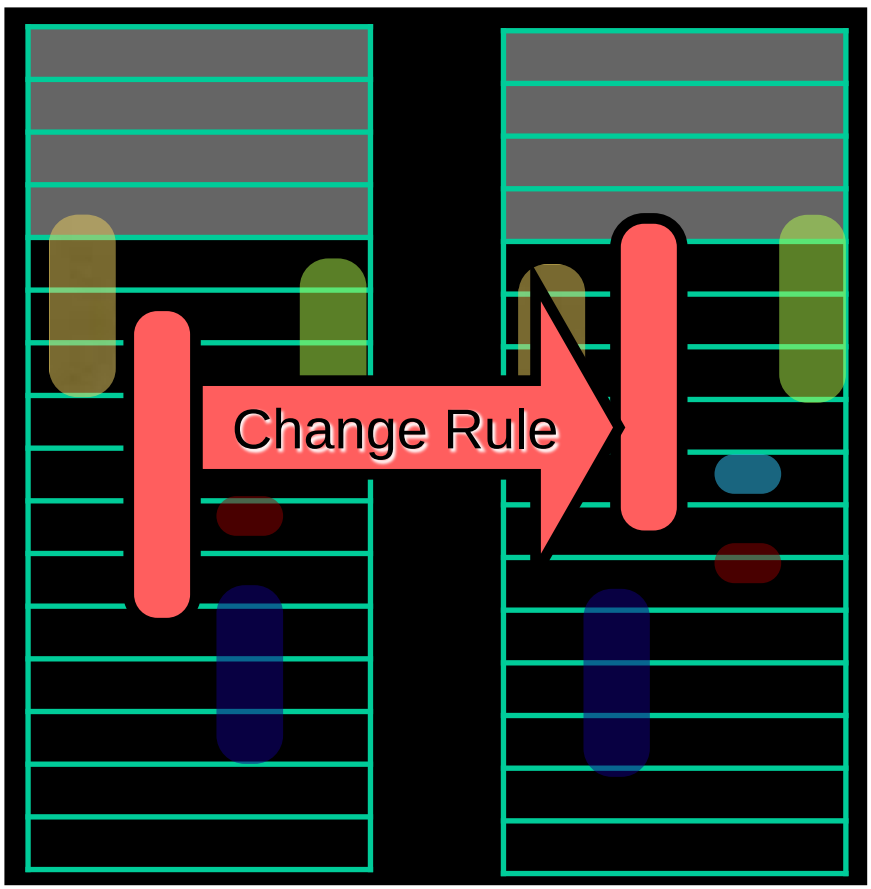
- Represent change explicitly using first-order relational rules
- Infer change rules from pairs of program versions
- May enable new ways to understand software evolution and to support tools that aid in software evolution

Cross version matching



Limitation of existing matching approaches:
hard to examine and to extract high-level change intent

Change rule

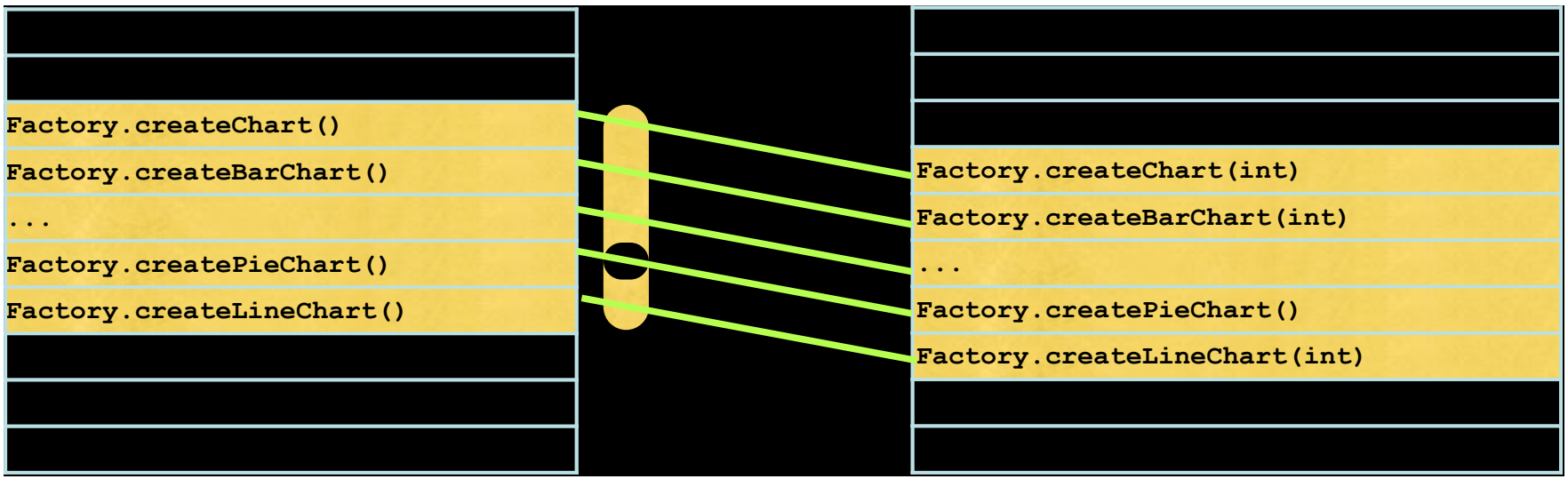



```
For all x in (scope - exceptions)
    transform(x)
```

Transformations
(Above the Level of Method Header)

- **packageReplace(x:Entity, fr:Text, t:Text)**
- **classReplace(x:Entity, fr:Text, t:Text)**
- **procedureReplace(x:Entity, fr:Text, t:Text)**
- **returnReplace(x:Entity, fr:Text, t:Text)**
- **inputSignatureReplace(x:Entity, fr:List[Text], t:List[Text])**
- **argReplace(x:Entity, fr:Text, t:Text)**
- **argAppend(x:Entity, t:List[Text])**
- **argDelete(x:Entity, t:Text)**
- **typeReplace(x:Entity, fr:Text, t:Text)**

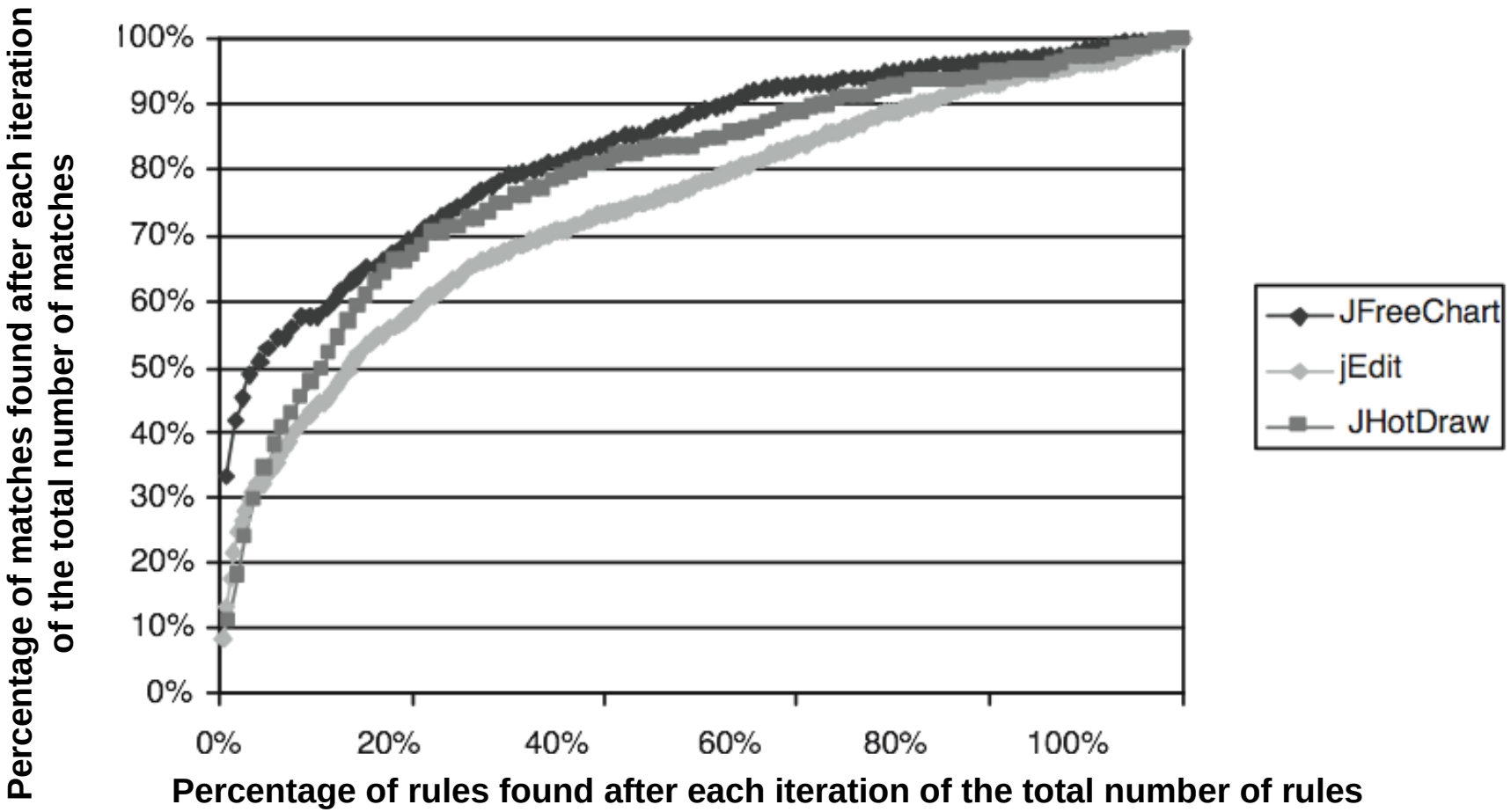
Example change rule




```
For all x
    in Factory.create*Chart(*)
    except {Factory.createPieChart()}
  argAppend(x, [int])
```

14 matches and 1 exception

Initial results



Top 20% of the rules find over 55% of the matches

Top 40% of the rules find over 70% of the matches

An initial step…

- …in making change a first class notion
- Many other choices of
  - transformations
  - inference algorithms
  - rule representations
  - …
- Haven't yet showed benefit of rules to drive applications: documentation assistant, bug finding, API evolution analysis, API update, …

Can it help with mining software repositories?

- Maybe
- In particular, it may be the more effective rules that provide insight and potential for representing higher-level changes

Discovering and Representing Logical Structure in Code [Kim, Beall, Notkin]

- Follow-up work to matching: logical structured delta (LSD)
- A significant gap between how programmers think about code change and how change is represented in widely used tools such as diff.
- LSD explicitly and concisely captures systematic changes to a program's dependency structure, along with an engine that automatically infers such changes as logic rules
- Each rule represents a set of atomic transformations that share similar structural characteristics: e.g., crosscutting concerns, refactorings, consistent updates of code, clones, etc.
- Initial evaluation on several open source projects shows that LSDs are orders of magnitude more concise than diff outputs

Sample rules: inferred

- **`added_type("AbsRegistry")`**
- **`current_inheritedmethod(m, "AbsRegistry", t) => added_inheritedmethod(m, "AbsRegistry", t)`**
- **`past_subtype("NameSvc", t) ^ past_field(f, "host", t) => deleted_field(f, "host", t) except t = "LmiRegistry"`**
- **`past_subtype("NameSvc", t) ^ past_method(m, "getHost", t) => deleted_method(m, "host", t) except t = "LmiRegistry"...`**

- **`host`** related fields and methods are pulled up from **`NameSvc's`** subclasses to **`AbsRegistry`** class except from **`LmiRegistry`**.

Sample rules

- **current_calls(m, "NamingExceptionHelper.create(Exception)" => added_calls(m, "NamingExceptionHelper.create(Exception)"**
- **past_calls (m, "JNDIRemoteSource.getResouce()") => deleted_calls(m, "Throwable.printStackTrace()") ...**
- **current_inheritedmethod(m, "AbsContext", t) => added_inheritedmethod(m, "AbsContext", t)**
- **past_method(m, mn, "JRMPContext") => deleted_method(m, mn,"JRMPContext")**

- All calls to **NameExceptionHelper** are newly added ones, and all methods that called **getResource** no longer call **printStackTrace**.
- Create **AbsContext** by extracting common methods from **Context** classes.

Conclusion

- There is no paradox
- Mining software repositories is promising
- We need to focus on change as an explicit, first-class notion …
- Lots of opportunities, but with a focus on the engineering

Interlude: configuration testing

- How do you test and/or analyze software that runs in many different configurations?