CSE503: Software Engineering

David Notkin
University of Washington
Department of Computer Science & Engineering
Winter 2002

1

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?

2

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
  - We've talked about many possible relations: calls, uses, registers-in, names, #includes, etc.

3

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

4

Information characteristics

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

5

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

6

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

7

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

8

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

9

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

10

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

11

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

12

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

13

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 *`

14

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

15

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`!

16

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);
    ```

17

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

18
    ```

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

19

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

20

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

21

Griswold's 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

22

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
  - The user touches only one place

23

Limited power

- The actual tool and approach has limited power
- Can help translate one of Parnas' KWIC decompositions to the other
- 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
- Some related work specifically in OO by Opdyke and Johnson
- Question: How do you find appropriate restructuring?

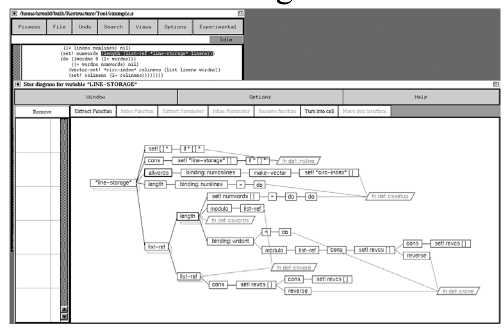
24

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

25

A star diagram

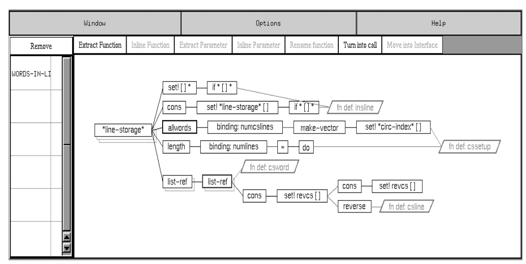


Interpreting a star diagram

- 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

27

After some changes



28

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 the star diagram tools adopted simpler strategies for handling completeness and consistency

29

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

30