CSE584: Software Engineering

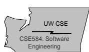
Lecture 4 (April 22, 1997)

David Notkin
Dept. of Computer Science & Engineering
University of Washington
www.cs.washington.edu/homes/notkin

UW CSE
CSE584: Software Engineering

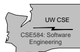
Lecture 4, Outline [approximate minutes]

- Software change
  - Basic background [15]
  - Approaches to change [15]
- Alternative approaches to maintenance
  - Introduction [15]
  - Source models [30]
- Break [10]
- Roundtable [15]
  - Tools you use for change
- Alternative approaches to maintenance (con't)
  - Visualizing source models [60]



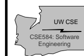
Software evolution

- 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



Why does it change?

- Software changes does not change primarily because it doesn't work right
- 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



Kinds of change

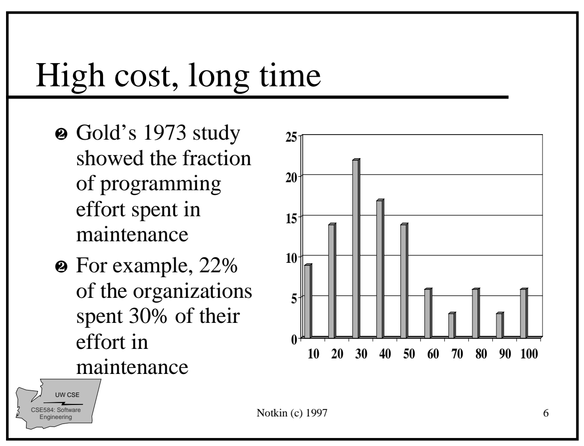
- Corrective maintenance
  - Fixing bugs in released code
- Adaptive maintenance
  - Porting to new hardware or software platform
- Perfective maintenance
  - Providing new functions

70
60
50
40
30
20
10
0
Corrective
Adaptive
Perfective
Lientz & Swanson 1980

UW CSE
CSE584: Software Engineering

High cost, long time

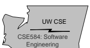
- Gold's 1973 study showed the fraction of programming effort spent in maintenance
- For example, 22% of the organizations spent 30% of their effort in maintenance



UW CSE
CSE584: Software Engineering

Total life cycle cost

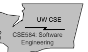
- Lientz and Swanson determined that at least 50% of the total life cycle cost is in maintenance
- There are several other studies that are reasonably consistent
- General belief is that maintenance costs somewhere between 50-75% of total life cycle costs



Notkin (c) 1997 7

Open question

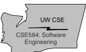
- How much maintenance cost is “reasonable?”
  - Corrective maintenance costs are ostensibly not “reasonable”
  - How much adaptive maintenance cost is “reasonable”?
  - How much perfective maintenance cost is “reasonable”?
- Measuring “reasonable” costs in terms of percentage of life cycle costs doesn’t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



Notkin (c) 1997 8

High-level answer

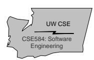
- For perfective maintenance, it seems that the objective should be for the cost of the change in the implementation to be proportional to the cost of the change in the specification (design)
  - Ex: Allowing dates for the year 2000 is (at most) a small specification change
  - Ex: Adding call forwarding is a more complicated specification change



Notkin (c) 1997 9

Observations about maintenance

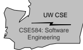
- Maintainers often get less respect than developers
- Maintenance is generally assigned to the least experienced programmers
- Software structure degrades over time
- Documentation is often poor and is often inconsistent with the code



Notkin (c) 1997 10

Laws of Program Evolution [Belady & Lehman]

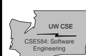
- Law of continuing change
  - “A large program that is used undergoes continuing change or becomes progressively less useful.”
  - Analogies to biological evolution have been made; the rate of change in software is far faster



Notkin (c) 1997 11

Law of increasing complexity

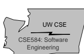
- “As a large program is continuously changed, its complexity, which reflects deteriorating structure, increases unless work is done to maintain or reduce it.”
  - Complexity, in part, is relative to a programmer’s knowledge of a system
    - » Novices vs. experts doing maintenance
  - Cleaning up structure is done relatively infrequently



Notkin (c) 1997 12

Law of statistically regular growth

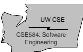
- “Measures of [growth] are cyclically self-regulating with statistically determinable trends and invariances.”
  - (You can run but you can’t hide)
  - Based on data from OS/360 and some other systems
  - Ex: Content in releases decreases or time between releases increases



Notkin (c) 1997 13

And two others

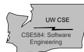
- “The global activity rate in a large programming project is invariant.”
- “For reliable, planned evolution, a large program undergoing change must be made available for regular user execution at maximum intervals determined by its net growth.”
  - This is related to Microsoft’s “daily builds”



Notkin (c) 1997 14

Approaches to reducing cost

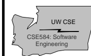
- Design for change
  - Information hiding, layering, open implementation, etc.
- Tools to support change
  - grep, etc.
  - Reverse engineering, program understanding, system summarization, …



Notkin (c) 1997 15

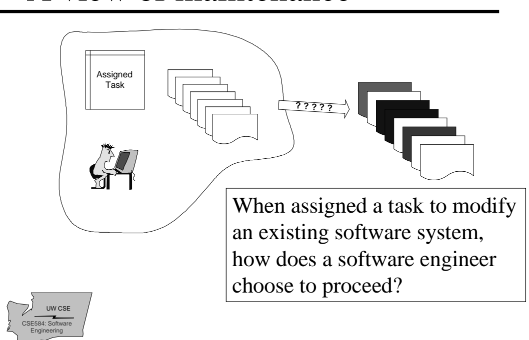
Approaches to reducing cost

- Improved documentation
  - Discipline, stylized approaches
- Reducing bugs
  - Many techniques, covered later in the quarter
- Increasing correctness of specifications
- Others?



Notkin (c) 1997 16

A view of maintenance



17

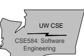
Sample task

- 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?

18

Recap: example

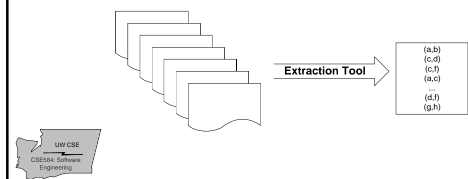
- What information did you need?
- What information was available?
- What tools produced the information?
  - Did you think about other pertinent tools?
- How accurate was the information?
  - Any false information? Any missing true information?
- How did you view and use the information?



19

Source models

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

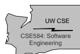


UW CSE
CSE584: Software Engineering

20

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?



21

Combining source models

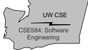
- Source models may be produced by combining other source models using simple relational operations; for example,
  - Extract a source model indicating which functions reference which global variables
  - Extract a source model indicating which functions appear in which modules
  - Join these two source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

UW CSE
CSE584: Software Engineering

22

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



23

Information characteristics

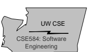
	no false positives	*false positives*
no false negatives	ideal	conservative
false negatives	optimistic	approximate

UW CSE
CSE584: Software Engineering

24

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, this is not possible
  - Ideal call graphs, for example, are uncomputable
- For some source models, achieving the ideal may be difficult in practice



25

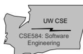
Conservative source models

- 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

26

Optimistic source models

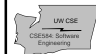
- These include only truth but may omit some true information
- Often come from dynamic extraction
- Ex: In white-box code coverage in testing
  - Indicating which statements have been executed by the selected test cases
  - Others statements may be executable with other test cases



27

Approximate source models

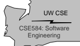
- May include some false information and may omit some true information
- These source models can be useful for maintenance tasks
  - Especially useful when a human engineer is using the source model, since humans deal well with approximation
- Turns out many tools produce approximate source models (more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



28

Static vs. dynamic source models

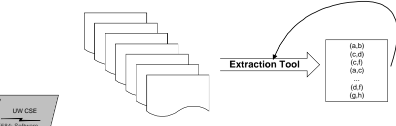
- Source model extractors can work
  - *statically*, directly on the system's artifacts, or
  - *dynamically*, on the execution of the system, or
  - a combination of both
- Ex:
  - A call graph can be extracted statically by analyzing the system's source code or can be extracted dynamically by profiling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



29

Must iterate

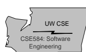
- Usually, the engineer must iterate to get a source model that is "good enough" for the assigned task
- Often done by inspecting extracted source models and refining extraction tools
- May add and combine source models, too



30

Another maintenance task

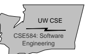
- Given a software system, rename a given variable throughout the system
  - Ex: `angle` should become `diffraction`
  - Probably in preparation for a larger task
- Semantics must be preserved
- This is a task that is done infrequently
  - Without it, the software structure degrades more and more



31

What source model?

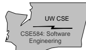
- Our preferred source model for the task would be a list of lines (probably organized by file) that reference the variable `angle`
- A static extraction tool makes the most sense
  - Dynamic references aren't especially pertinent for this task



32

Start by searching

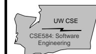
- Let's start with grep, the most likely tool for extracting the desired source model
- The most obvious thing to do is to search for the old identifier in all of the system's files
  - grep angle *



33

What files to search?

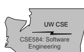
- It's hard to determine which files to search
  - Multiple and recursive directory structures
  - Many types of files
    - » Object code? Documentation? (ASCII vs. non-ASCII?) Files generated by other programs (such as `yacc`)? Makefiles?
  - Conditional compilation? Other problems?
- Care must be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



34

False positives

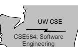
- `grep angle` *`[system's files]`*
- There are likely to be a number of spurious matches
  - `…triangle…, …quadrangle…`
  - `/* I could strangle this programmer! */`
  - `/* Supports the small planetary rovers presented by Angle & Brooks (IROS '90) */`
  - `printf("Now play the Star Spangled Banner");`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



35

More false negatives

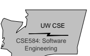
- Some languages allow identifiers to be split across line boundaries
  - Cobol, Fortran, PL/I, etc.
  - This leads to potential false negatives
- Preprocessing can hurt, too
  - ```
    #define deflection angle
    ...
    deflection = sin(theta);
    ```



36
    ```

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



37

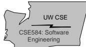
Tools vs. task

- 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

38

Finding vs. updating

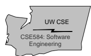
- Even after you have extracted a source model that identifies all of (or most of) the lines that need to be changed, you have to change them
- Global replacement of strings is at best dangerous
- Manually walking through each site is time-consuming, tedious, and error-prone



39

Downstream consequences

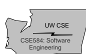
- After extracting a good source model by iterating, the engineer can apply the renaming to the identified lines of code
- However, since the source model is approximate, regression testing (and/or other testing regimens) should be applied



40

An alternative approach

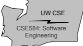
- Griswold developed a meaning-preserving program restructuring tool that can help
- For a limited set of transformations, the engineer applies a local change and the tool applies global compensating changes that maintain the program’s meaning
  - Or else the change is not applied
  - Reduces errors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



41

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 in terms of either computation size or space



42

Recap

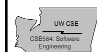
- ❷ "There is more than one way to skin a cat"
- ❷ The engineer must decide on a source model needed to support a selected approach
- ❷ The engineer must be aware of the kind of source model extracted by the tools at hand
- ❷ The engineer must iterate the source model as needed for the given task

UW CSE
CSE584: Software Engineering

43

Build up your idioms

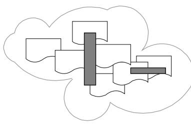
- ❷ Handling each task independently is hard
- ❷ You can build up some more common idiomatic approaches
  - Some tasks, perhaps renaming, are often part of larger tasks and may apply frequently
  - Also internalize source models, tools, etc. and what they are (and are not) good at
- ❷ But don't constrain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

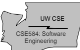


44

Another task: isolating a subsystem

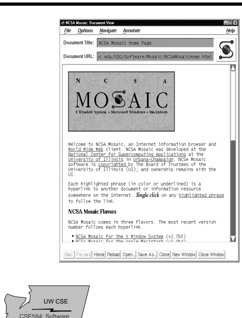
- ❷ Many maintenance tasks require identifying and isolating functionality within the source
  - sometimes to extract the subsystem
  - sometimes to replace the subsystem





45

Mosaic

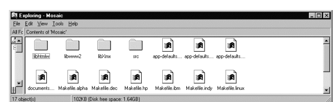


- ❷ The task is to isolate and replace the TCP/IP subsystem that interacts with the network with a new corporate standard interface
- ❷ First step in task is to estimate the cost

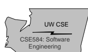
UW CSE
CSE584: Software Engineering

46

Mosaic source code

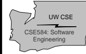


- ❷ After some configuration and perusal, determine the source of interest is divided amongst 4 directories with 157 C header and source files
- ❷ Over 33,000 lines of non-commented, non-blank source lines



Some initial analysis

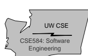
- ❷ The names of the directories suggest the software is broken into:
  - code to interface with the X window system
  - code to interpret HTML
  - two other subsystems to deal with the world-wide-web and the application (although the meanings of these is not clear)



48

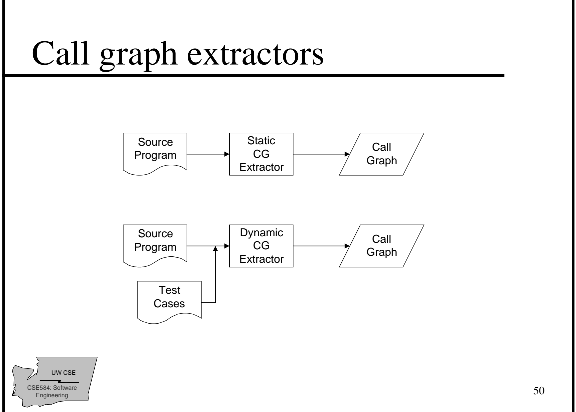
Apply the framework

- What source model would be useful?
  - calls between functions (particularly calls to Unix TCP/IP library)
- How do we get this source model?
  - *statically* with a tool that analyzes the source or *dynamically* using a profiling tool
  - these differ in information characterization produced



49

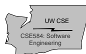
Call graph extractors



50

Apply the framework...

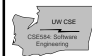
- What precision in the source model is appropriate?
  - want ideal but can't get it
  - second best would be conservative, which implies using the tools that produce calls information through a static analysis of the source



51

Call graph extraction tools (C)

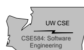
- Two basic categories: lexical or syntactic
  - lexical
    - » e.g., awk, mkfunctmap, lexical source model extraction (LSME)
    - \- likely produce an approximate source model
    - \+ extract calls across configurations
    - \+ can extract even if we can't compile
    - \+ typically fast



52

CGE 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

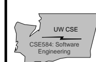


53

Apply a syntactic CGE tool

- C Information Abstractor (CIA)
  - http://www.research.att.com:80/library/books/reuse/license/packages/95/cia.html
  - 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
```



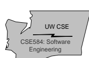
54

Querying the source model

- Continue to follow the trace of references:

```
cref func - func HTDoConnect
HTFTP.c    get_connection -> HTTCP.c HTDoConnect
HTGopher.c HTLoadGopher   -> HTTCP.c HTDoConnect
HTNews.c   HTLoadNews     -> HTTCP.c HTDoConnect
HTTP.c     HTLoadHTTP     -> HTTCP.c HTDoConnect
```

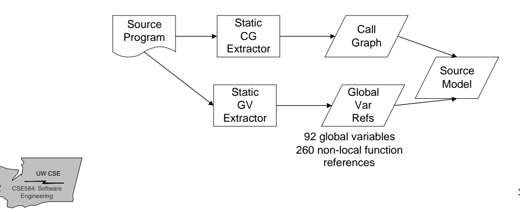
- Can dump the entire source model
  - 3966 references



55

Iterate and add information

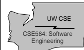
- Sometimes may want to augment the source model
  - for example, add in global variable information



56

How precise is the source model?

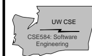
- Are the source models extracted by CIA conservative?
- It is typically difficult to determine the answer to this kind of question
- But, to perform a task confidently, you need to get a handle on the precision
  - maybe by reading the tool's documentation
  - maybe by comparison to other tools
  -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



57

A CGE experiment

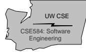
- To investigate several call graph extractors for C, we ran a simple experiment
  - For several applications, extract call graphs using several extractors
  - Applications: mapmaker, mosaic, gcc
  - Extractors: CIA, rigiparse, Field, cflow, mkfunctmap



58

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)?

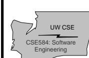


59

Pairwise comparison (example)

- CIA vs. Field for Mosaic
  - 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

89% 5% 12%
Field
4258



60

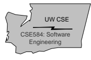
Quantitative Results

- 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

61

Qualitative results

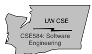
- Sampled elements to identify false positives and false negatives
- Mapped the tuples back to the source code and performed manual analysis by inspection
- Every extractor produced some false positives and some false negatives



62

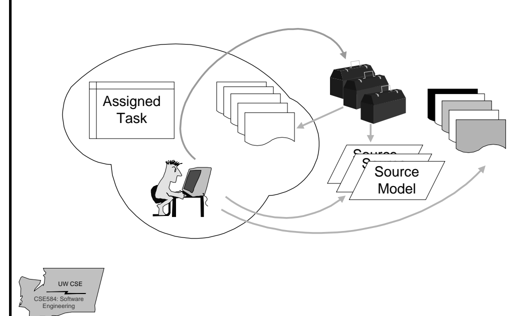
Call graph characterization

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



63

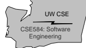
Back to the isolation task



64

Back to the isolation task...

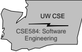
- What we have
  - approximate call and global variable reference information
- What we want
  - increase confidence in source model
- Action:
  - collect dynamic call information to augment source model



65

Augmenting with dynamic calls

- Compile Mosaic with profiling support
- Run with a variety of test paths and collect profile information
- Extract CG source model from profiler output
  - 1872 calls
  - 25% overlap with CIA
  - 49% of calls reported by gprof not reported by CIA



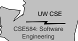
66

Alternative action

- Alternatively, we may have wanted to augment with calls information extracted using a lexical technique
- For example, lexical source model extraction tool:

```
[ <type> ] <fn> \( [ { <arg> }+ ] \)
   [ { { <ty> }+ ; }+ ] \{

       <cf> \( [ { <arg> [ , ] }+ ] \)
```

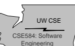


67

Are we done?

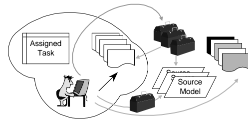
- We are still left with a fundamental problem: how to deal with one or more large source models?
  - Mosaic source model:

static function references (CIA)	3966
static function-global var refs (CIA)	541
dynamic function calls (gprof)	1872
Total	6379

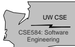


68

One approach



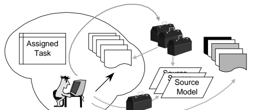
- Use a query tool against the source model(s)
  - maybe grep?
  - maybe source model specific tool?
- As necessary, consult source code

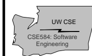


69

Other approaches

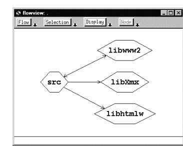
- Visualization
- Reverse engineering
- Summarization





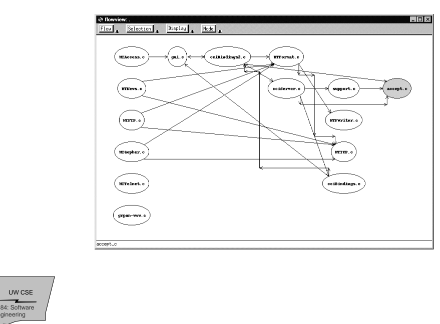
70

Visualization

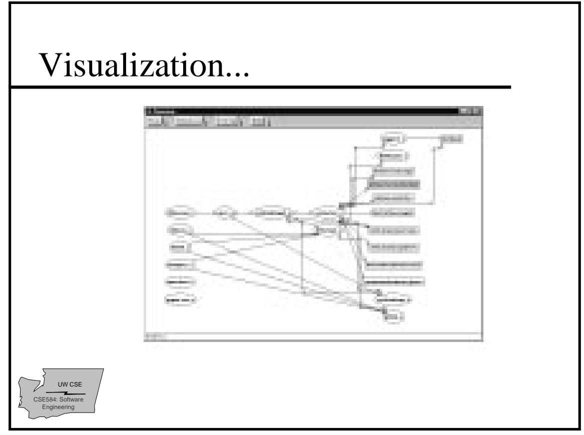


- e.g., Field, Plum, Imagix 4D, McCabe, etc. (Field's flowview is used above and on the next few slides...)

Visualization...

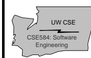


Visualization...



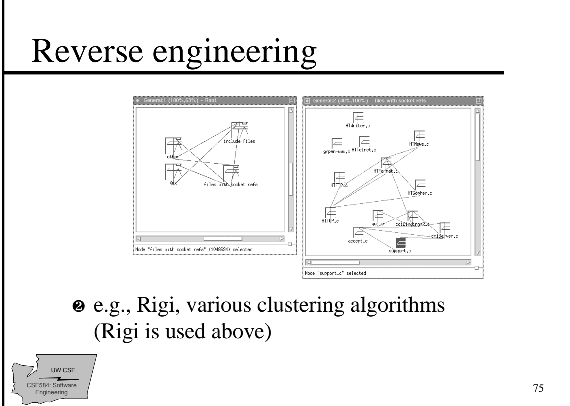
Visualization...

- provides a “direct” view of the source model
  - can produce a “precise” view
- – view often contains too much information
  - use elision, producing an “optimistic” view



74

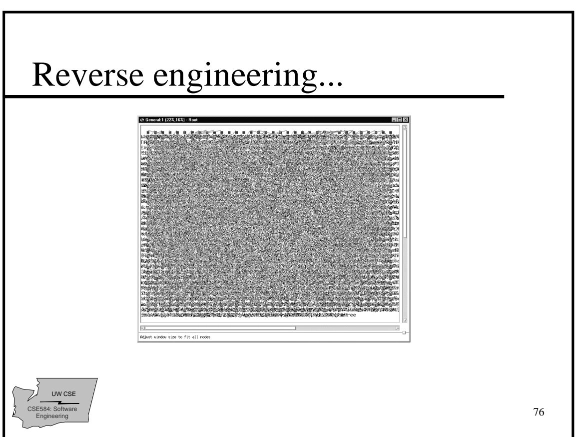
Reverse engineering



❷ e.g., Rigi, various clustering algorithms (Rigi is used above)

75

Reverse engineering...



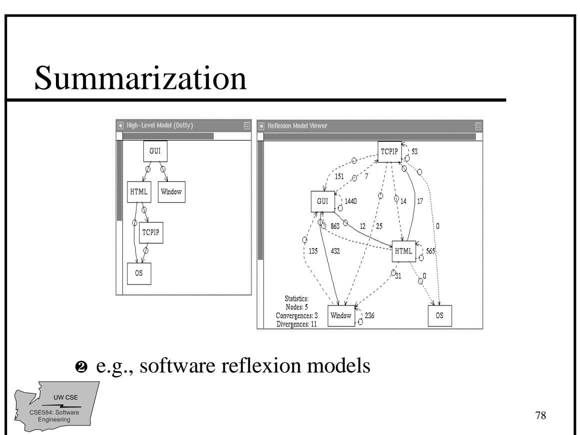
76

Reverse engineering...

- \+ generally produces a higher-level view that is consistent with source
  - similar to visualization, can produce a “precise” view (although this might be a precise view of an approximate source model)
- \- sometimes view still contains too much information leading again to the use of techniques like elision
  - may end up with “optimistic” view

77

Summarization



❷ e.g., software reflexion models

78

Summarization...

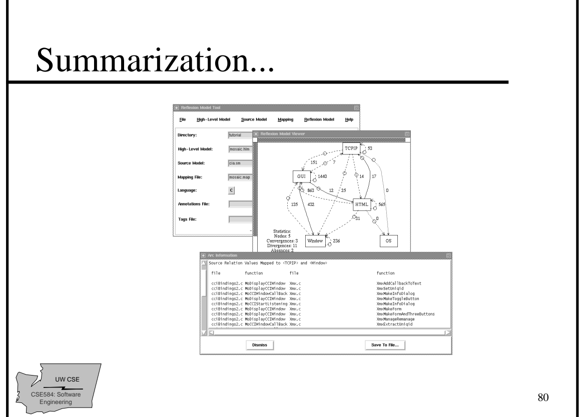
- a map file specifies the correspondence between parts of the source model and parts of the high-level model

```
[ file=HTTCP          mapTo=TCPIP ]
[ file=^SGML          mapTo=HTML ]
[ function=socket     mapTo=TCPIP ]
[ file=accept         mapTo=TCPIP ]
[ file=cci            mapTo=TCPIP ]
[ function=connect    mapTo=TCPIP ]
[ file=Xm             mapTo=Window ]
[ file=^HT            mapTo=HTML ]
[ function=.*         mapTo=GUI ]
```

UW CSE
CSE584: Software Engineering

79

Summarization...

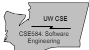


UW CSE
CSE584: Software Engineering

80

Summarization...

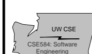
- + condense (some or all) information in terms of a high-level view quickly
  - in contrast to visualization and reverse engineering, produce an "approximate" view
  - iteration can be used to move towards a "precise" view
- + some evidence that it scales effectively
- – may be difficult to assess the degree of approximation



81

Case study: A task on Excel

- A series of approximate tools were used by a Microsoft engineer to perform an experimental reengineering task on Excel
- The task involved the identification and extraction of components from Excel
- Excel comprises about 1.2 million lines of C source
  - about 15,000 functions spread over ~400 files



82

The process used

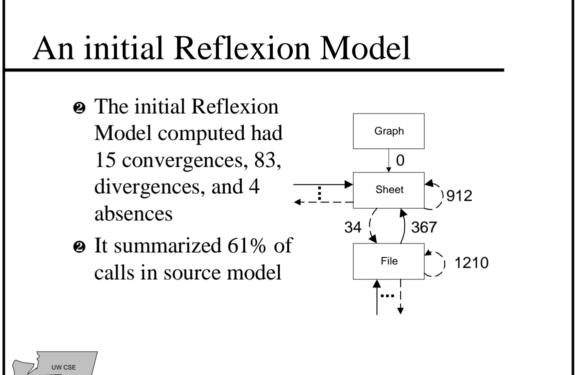
Reflexion Model
4
3
170 entries
1 Model
Mapping
RM Tools
13 nodes
~19 arcs
Extraction Tool
Source Model
77,746 calls
System Artifacts
2

UW CSE
CSE584: Software Engineering

83

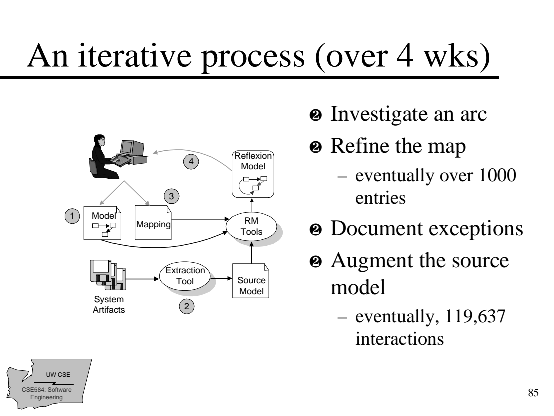
An initial Reflexion Model

- The initial Reflexion Model computed had 15 convergences, 83, divergences, and 4 absences
- It summarized 61% of calls in source model

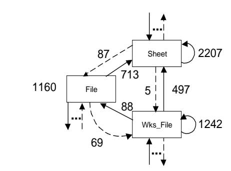


UW CSE
CSE584: Software Engineering

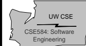
84



A refined Reflexion Model



- A later Reflexion Model summarized 99% of 131,042 call and data interactions
- This approximate view of approximate information was used to reason about, plan and automate portions of the task

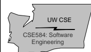


86

Results

- Microsoft engineer judged the use of the Reflexion Model technique successful in helping to understand the system structure and source code

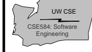
  “Definitely confirmed suspicions about the structure of Excel. Further, it allowed me to pinpoint the deviations. It is very easy to ignore stuff that is not interesting and thereby focus on the part of Excel that I want to know more about.” — Microsoft engineer



87

Wrap up

- Maintenance is done in a relatively *ad hoc* way
  – Much more *ad hoc* than design, I think
- Putting some intellectual structure on the problem might help



Notkin (c) 1997 88