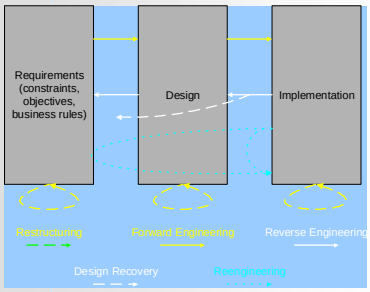
Three related topics

Reverse engineering
Visualization
Software summarization

Chikofsky & Cross taxonomy



Taxonomy

- Design recovery is a subset of reverse engineering
- The objective of design recovery is to discover designs latent in the software
  - These may not be the original designs, even if there were any explicit ones
  - They are generally recovered independent of the task faced by the developer
- It's a way harder problem than design itself

Restructuring

- One taxonomy activity is restructuring
- There are lots of reasons why people don't restructure in practice
  - 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

- It's not a local change nor a syntactic change
  - Based on PDGs, a rich program representation that captures both control and data flow
  - They are expensive to build
- Programmers only have to touch one place in the code

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

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

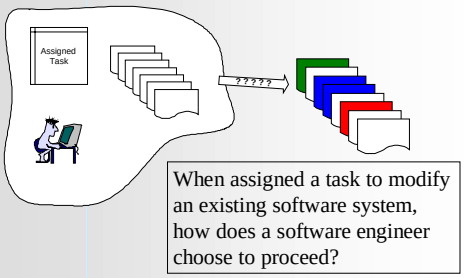
Evaluation

- 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

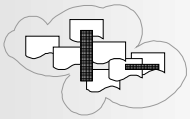
A view of maintenance



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

A task: isolating a subsystem

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



Mosaic

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

Mosaic source code

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

Some initial analysis

- 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)

How to proceed?

- 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
    - False positives, false negatives, etc.

Augment 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

Are we done?

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

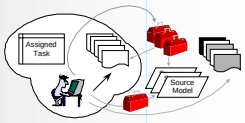
static function references (CIA)	3966
static function-global var refs (CIA)	541
dynamic function calls (gprof)	1872
Total	6379

One approach

- Use a query tool against the source model(s)
  - maybe grep?
  - maybe source model specific tool?
- As necessary, consult source code
  - "It's the source, Luke."
  - Mark Weiser. Source Code. *IEEE Computer 20*,11 (November 1987)

Other approaches

- Visualization
- Reverse engineering
- Summarization



Visualization

- e.g., Field, Plum, Imagix 4D, McCabe, etc. (Field's flowview is used above and on the next few slides...)
- Note: several of these are commercial products

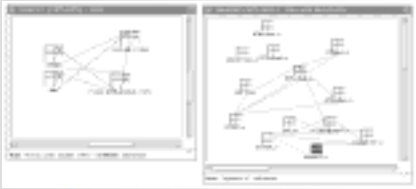
Visualization...

Visualization...

Visualization...

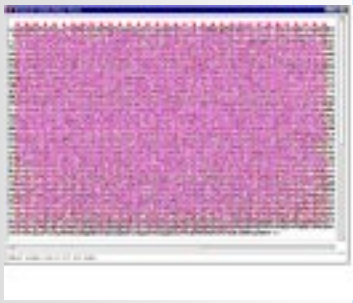
- provides a "direct" view of the source model
  - can produce a "precise" view
- view often contains too much information
  - use elision, producing an "optimistic" view
  - with elision you describe what you are not interested in, as opposed to what you are interested in

Reverse engineering



- e.g., Rigi, various clustering algorithms (Rigi is used above)
  - `http://www.rigi.csc.uvic.ca/rigi/rigiframe1.shtml`

Reverse engineering...



Clustering

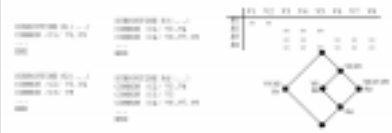
- The basic idea is to take one or more source models of the code and find appropriate clusters that might indicate "good" modules
- Coupling and cohesion, of various definitions, are at the heart of most clustering approaches
- Many different algorithms

Rigi's approach

- Extract source models (they call them resource relations)
- Build edge-weighted resource flow graphs
  - Discrete sets on the edges, representing the resources that flow from source to sink
- Compose these to represent subsystems
  - Looking for strong cohesion, weak coupling
- The papers define interconnection strength and similarity measures (with tunable thresholds)

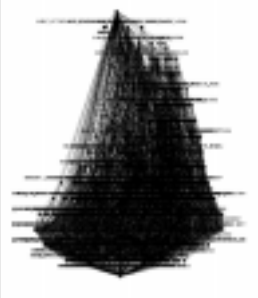
Math. concept analysis

- Define relationships between (for instance) functions and global variables [Snelting et al.]
- Compute a concept lattice capturing the structure
  - "Clean" lattices = nice structure
  - "ugly" ones = bad structure



An aerodynamics program

- 106 KLOC Fortran
- 20 years old
- 317 subroutines
- 492 global variables
- 46 COMMON blocks



Other concept lattice uses

- File and version dependences across C programs (using the preprocessor)
- Reorganizing class libraries
- Not yet clear how well these work in practice on large systems

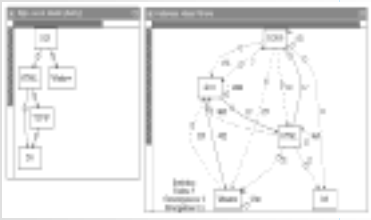
Reverse engineering recap

- Generally produces a higher-level view that is consistent with source
  - Like 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

More recap

- Automatic clustering approaches must try to produce "the" design
  - One design fits all
- User-driven clustering may get a good result
  - May take significant work (which may be unavoidable)
  - Replaying this effort may be hard
- Tunable clustering approaches may be hard to tune; unclear how well automatic tuning works

Summarization



- e.g., software reflexion models

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 ]
```


Summarization…

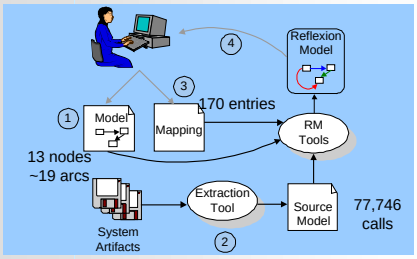
Summarization…

- 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

Case study: A task on Excel

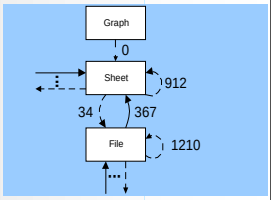
- A series of approximate tools were used by a Microsoft engineer to perform an experimental reengineering task on Excel
- The task involved the identification and extraction of components from Excel
- Excel comprises about 1.2 million lines of C source
  - About 15,000 functions spread over ~400 files

The process used

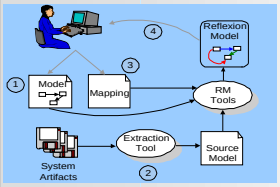


An initial Reflexion Model

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

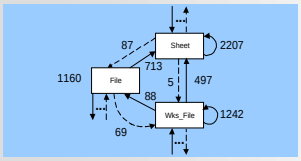


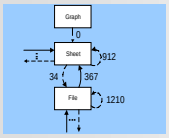
An iterative process



- Over a 4+ week period
- Investigate an arc
- Refine the map
  - Eventually over 1000 entries
- Document exceptions
- Augment the source model
  - Eventually, 119,637 interactions

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

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 A.B.C. (anonymous by choice) engineer

Open questions

- How stable is the mapping as the source code changes?
- Should reflexion models allow comparisons separated by the type of the source model entries?
- ...

Which ideas are important?

- Source code, source code, source code
- Task, task, task
  - The programmer decides where to increase the focus, not the tool
- Iterative, pretty fast
- A computation that the programmer fundamentally understands
  - Indeed, could do manually, if there was only enough time
- Graphical may be important, but also may be overrated in some situations

Wrap up

- Evolution 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
  - Sometimes tools can help with this structure, but it is often the intellectual structure that is more critical

Why is there a lack of tools to support evolution?

- Intellectual tools
- Actual tools