ELEC 875
Design Recovery
and
Automated Evolution

Documentary Structure
Van De Vanter - Background

- Michael Van De Vanter
- Works at Sun (now Oracle)
  ◊ programming environments
  ◊ editors
  ◊ code is in process of being edited
    - almost always ‘broken’

- I want to move to a more discussion based format starting today
Documentary Structure of Src Code

- Most tools based on formal structure of source code
  - linguistic structure
  - syntax trees
  - lexical structure
  - mimic compilers

- requires correct or at least (parseable) code
  - the formal linguistic part is what is executing
    - final authority on meaning of the system
  - Analysis of legacy code
Correct Parseable Code?

- Robust Parsing
  ◊ van Deursen and Kuipers (1999)
  ◊ Moonen (2001)
  ◊ Dean, Cordy, Malton and Schneider (2003)

- island grammars
  ◊ represent the grammar as interesting elements (islands) in a sea of water
  ◊ only the islands need be correct.
  ◊ concept nests (island may have lakes which may have islands …)
Island Grammar

define program
    [repeat element]
end define

define element
    [function]
    | [water]
end define

define water
    [token] | [key]
end define
Island Grammar

define function
    [id] [repeat ''] [id] '(', [repeat parm] ')
    [block]
end define

define parm
    [id] [repeat ''] [id] [repeat suffix]
end define
Island Grammar

define block
    '{
        [repeat body_element]
    '}
end define

define body_element
    [block]
    | [water]
end define
Island Grammar

- Find elements without parsing code
  - function headers
  - embedded sql
  - specific api calls (within limits)
  - distinct markers in syntax.
Documentary Structure

- Part of the program that is not formally part of the language
  
  - sole purpose is aiding the human reader
    - one of the main purpose of linguistic code is also human comprehension

- formatting
- comments
- inter token spacing
- line breaks
- Issues covered in Ugrad Soft Engineering
- Religious wars
Documentary Structure

• example: brace styles in C

```c
if () {
    ...
    K&R
}

if () {
    ...
    GNU
}

if () {
    ...
    BSD/Allman
}

if () {
    ...
    Whitesmith
}
```
Formal Language

- Documentary structure is outside of formal language
  - orthogonal
  - compilers discard information
    - biggerstaff minimized programs

- Source Code is a document
- Human as well as machine components
- Information that cannot be derived from semantics
  - similar to biggerstaff
Structural Mismatch

- Transformation and Restructuring tools have problems with comments and formatting.
- Since compilers have treated comments as whitespace, many different conventions to the use of comments:
  - many different ways to format comments
  - different ways of associating comments with code
  - almost any heuristic for transformation is bound to be wrong
- Syntax based editors failed in part because they tried to enforce specific commenting conventions.
Comments

• Notion of a single comment is not well defined
  ◦ comment boundaries
  ◦ white space in comments

• structural referent of a comment is not well defined
  ◦ comments placed in strange places

• Meaning of a comment depends on white space and natural language concerns
  ◦ subject changes in comments
Structural Referent

• Comments do refer to structural entities
  ◊ finding them are difficult for software
  ◊ easy for humans (noise ignored by humans).
  ◊ semantics of words involved

• Two dimensional concepts
  ◊ analysis software tends to be one dimensional

• Structural referents may be missing
  ◊ example in paper: while compilers throw away empty else clauses, many analysis tools keep them because they are important
Structural Referent?

const int hexVal[256] = {
    -1, -1, -1, -1, -1, -1, -1, -1, // null-bell
    -1, -1, -1, -1, -1, -1, -1, -1, // bs - si
    -1, -1, -1, -1, -1, -1, -1, -1, // dle - etb
    -1, -1, -1, -1, -1, -1, -1, -1, // can - us
    -1, -1, -1, -1, -1, -1, -1, -1, // sp ! " # $ % & '
    -1, -1, -1, -1, -1, -1, -1, -1, // ( ) * + , - . / 
    0,  1,  2,  3,  4,  5,  6,  7,     //  0 1 2 3 4 5 6 7
    8,  9, -1, -1, -1, -1, -1, -1,      //  8 9 : ; < = > ?
    -1, 10, 11, 12, 13, 14, 15, -1, // @ ABC D E F G
    -1, -1, -1, -1, -1, -1, -1, -1, // H I J K L M N O
    -1, -1, -1, -1, -1, -1, -1, -1, // P Q R S T U V W
    -1, -1, -1, -1, -1, -1, -1, -1, // X Y Z \ [ ] ^ _
    -1, 10, 11, 12, 13, 14, 15, -1, // ` a b c d e f g
    -1, -1, -1, -1, -1, -1, -1, -1, // h i j k l m n o
    -1, -1, -1, -1, -1, -1, -1, -1, // p q r s t u v w
    -1, -1, -1, -1, -1, -1, -1, -1, // x y z { | } ~ del
...
};

ELEC 875 – Design Recovery and Automated Evolution
Naming Convention

- CamelCase
- under_scores
- ALLCAPS

- Empirical studies have shown no real advantage to any.
  ◊ Consistent use is more important
  ◊ Use each one for a different type of id.
  – ALLCAPS for C defines
  – Java: Leading Cap for Class, leading lowercase for fields/methods
Approaches

• Hand crafted patches
• Automated (LS/2000)
• Unparsing
Van De Vanter

• Discussion
Analysis Graphs

- AST/ASG
- Control Flow Graph
- Data Dependency Graph

- Analysis technique: Slicing
AST/ASG

• AST - Abstract Syntax Tree
  ◊ Parse Tree based on an abstract grammar
  ◊ Not a compiler specific grammar

• ASG - Abstract Syntax Graph
  ◊ AST + edges
  ◊ edges from variable reference nodes back to variable declaration nodes
  ◊ edges from expression nodes to types to indicate types of operations
  ◊ invokes edges from call exprs to function defns
Control Flow Graphs

• Originally for compilers
  ◊ Basic Blocks - a sequence of statements with only one entrance and one exit
  ◊ edges between blocks represent control flow
  ◊ multiple edges at decision points (e.g. if)
  ◊ back edges for loops

• Analysis
  ◊ reachability

• Design Recovery
  ◊ Statements instead of basic blocks
Data Dependency/Flow Graphs

- Again, originally for compilers and basic blocks
- For design recovery, usually each node is a statement
- Edges represent a dependency on a value computed in a previous statement
- Good for impact analysis

\[
\begin{align*}
t1 &= 1; \\
t2 &= t1 + 3; \\
t3 &= 4; \\
t5 &= t1 + t3;
\end{align*}
\]
Slicing

- Mark Weiser (1981)
- Given a set of variables $v$ and a statement $p$,
  - The set of all statements that affect the values of the variables in $v$ at statement $p$
  - You have a hammer and you knock out any statement that doesn't affect the values
- a subset of the statements in a program
  - executable subset
- annotate the statement with the variables
  - move backwards in the data dependency graph annotating each node with a set of variables.
Slicing

- Static slicing - static analysis, based on if it is possible for the statement to affect the given variables.
- Dynamic slicing - those statements that affect the variables for a given set of inputs.

- Original motivation was for debugging.

- As described, called backwards slicing
  ♦ starting from $p$, all statements affected by $v$ is called a forward slice.
Concepts

• Concepts in comprehension research
  ◊ Václav Rajlich, Wayne State
    (one of founders of ICPC)

• An introductory survey of various research in the area
Concepts

• Fundamental block of human comprehension
  ◊ Important in learning
  ◊ attributes, lattice of concepts
  ◊ real world entities and classes of entities are concepts
    – cup, laptop, classroom, professor, student, conference
  ◊ actions are concepts too
    – travel, teaching, presenting a paper

◊ granularity
  – major concepts, minor concepts
Concepts and Software

- Play an import part in software
  - Object Oriented
  - not all concepts are objects
  - granularity
  - entities vs actions
  - central concepts/distributed concepts

- SA&D
  - central data structures are major concepts
  - actions are major software components
Concepts and Maintenance

• Concepts for software change over time
  ◊ Unexpected use of software
    – consequential requirements
  ◊ Change in Technology
    – batch to online
    – privileged online to consumer online
Concepts and Maintenance

• Programmers understand domain concepts
  ◊ real time systems, event driven systems, transactions, etc.
    – on-the-job training?
  ◊ many domain concepts are user concepts
    – easier to learn
  ◊ change requests are often in terms of domain concepts

◊ Program comprehension is identifying where the concepts are represented in the code.
Concepts Location

- Always done
  - informally in many cases
    - similar to Lethbridge & Singer
  - Sometimes easy and intuitive
    - fall back to searching tools
    - grep
  - link between naming conventions and concepts
    - date variable names involve ‘date’ or date words
    - customer variable names involve ‘cust’ or customer words

- doesn’t always work
Concepts Location Problems

• Link between concept and names
  ◊ language
    – mmddyy vs aammjj
  ◊ Names of concepts change in different environments
    – IPL vs Boot
    – Sysgen
  ◊ Concept terminology changes over time
    – father/son vs. parent/child
    – classes of phone numbers
Concepts Location Strategies

- Dynamic
  - execution traces
    - instrumentation (profiling)
    - analysis of input grammar used to identify test cases
- Static
  - static tracing
  - smart code searching
Case Studies

• NCSA Mosaic
  ◊ add audio files
    – 3 parts: open file, mapping, global vars based used by mapping routines
  ◊ partial comprehension - 2% of code visited

• ATAC test coverage monitor (Bellcore)
  ◊ showed that concepts delocalized
  ◊ 19 of 24 concepts had code in two or more source files
  ◊ regularity of naming
Domain Knowledge from Code

• Detailed design information
  ◊ often only documented in the code
    – bank gets sued for improper foreclosure, memo from legal “not to do this again”
  ◊ issue for reimplementation

• Case Study
  ◊ Fortran modelling system
  ◊ breaks solids into polygons
  ◊ older obsolete problems (file system optimization, scratch files, etc).
Other Work

• Change impact analysis
  ◊ what happens if I change this line??
  ◊ traceability from design documents to code and back

• Fault Location
  ◊ smarter debugging