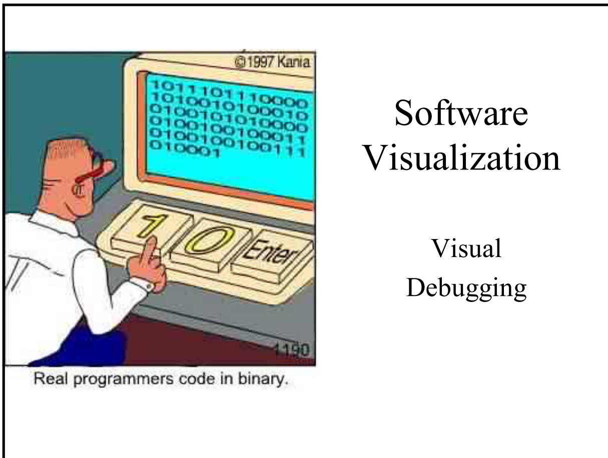


Real programmers code in binary.

# Software Visualization

Visual

Debugging

## Visual Debugging

- What is „Debugging“
- Visualizing
  - Program State
    - Incremental interactive unfolding (DDD)
    - Abstract representations (traversal-based)
    - Focusing: memory graphs
    - Reference pattern extraction
  - Test Results (test suites)
    - Dices
    - Participation in coverage and failures

} Memory (Program State)

} Code (Test Results)

## Debugging

- Debugging
  = Detecting, locating and fixing errors in programs
- Common Tasks [see Pan&DeMillo&Spafford:97]
  - Identify statements involved
  - Select statements which might contain faults
  - Hypothesize about suspicious faults
  - Restore program variables to a specific state

## Data Display Debugger DDD

- Visualize Program State
- Interactive debugger
  - Execute program in a defined environment
  - Stop execution at specified situations (conditional break points)
  - Inspect program state
  - Modify program state and continue execution

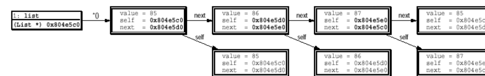
See [Zeller:IFUE01,Zeller&Lütkehaus:96]

## Nested Boxes

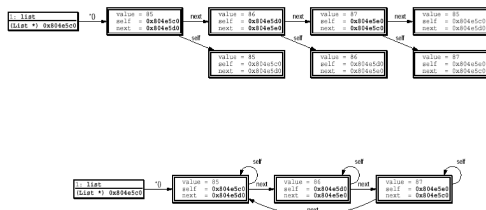
(gdb) **display *tree**
*tree = { fvalue = 7, _name = 0x8049e88 "Ada",
_left = 0x804d7d8, _right = 0x0,
_left_thread = false, right_thread = false,
date = {day of week = Thu, day = 1, month = 1, year = 1970,
_vptr. = 0x8049f78 <Date virtual tablei>},
static shared = 4711}
(gdb) _

```
1: *tree
value        = 7
_name        = 0x8049e88 "Ada"
_left        = 0x804d7d8
_right       = 0x0
left_thread  = false
right_thread = false
                 day_of_week = Thu
                 day         = 1
date         =   month       = 1
                 year        = 1970
                 _vptr.      = 0x8049f78 <Date virtual table>
shared       = 4711
```

## Incremental, Interactive Unfolding of Data Structures

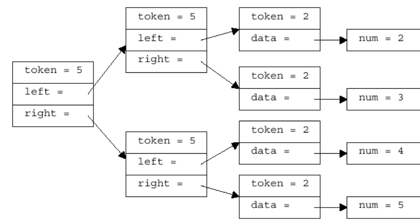


## Aliasing Detection → Sharing Nodes



## Traversal-based Visualization

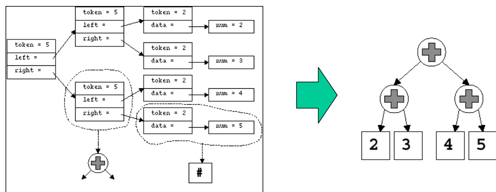
- Typical visualization produced by a visual debugger




http://www.cs.princeton.edu/~jlk/viz

## Traversal-based Visualization

- Traverse linked data structure
- Match found data with rules:
  - rules produce visual objects (model),
  - these are then rendered by a separate component.



2 3 4 5

## Example Rule

Class of Objects to apply rule to

Name of rule

Pattern to match object with

```
Op plusPattern =
  { int op = Op.PLUS; } :
     node=TreeNode(icon="plus.bmp"),
     TreeEdge(from=parent, to=node),
      → plusPattern.left(parent=node),
      → plusPattern.right(parent=node);
```

Create objects of visual model

Traverse referenced objects, pass node in environment

Here: class Op { final static int PLUS=1; int op; Expr left, right ...}

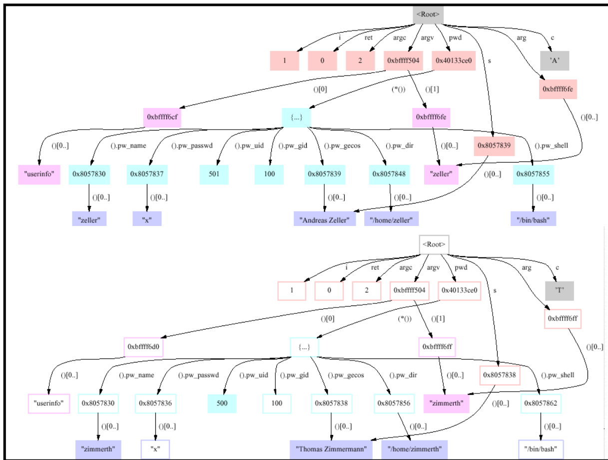
class TreeNode { String icon; ... }
class TreeEdge { TreeNode from, to; ... }

## What if the data structures to be visualized are really large?

- Focus on modified parts (c.f. abstract algorithm animation)
- Group elements, form collections of data with similar structure.

## Memory Graphs

- represent the memory of a program.
- *Nodes* = memory content
- *Arrows* = possible access paths.
- unfolding **all** accessible data structures in the program. All common data structures like structs, unions, arrays or pointers are properly represented.
- Memory graph of GNU compiler has about 40.000 nodes !!!
- **Applications**
  - *common subgraphs* to isolate differences between program states.



# Reference Pattern Extraction

Jinsight (http://www.research.ibm.com/jinsight/)

- At each level unfolding groups objects of the same class together.

# Slicing

- Static slice = set of all program points that may affect the value of a particular output or a instance of a variable at a certain program point. → data flow analysis
- Dynamic slice = set of all program points that for a given input actually affect a program point or instance of a variable at a certain program point.
- Execution slice = set of all program points executed for a given input.
- Dice = set difference of two slices

# X-Slice

http://xsuds.argreenhouse.com/html-man/xslice.html#770301

- X-Slice is a slicing and dicing tool for C programs.
- the dice represents the intersection of a failing test case and a sucessful test case.

Code in red is executed by the failed test *sort.4* but not the successful test *sort.3*.

This is the most likely location of the fault.

# Coverage and Tests

**Program execution:**
run : $L_{G_{Simp}}(S) \times I \rightarrow O$
$O$ : output values or final states
$I$ : input values or start states

**Test Suite:**
$T \subseteq I \times O$ and each pair $(in, out) \in T$ is a test case.

**Coverage (execution slice):**
$\text{coverage}(s, in) = \{p \mid \text{ program point } p \text{ is executed for input } in\}$

**Actual test runs:**
$\text{passed}(s, T) = \{(in, out) \in T \mid \text{run}(s, in) = out\}$
$\text{failed}(s, T) = \{(in, out) \in T \mid \text{run}(s, in) \neq out\}$

$\text{passed}(p, s, T) = \{(in, out) \in \text{passed}(s, T) \mid p \in \text{coverage}(s, in)\}$
$\text{failed}(p, s, T) = \{(in, out) \in \text{failed}(s, T) \mid p \in \text{coverage}(s, in)\}$

**Relative to all failed or to all successful runs:**
$\%\text{passed}(p, s, T) = \frac{|\text{passed}(p,s,T)|}{|\text{passed}(s,T)|}$ $\%\text{failed}(p, s, T) = \frac{|\text{failed}(p,s,T)|}{|\text{failed}(s,T)|}$

# Discrete Approach

- Input
  - Source code
  - For each test case
    - its pass/fail status
    - statements that it executes
- Display statements in program according to the test cases that execute them

Statements executed by: Only failed test cases — Both passed & failed test cases — Only passed test cases

# Example

Test Cases

| | Code | 3,3,5 | 1,2,3 | 3,2,1 | 5,5,5 | 5,3,4 | 2,1,3 |
|---|---|---|---|---|---|---|---|
| | `mid() {` | | | | | | |
| | `int x,y,z,m;` | | | | | | |
| 1: | `read("Enter 3 numbers:",x,y,z);` | • | • | • | • | • | • |
| 2: | `m = z;` | • | • | • | • | • | • |
| 3: | `if (y<z)` | • | • | • | • | • | • |
| 4: | `if (x<y)` | | • | | | | |
| 5: | `m = y;` | | • | | | | |
| 6: | `else if (x<z)` | • | | | | • | • |
| 7: | `m = y;` | • | | | | | • |
| 8: | `else` | • | | • | • | | |
| 9: | `if (x>y)` | | | • | | | |
| 10: | `m = y;` | | | • | | | |
| 11: | `else if (x>z)` | | | | | | |
| 12: | `m = x;` | | | | | | |
| 13: | `print("Middle number is:", m);` | • | • | • | • | • | • |
| | `}` | | | | | | |
| | Pass Status: | P | P | P | P | P | F |

# Example

Test Cases

| | Code | 3,3,5 | 1,2,3 | 3,2,1 | 5,5,5 | 5,3,4 | 2,1,3 |
|---|---|---|---|---|---|---|---|
| | `mid() {` | | | | | | |
| | `int x,y,z,m;` | | | | | | |
| 1: | `read("Enter 3 numbers:",x,y,z);` | • | • | • | • | • | • |
| 2: | `m = z;` | • | • | • | • | • | • |
| 3: | `if (y<z)` | • | • | • | • | • | • |
| 4: | `if (x<y)` | | • | | | | |
| 5: | `m = y;` | | • | | | | |
| 6: | `else if (x<z)` | • | | | | • | • |
| 7: | `m = y;` | • | | | | | • |
| 8: | `else` | • | | • | • | | |
| 9: | `if (x>y)` | | | • | | | |
| 10: | `m = y;` | | | • | | | |
| 11: | `else if (x>z)` | | | | | | |
| 12: | `m = x;` | | | | | | |
| 13: | `print("Middle number is:", m);` | • | • | • | • | • | • |
| | `}` | | | | | | |
| | Pass Status: | P | P | P | P | P | F |

# Continuous Approach

- Distribute statements executed by both passed and failed test cases over spectrum
- Indicate the relative success rate of each statement by its hue

Only failed test cases

Both passed & failed test cases

Only passed test cases

Discrete Approach:

Continuous Approach:

# Hue

$\frac{10 \text{ failed test cases}}{10} = 100\%$

$\frac{0 \text{ passed test cases}}{10} = 0\%$

`a = b;`

$\frac{1 \text{ failed test cases}}{10} = 10\%$

$\frac{0 \text{ passed test cases}}{10} = 0\%$

`c = d;`

$\frac{10 \text{ failed test cases}}{10} = 100\%$

`m = y;`

$\frac{20 \text{ passed test cases}}{200} = 10\ \%$

$$\text{hue}(s) = \text{low hue (red)} + \frac{\%\text{passed}(s)}{\%\text{passed}(s) + \%\text{failed}(s)} * \text{hue range}$$

# Brightness

- Using total percentage of test cases that execute a statement may cause important statements to be overlooked
- Instead we use the higher of the two independent percentages

$\frac{10 \text{ failed test cases}}{10} = 100\%$

`m = y;`

$\frac{20 \text{ passed test cases}}{200} = 10\ \%$

$\frac{30 \text{ total test cases}}{210} = 14\%$

$$\text{bright}(s) = \max(\%\ \text{passed}(s), \%\ \text{failed}(s))$$

# Brightness

$\frac{10 \text{ failed test cases}}{10} = 100\%$

$\frac{0 \text{ passed test cases}}{10} = 0\%$

`m = y;`

$\frac{1 \text{ failed test cases}}{10} = 10\%$

$\frac{0 \text{ passed test cases}}{10} = 0\%$

`m = y;`

$\frac{10 \text{ failed test cases}}{10} = 100\%$

`m = y;`

$\frac{20 \text{ passed test cases}}{200} = 10\ \%$

# Example

Test Cases

| | Code | 3,3,5 | 1,2,3 | 3,2,1 | 5,5,5 | 5,3,4 | 2,1,3 |
|---|---|---|---|---|---|---|---|
| | `mid() {` | | | | | | |
| | `int x,y,z,m;` | | | | | | |
| 1: | `read("Enter 3 numbers:",x,y,z);` | • | • | • | • | • | • |
| 2: | `m = z;` | • | • | • | • | • | • |
| 3: | `if (y<z)` | • | • | • | • | • | • |
| 4: | `if (x<y)` | | • | | | | |
| 5: | `m = y;` | | • | | | | |
| 6: | `else if (x<z)` | • | | | | • | • |
| 7: | `m = y;` | • | | | | | • |
| 8: | `else` | • | | • | • | | |
| 9: | `if (x>y)` | | | • | | | |
| 10: | `m = y;` | | | • | | | |
| 11: | `else if (x>z)` | | | | | | |
| 12: | `m = x;` | | | | | | |
| 13: | `print("Middle number is:", m);` | • | • | • | • | • | • |
| | `}` | | | | | | |
| | Pass Status: | P | P | P | P | P | F |

# Scalability

- Large programs difficult to display
- Use the line-of-pixels, SeeSoft, view
- Each character in the source is displayed as a pixel

```
mid() {
  int x,y,z,m;
  read("Enter 3 numbers:",x,y,z);
  m = z;
  if (y<z)
    if (x<y)
      m = y;
    else if (x<z)
      m = y;
  else
    if (x>y)
      m = y;
    else if (x>z)
      m = x;
  print("Middle number is:", m);
}
```

[Eick,Steffen,Sumner, TSE 1992]

# Tarantula

# Visualization Pipeline

| | Memory | Code |
|---|---|---|
| Data Acquisition | Read Program Memory | Trace program execution, visited program points, success or failure |
| Filtering | Alias Detection, Common Subgraph of Memory graphs | Slicing and Dicing |
| Visualization | Text, Nested Boxes, Graphs | Program Code,Color coding, „SeeSoft“ Approach |