Program Understanding and Visualization

Understanding other people's code

Thanks for their slides!

Tom Zimmermann
Microsoft Research

Michele Lanza

Professor University of Lugano

for (a=a; a>=a && a[a-a]>a; a-=a)

```
public static void a(int[] a) {
  int a, a;
  int a = 1;
  do {
    a = a * 3 + 1;
  } while (a <= a.length);
  do {
    a /= 3;
    for (a = a; a < a.length; a++){
        int a = a[a];
      for (a=a; a>=a&&a[a-a]>a; a-=a)
        a[a] = a[a - a];
      if (a != a)
        a[a] = a;
    }
  } while (a != 1);
}
```

What does method a() compute?
Does it compile at all?

TYPES OF SOFTWARE DEVELOPMENT PROJECTS

1. Green field projects → new development

   Our primary focus.

2. Evolutionary projects → maintenance

   Most common type.

   – corrective – fix defects

   – adaptive – adapt to new technology, new laws, etc.

   – enhancement – add new features

   – perfective (re-engineering) – make more maintainable

3. Framework/component projects → reuse

   – Use an existing framework or plug together several existing components.

   A framework is a software system designed specifically to be reused in different projects or a product line, but needing to be adapted to handle specific requirements.

Simplified overview of software maintenance

- **Understanding** the current system
- Carrying out the change
- Testing

Program understanding

- What does the software system do? (general knowledge)
- Where in the system are changes needed?
- How do these parts to be corrected or modified work? (in-depth knowledge)
- It is an essential part of development

Today

- How we understand software?
- How can we make our software understandable?

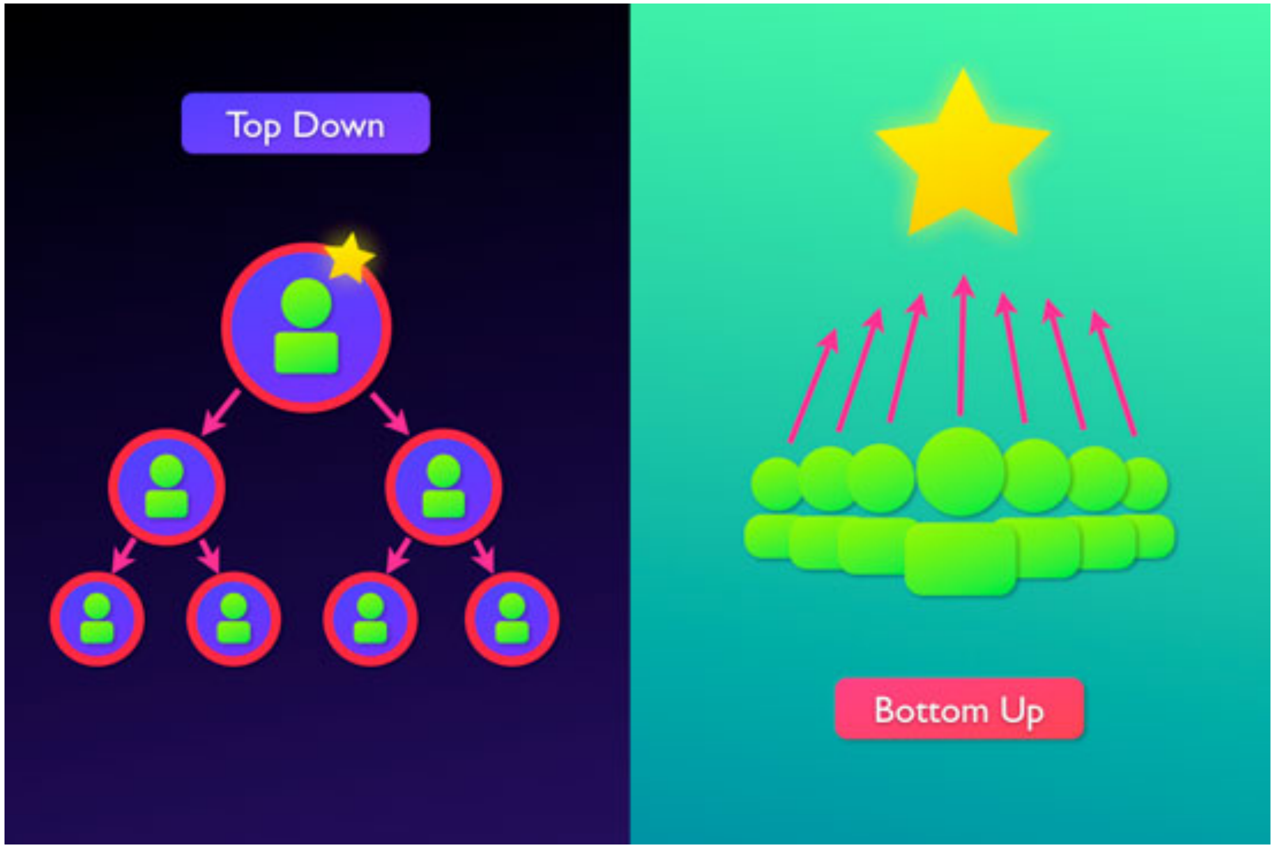
Program understanding

- Manual inspection
  - Static: read, read, and read
  - Dynamic approach
- Class diagrams
- Dependancies
- Program slicing
- Visualization

Static inspection: reading

- What to read?
  - Code
  - Naming convention
  - Comments
  - Documents (if you are lucky)

Strategies



http://jurmo.us/2007/04/02/top-down-bottom-up/

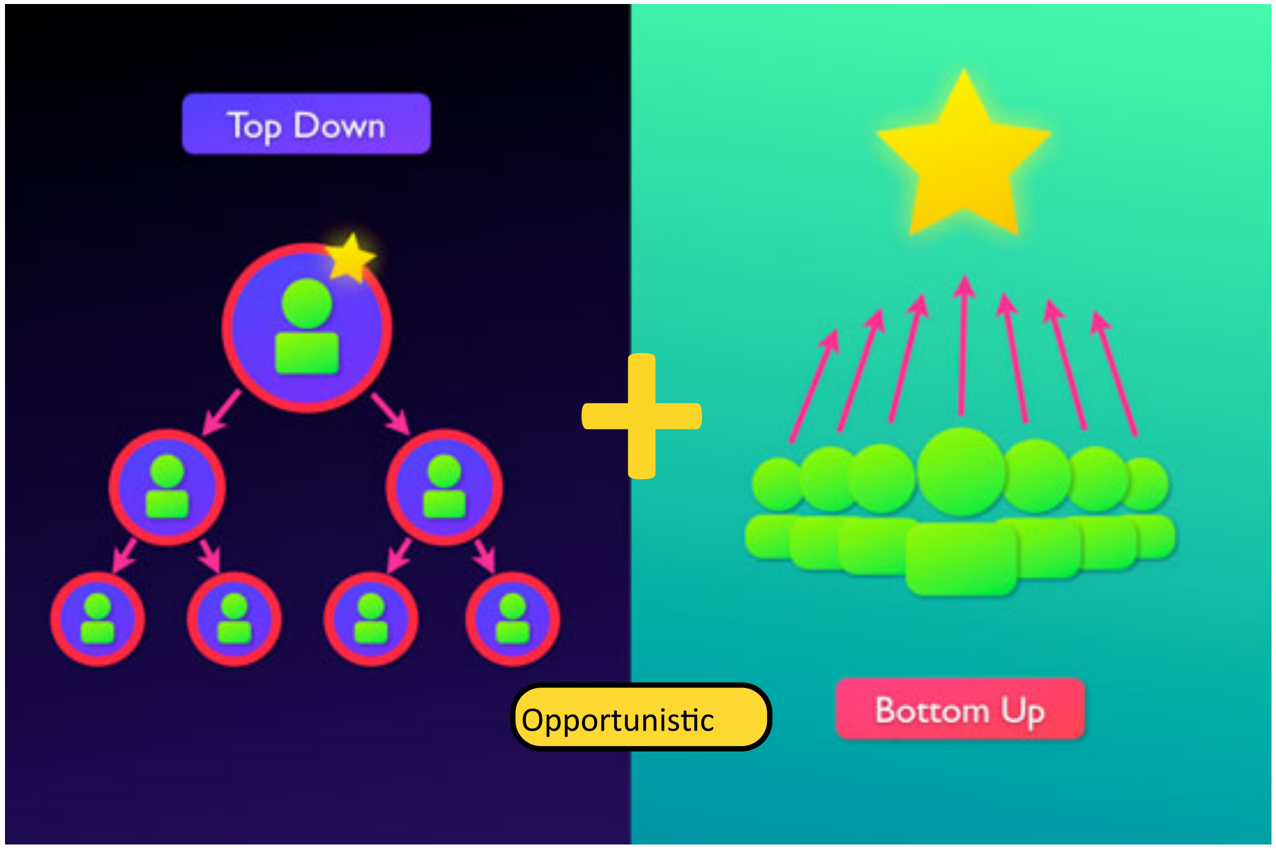
Strategies for program comprehension

- Top-down model (Brooks, 1983)
  - start at top-level details (e.g., what is executed when) and gradually work towards understanding low-level details
- Bottom-up/Chunking model
  - recognize patterns in the program
  - iteratively group them into high-level, semantically more meaningful structures

Strategies for program comprehension

- Weaknesses of top-down & bottom-up model
  - do not take other factors into account, such as available tool support
  - people rarely understand a program in such a well-defined fashion
  - programmers take ANY clue they get $\Rightarrow$ opportunistic way

Strategies for program comprehension



Based on http://jurmo.us/2007/04/02/top-down-bottom-up/

Strategies for program comprehension

- Opportunistic model
  - knowledge base: the expertise and background of the maintainer
  - mental model: the current understanding of the target program—model is updated continuously
  - assimilation process: obtain information from various sources using various techniques

"...the human understander is best viewed as an opportunistic processor capable of processing both bottom-up and top-down cues as they become available" —Letovsky, 1986

Program understanding

- Manual inspection
  - Static: read, read, and read
  - Dynamic approach
- Class diagrams
- Dependancies
- Program slicing
- Visualization

Program execution

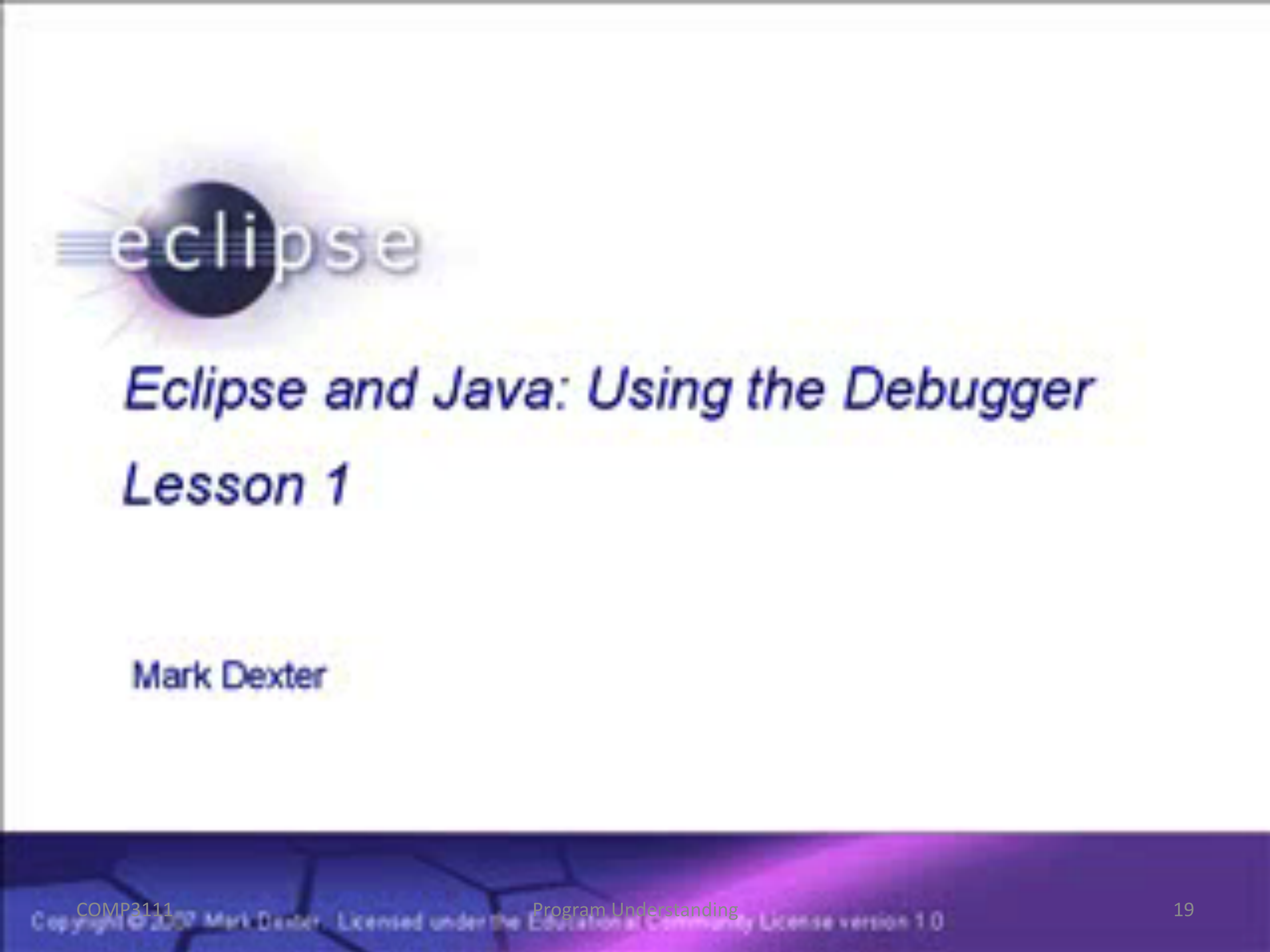
- Run and look inside program behaviors
- printf(“Hello!”);
- Debugger

Debugger

- Be familiar with them
- Step execution
- Observe variables

Eclipse debugger - quick demo

http://www.youtube.com/watch?v=WeSitNPAExg



eclipse

Eclipse and Java: Using the Debugger

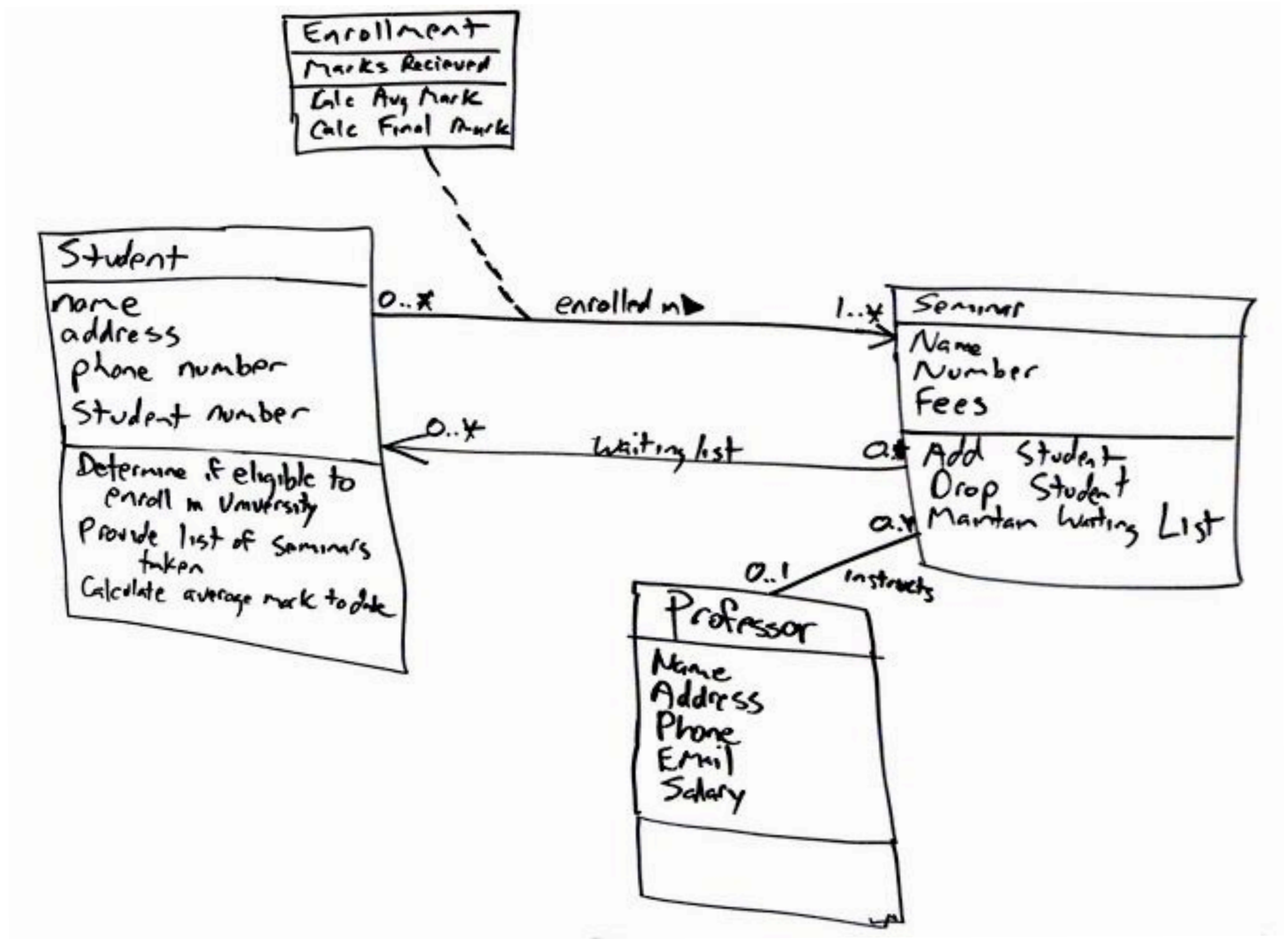
Lesson 1

Mark Dexter

Program understanding

- Manual inspection
  - Static: read, read, and read
  - Dynamic approach
- Class diagrams
- Dependancies
- Program slicing
- Visualization

Class diagrams



http://www.agilemodeling.com/images/models/classDiagramSketch.JPG

Tool support

- Draw class diagrams
- Java to class diagrams

Program understanding

- Manual inspection
  - Static: read, read, and read
  - Dynamic approach
- Class diagrams
- Dependancies
- Program slicing
- Visualization

Program dependencies

A key to program understanding is unravelling the interrelationships of program components. This module discusses the different methods and tools that aid a programmer in answering the questions "How does this system fit together?" and "If I change this component, what other components might be affected?"

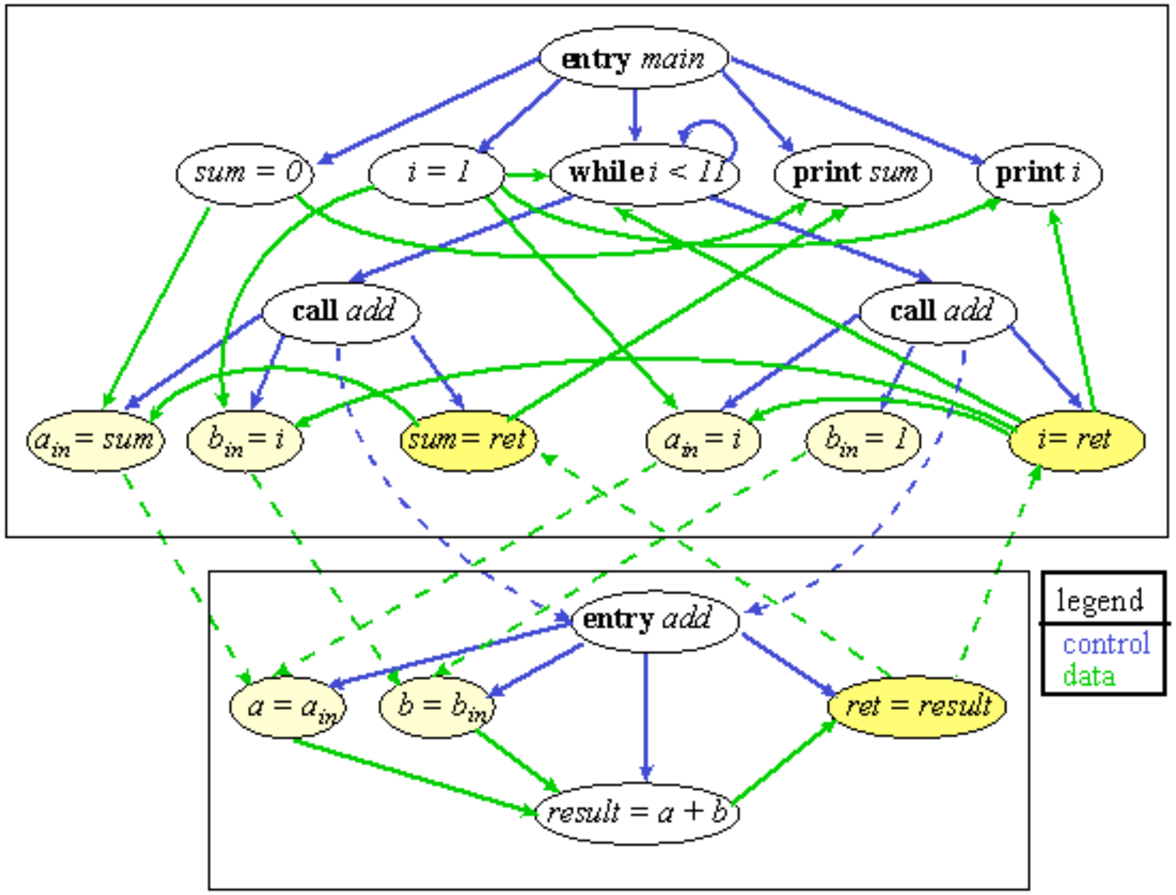
- Norman Wilde

Program dependencies

- Control - control flow graph (CFG)
- Data - data dependency graph
- Call graphs

Program dependencies

```
void main() {
    int i = 1;
    int sum = 0;
    while (i<11) {
        sum = add(sum, i);
        i = add(i, 1);
    }
    printf("sum = %d\n", sum);
    printf("i = %d\n", i);
}
static int add(int a, int b)
{
    return(a+b);
}
```

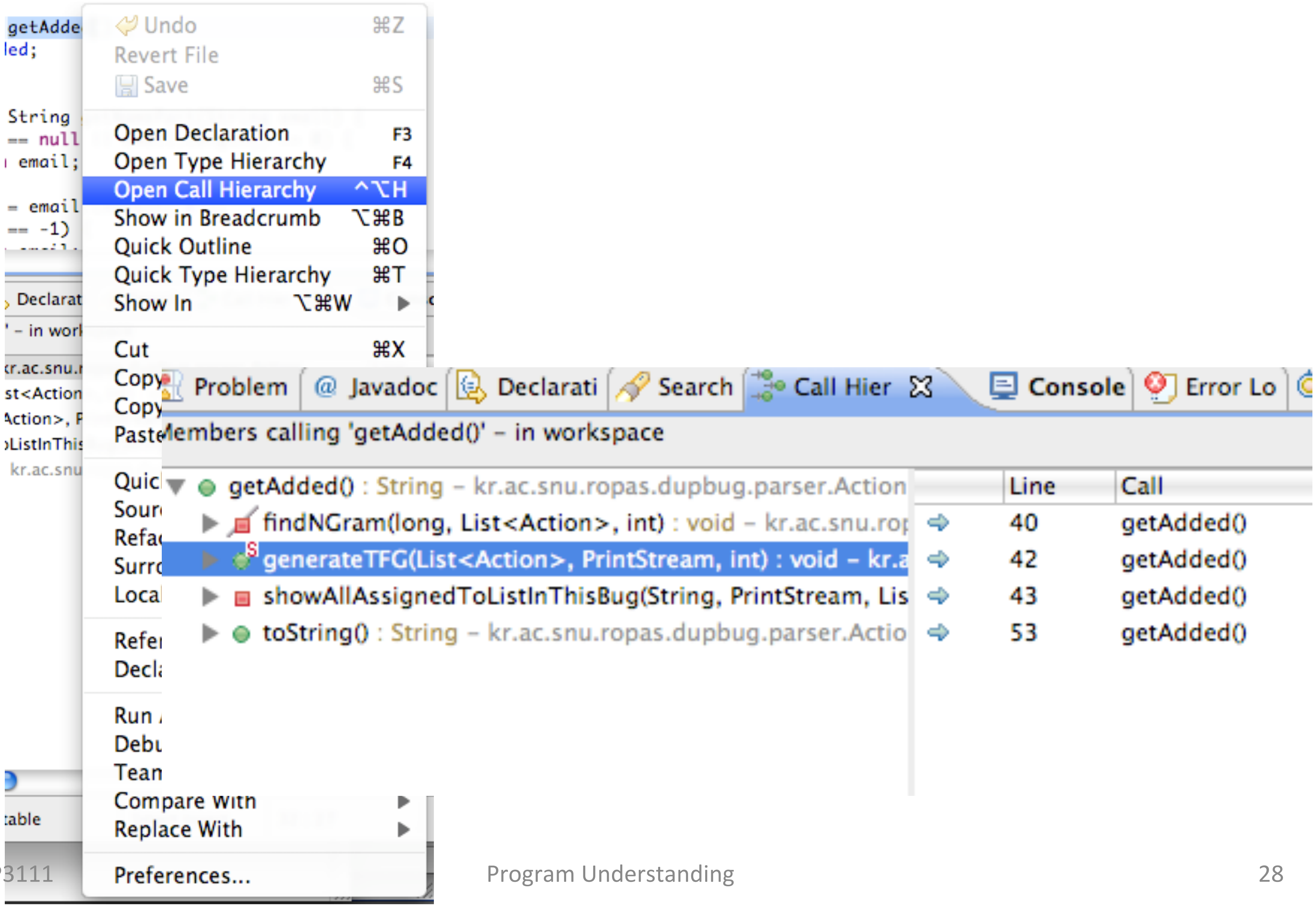


http://www.grammatech.com/research/papers/slicing/slicingWhitepaper.html

Call graphs

- method invocation relationships
- Tool supports

Call Hierarchy

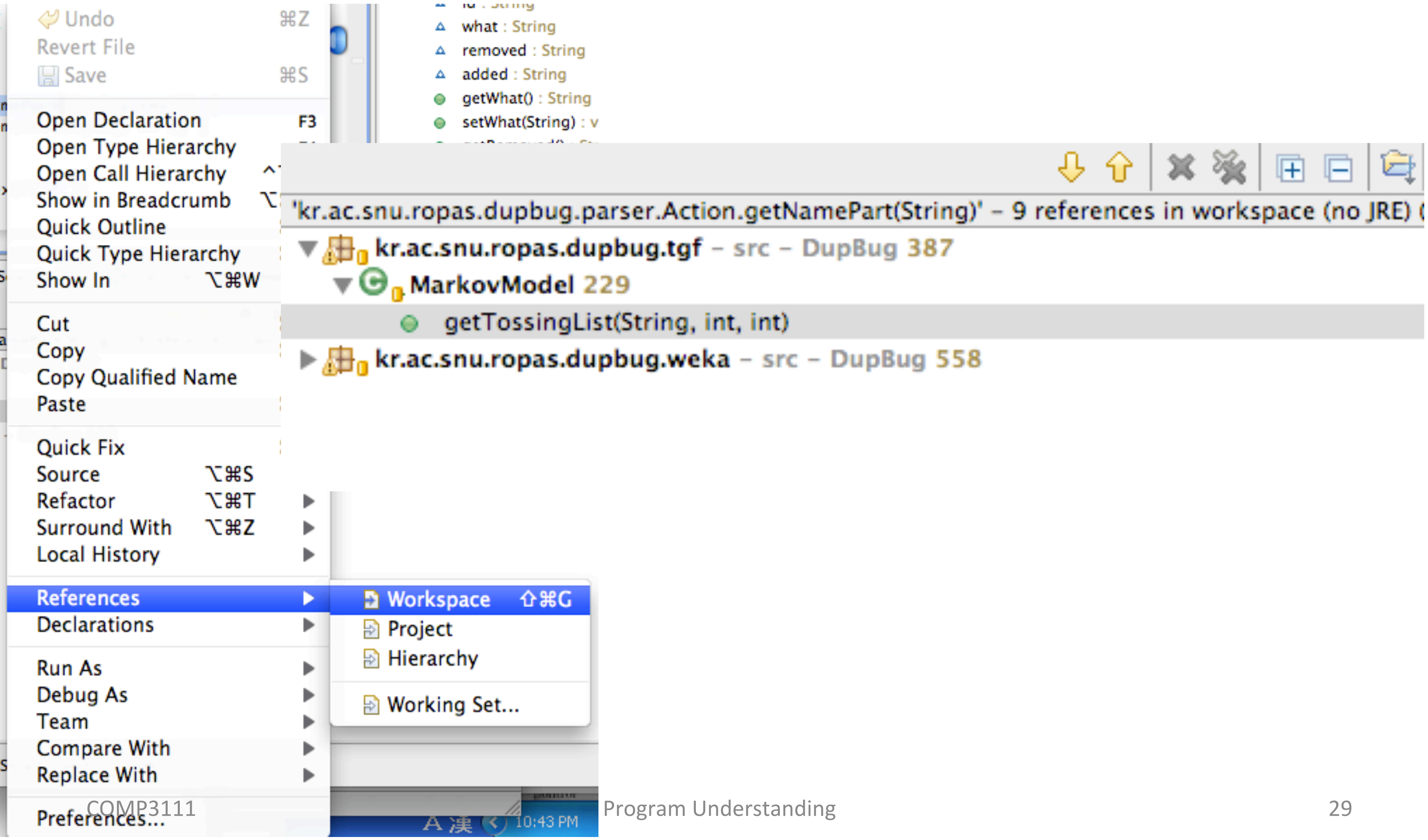


Members calling 'getAdded()' – in workspace

```
getAdded() : String – kr.ac.snu.ropas.dupbug.parser.Action
  findNGram(long, List<Action>, int) : void – kr.ac.snu.rop
  generateTFG(List<Action>, PrintStream, int) : void – kr.a
  showAllAssignedToListInThisBug(String, PrintStream, Lis
  toString() : String – kr.ac.snu.ropas.dupbug.parser.Actio
```

Line	Call
40	getAdded()
42	getAdded()
43	getAdded()
53	getAdded()

References



'kr.ac.snu.ropas.dupbug.parser.Action.getNamePart(String)' – 9 references in workspace (no JRE)

```
▼ kr.ac.snu.ropas.dupbug.tgf – src – DupBug 387
  ▼ MarkovModel 229
      getTossingList(String, int, int)
▶ kr.ac.snu.ropas.dupbug.weka – src – DupBug 558
```

Program understanding

- Manual inspection
  - Static: read, read, and read
  - Dynamic approach
- Class diagrams
- Dependancies
- Program slicing
- Visualization

Program slicing

- See what you want to see
- 1981 by Mark Weiser

Program slicing

```
void main() {
     int i = 1;

     int sum = 0;
     while (i<11) {
          sum = add(sum, i);
          i = add(i, 1);
     }
     printf("sum = %d\n",sum);
     printf("i = %d\n", i);
}

static int add(int a, int b)
{
     return(a+b);
}
```

```
void main() {
     int i = 1;

     int sum = 0;
     while (i<11) {
          sum = add(sum, i);
          i = add(i, 1);
     }
     printf("sum = %d\n", sum);
     printf("i = %d\n", i);
}

static int add(int a, int b)
{
     return(a+b);
}
```

Program slicing

- Static slicing
- Dynamic slicing

Program understanding

- Manual inspection
  - Static: read, read, and read
  - Dynamic approach
- Class diagrams
- Dependancies
- Program slicing
- Visualization

Software visualization

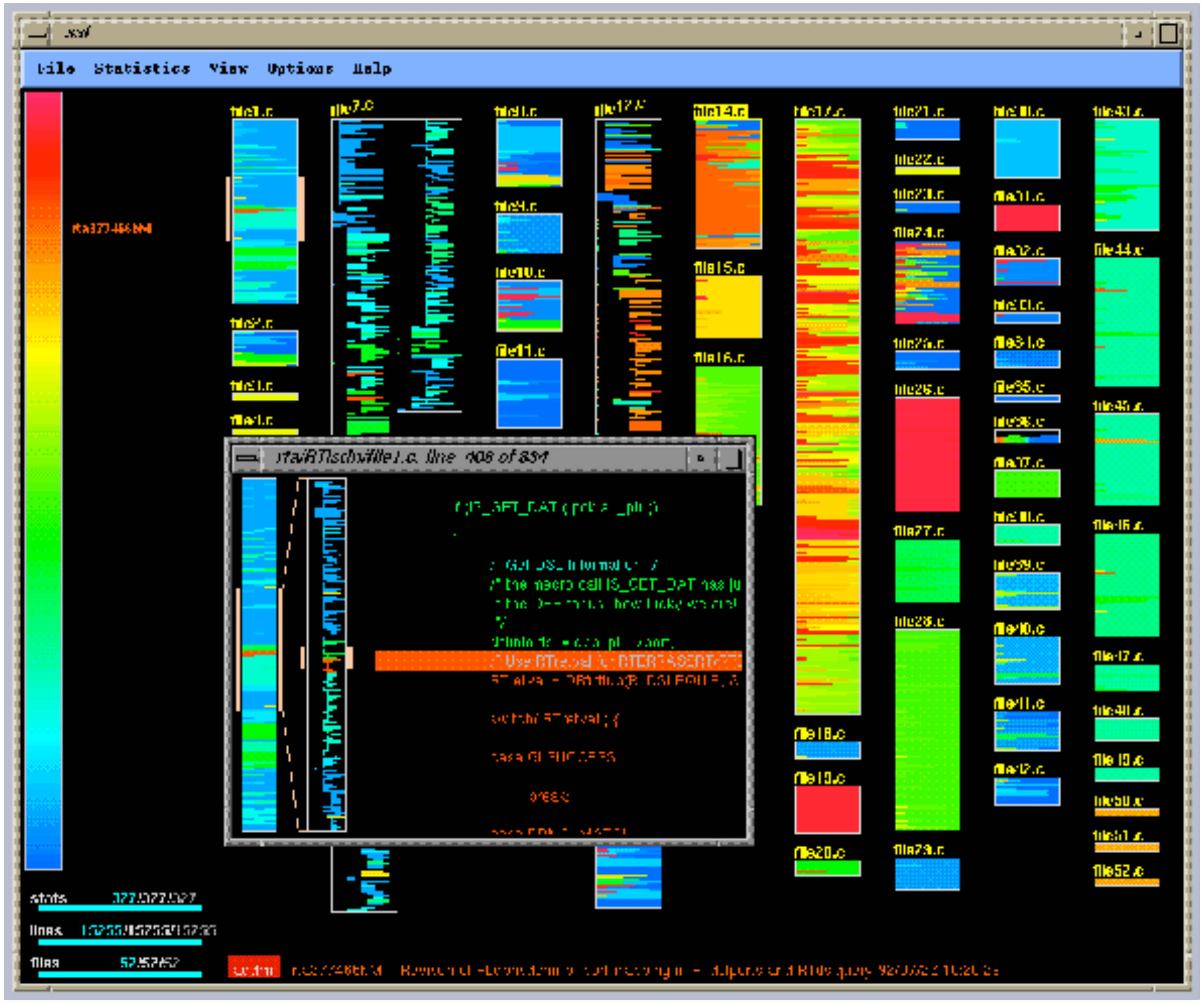
Software is intangible,
having no physical shape or size.

- Thomas Ball

Software visualization

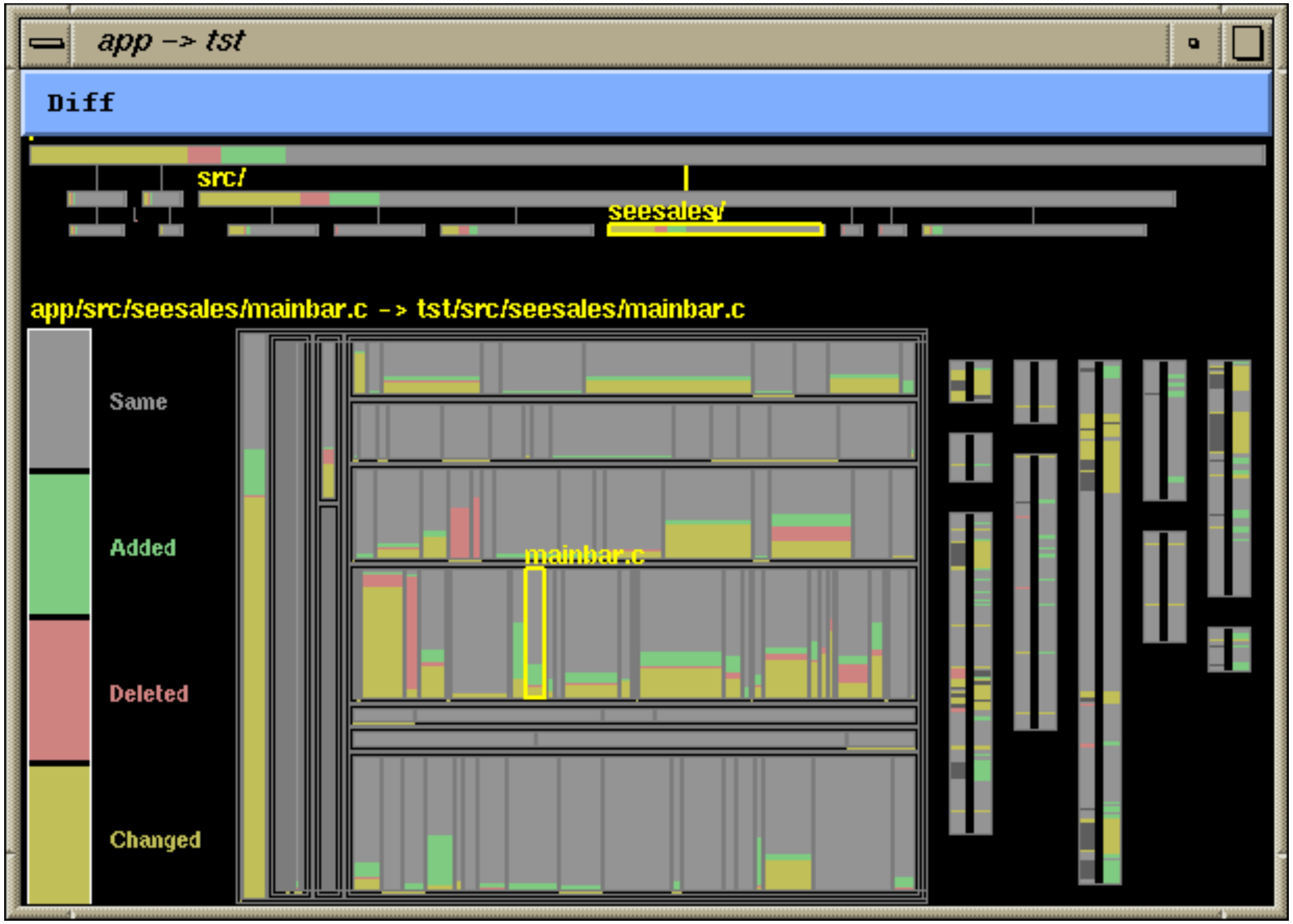
- SeeSoft (the pioneer!)
- Xray
- CodeCity

SeeSoft



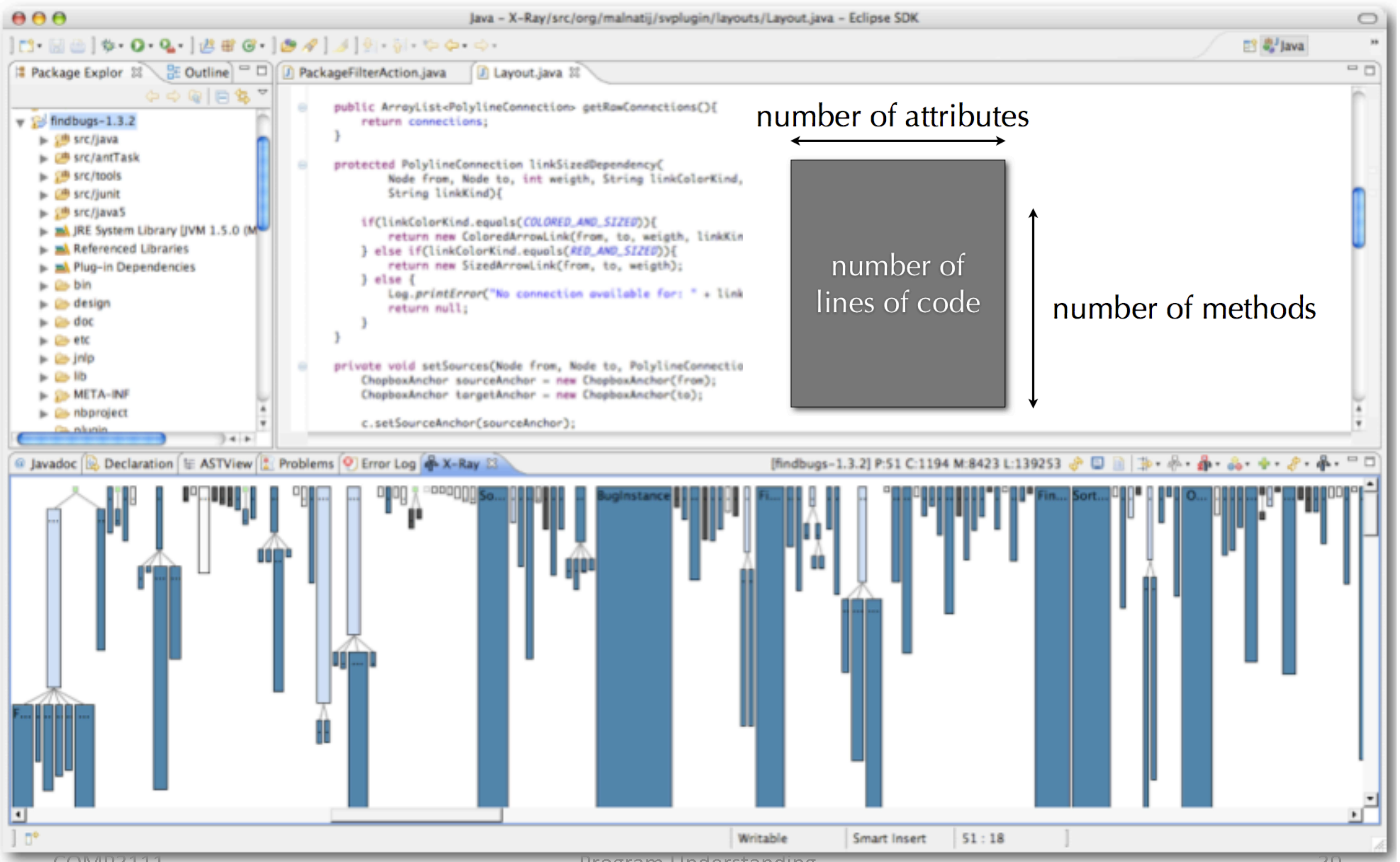
Eick, S. G., Steffen, J. L., and Sumner, E. E. 1992. Seesoft-A Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. Softw. Eng.* 18, 11 (Nov. 1992), 957-968. DOI= http://dx.doi.org/10.1109/32.177365

SeeSoft



Eick, S. G., Steffen, J. L., and Sumner, E. E. 1992. Seesoft-A Tool for Visualizing Line Oriented Software Statistics. *IEEE Trans. Softw. Eng*. 18, 11 (Nov. 1992), 957-968. DOI= http://dx.doi.org/10.1109/32.177365

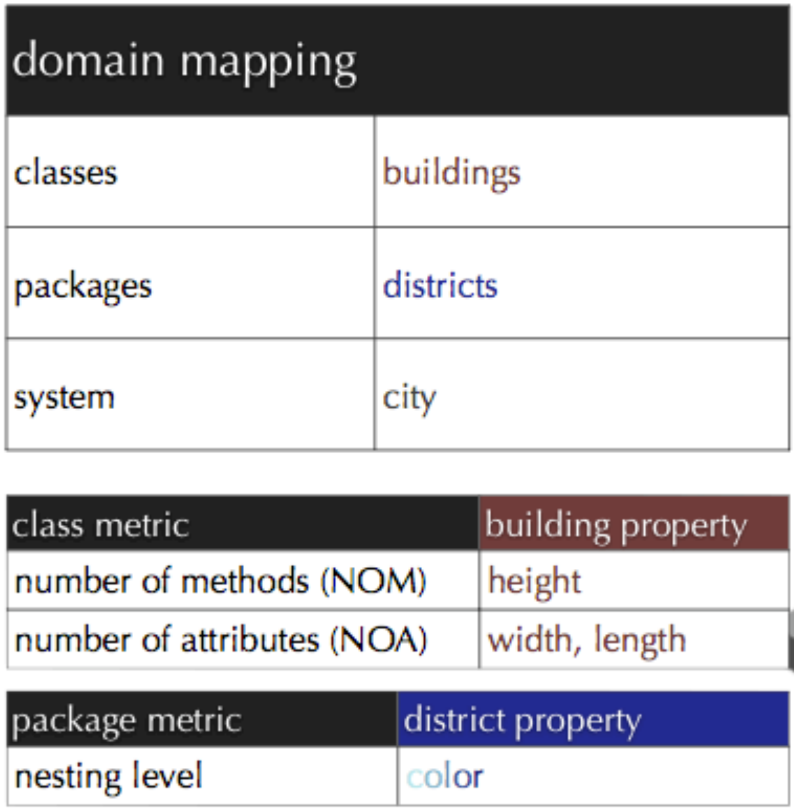
XRay



```
    public ArrayList<PolylineConnection> getRowConnections(){
        return connections;
    }

    protected PolylineConnection linkSizedDependency(
            Node from, Node to, int weigth, String linkColorKind,
            String linkKind){

        if(linkColorKind.equals(COLORED_AND_SIZED)){
            return new ColoredArrowLink(from, to, weigth, linkKin
        } else if(linkColorKind.equals(RED_AND_SIZED)){
            return new SizedArrowLink(from, to, weigth);
        } else {
            Log.printError("No connection available for: " + link
            return null;
        }
    }

    private void setSources(Node from, Node to, PolylineConnectio
        ChopboxAnchor sourceAnchor = new ChopboxAnchor(from);
        ChopboxAnchor targetAnchor = new ChopboxAnchor(to);

        c.setSourceAnchor(sourceAnchor);
```

http://atelier.inf.unisi.ch/~malnatij/xray.php

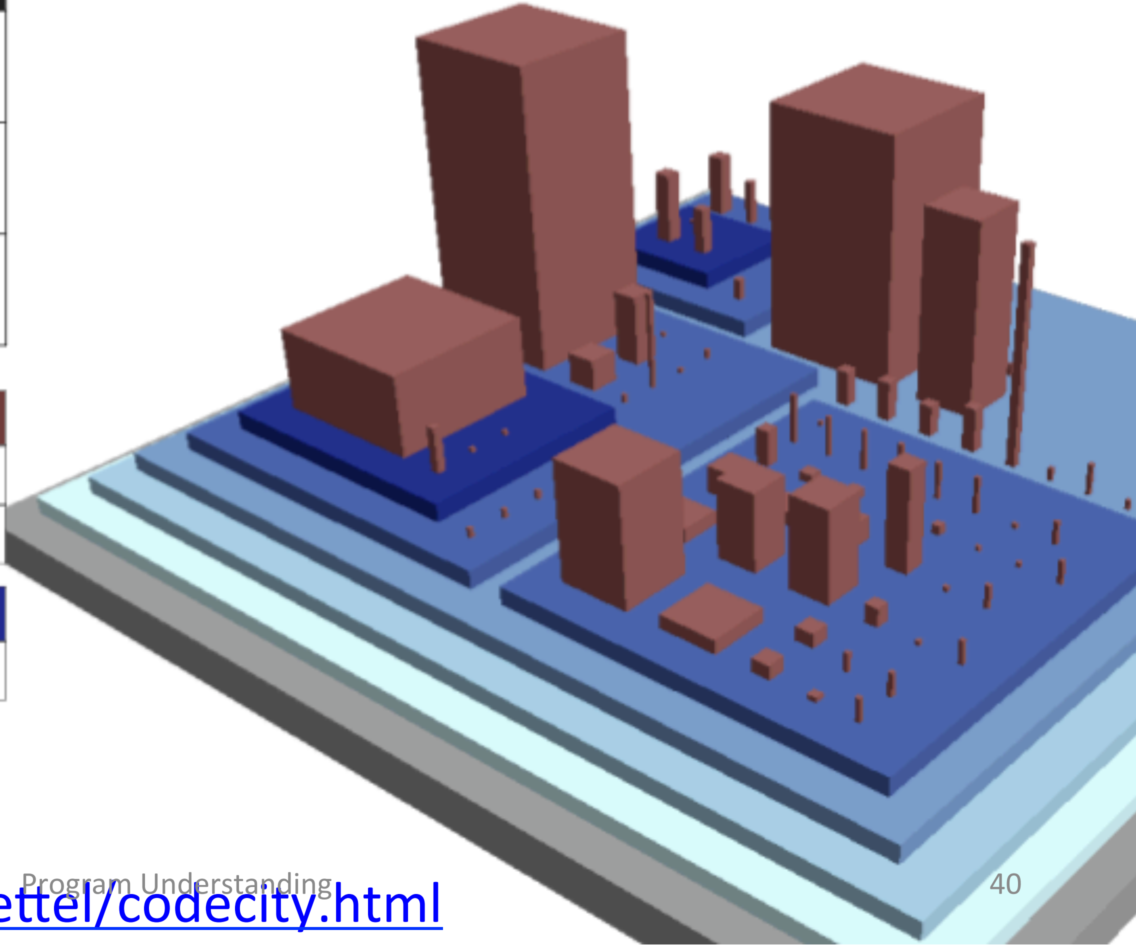
CodeCity



domain mapping	
classes	buildings
packages	districts
system	city

class metric	building property
number of methods (NOM)	height
number of attributes (NOA)	width, length

package metric	district property
nesting level	color



http://www.inf.unisi.ch/phd/wettel/codecity.html

Program understanding

- Manual inspection
  - Static: read, read, and read
  - Dynamic approach
- Class diagrams
- Dependancies
- Program slicing
- Visualization

Making your code understandable

How to write unmaintainable code

- Keep your job forever! ☺
- http://mindprod.com/unmain.html
- http://lgm.fri.uni-lj.si/tpo/ANTI_PATTERNS/unmaintable_code.ppt

Warning!

- This presentation is based on a joke essay.
- It is NOT a good coding practice
- Any attempt to follow these guideline in the project will result in a massive retaliation!

- Be warned!

General principles

- Maintenance programming is based on the ability to locate a specific point within a giant program, and repair it.
- Therefore, you want the maintenance programmer not to be able to ignore anything.

“If it was hard to write, it should be hard to understand”

Naming

- A name gives meaning, so either don't name things or deliberately misname them.
- Single-letter naming
- Creative misspelling:
  - setColor() / setColour()…
- Be abstract! do_the_thing(), run_it()…
- Acronyms and TLA's: "Real men never define acronyms; they understand them genetically"
- Thesaurus: show/display/present/print…

More destructive naming

- Ignore conventions
  - Or better, change them subtly
- Reuse names to override variables
- Cd wrttn wtht vwls s mch trsr
- Misleading names – always have side effects!
  - isValid(x) – return a boolen and commit x to the DB
- Use other languages and film references
- Define l=2 (lowercase L)

Camouflage

- Long, similar names:
  - parseInt() / pasreInt()
- The best of C++/C#
  - Overload operators!
  - Overload new()
  - Macros
  - More macros
  - Even more macros
- In Java you don't have macros, so use AOP…

Documentation

- Lie in the comments!
  - … Or just don't change them with the code
- Document obvious instructions, not intentions
- How to use design documents:
  - Name the functions after the requirement number
    - E.g. do_1_2_4_A()
  - Requirements are auto-numbered, the code is not…
- Use large comments to "drown" the code

What not to document

- Don’t document problems (even potential ones)
- Never comment on the usage of a variable
- Avoid recording the units of measurement used
  - And crash a billion dollar satellite into Mars!

- “Incorrect documentation is worse then no documentation at all”
- Bertrand Mayer

Design

- Use as much casting as you can
  - Use dummy interfaces and casting to hide the type
- Never validate or assert
- Avoid any encapsulation (it's inefficient!)
- Clone a lot of code (it's more efficient!)
- Rely on environment variables a lot
  - Global and static variables are also good
- Never use table-logic or configuration files

More on design

- Make everything public
- Make a lot of final classes and variable
  - Can't be changed!
- Reverse parameters order:
  - (width, height) $\neq$ (height, width)
- Use a lot of unneeded delegation and subclasses
- Never delete dead code
  - Someone might need it someday

Code obfuscation

- Use number literals (“magic numbers”) always
- Nest switches and ifs as much as you can
- The best of C:
  - arr[i] = *(arr + i) =
    *(i + arr) = i[arr]
  - c = a+++++b
- Avoid {} and tabs, use ‘;’ a lot
- Don’t use exceptions (good code never fails)
  - Checking return values is fun!

Summary

- It doesn't take a great skill to write bad code
  - "Real programmers can write assembly code in any language" --- Larry Wall
- Keep in mind that code spends ~70% of its lifetime in debugging and maintenance
  - "Give someone a program, frustrate him for a day; teach him to program and frustrate him for life"

Summary

- It doesn't take a great skill to write bad code
  - "Real programmers can write assembly code in any language" --- Larry Wall
- Keep in mind that code spends ~70% of its lifetime in debugging and maintenance
  - "Give someone a program, frustrate him for a day; teach him to program and frustrate him for life"
  - Try to avoid this – you are your own code's main maintainer!

Critiques of Coding

```
class WorkingCode {
      public int myState;
       void doWork();
      int fetchRecord();
}
```

Should not have public fields

Should not have generic method names

Critiques of coding

```
class WorkingClass {
    private int initialState = 3;
    public boolean isComplete() {
        if(initialState == 5)
         return true;
        else
         initialState = 5;
        return false;
    }
}
```

Should not use "magic numbers"

Should not have side effects in "yes/no" methods

- "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies."

  C. A. R. Hoare

Why is program understanding so hard?

- Size
- Significant features require multiple classes
- Static versus run-time views
  - polymorphism/pointers
- Concepts/assumptions not obvious or inconsistent
- Documentation absent/wrong/outdated

Comments in source code

- Comments often help to understand code

```
/** a() sorts an int array  */
public static void a(int[] a) {
  int a, a;
```

- Many different kinds.
  - line (`//`), block (`/* */`), and Javadoc (`/** */`)
- Comments not always perfect, often outdated
- http://www.google.com/codesearch#search&q=backdoor++lang:C

Decomposition mechanisms

- Modular decomposition
  - divide large software systems into manageable parts called module
  - changes should be localized to a module
- Structured programming
  - high-level programming languages reduce size and complexity

Documentation

- Be careful, documentation is often outdated and of various quality

  "Unfortunately, much computer system documentation is difficult to understand, badly written, out-of-date or incomplete." – Ian Sommerville

- Often code is not documented on purpose

  "By their very nature, API elements are documented and have specification, in contrast to non-API elements which are <u>internal implementation details usually without published documentation</u> or specifications"

  – Jim des Rivières in "How to Use the Eclipse API"

Presentation of programs

- Remember for (a=a; a>=a && a[a-a]>a; a-=a)?

```
public static void a(int[] a) { int a, a; int a = 1; do { a = a * 3 + 1; }
while (a <= a.length); do { a /= 3; for (a = a; a < a.length; a++) { int a
= a[a]; for (a=a; a>=a&&a[a-a]>a; a-=a) a[a] = a[a - a]; if (a != a) a[a]
= a; } } while (a != 1); }
```

- Indentation, spacing, and syntax highlighting increase the readability of source code

"Effective program presentation makes good programs more understandable and bad programs more obvious"
— Baecker & Marcus, 1990

Mistake #1

- Dive in without a plan
- Hope to stumble across something significant
- Keep going until that significant something is found

Mistake #2

- “I don’t have enough time to get a rough understanding”
- “I’ll just come back to it if I need to”
- A rough understanding will save a lot of headaches, even in the short-term.

A good approach when starting from scratch

- Get a rough understanding of major parts, via browsing/scrolling/reading
- Formulate hypothesis
- Evaluate hypothesis
- If false, refine understanding of system structure and details
- Iterate

Issues even with successful approaches

- Aha, here's the answer but … how did I get here?
- I'm sure this key word matters, but I keep getting false hits
- How do I get the rough understanding or formulate a reasonable hypothesis?
- Re-assess constantly

TOOL SUPPORT CAN HELP

Effective keyword searches

- Lexical
  - no knowledge of programming language (“grep”)
- Syntactic
  - e.g., a class declaration is different from local variable declaration
- Semantic
  - import statement alters names
- Conceptual
  - very very hard ⇒ job for humans

Final advice

- Everyone gets lost in big systems
- Standard navigation strategies
- Applying strategies systematically better than crossing your fingers
- Don't count on the documentation or other easy answers
- In "good" software, structure and names matter: use them to your advantage

Good names

- No single letter names: `int t` → `int time;`
- No generic names: `void dothings()` → `void handleEvent();`
- No acronyms in names: `int SAT` → saturday? satisfiable? SAT test?
- No variations: display/print/output/show → show
- Observe convention:
  - Field names → `width_, height_`
  - Method names → `smallLarge, getLocation, performAction`

Good design

- Don’t make your private parts public, fields are never public
- Don’t use “magic numbers”: `int state = 5` → `#define START 5; int state = START;`
- Use exceptions in favor of return codes

```
int process() {
    if(…) return SUCCESS;
    if(...) return FAILURE;
}
```

→

```
int process() {
    if(...) throw new FAILURE();
}
```