

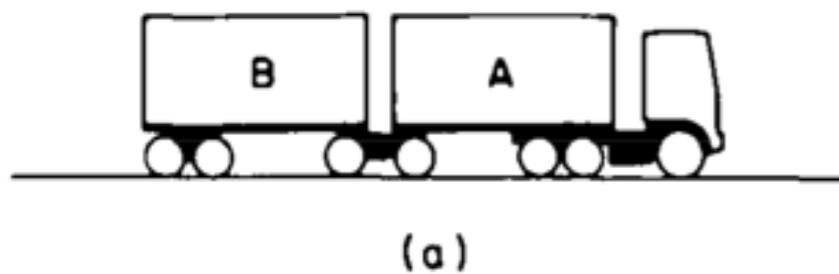
Software Visualization

SWE 795, Spring 2017
Software Engineering Environments

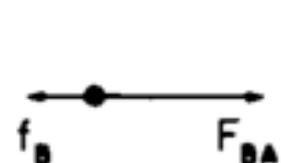
Today

- Part 1 (Lecture)(~60 mins)
 - Software visualization
- Part 2 (In class activity)(~30 mins)
 - Sketch a software visualization
- Break!
- Part 2 (Discussion)(45 mins)
 - Discussion of readings

Why a diagram is (sometimes) worth ten thousand words



(a)



$$F_{BA} - f_B = M_B a$$



$$F - f_A - F_{AB} = M_A a$$

- Diagrams can group together all information that is used together, thus avoiding large amounts of search for the elements needed to make a problem-solving inference.
- Diagrams typically use location to group information about a single element, avoiding the need to match symbolic labels.
- Diagrams automatically support a large number of perceptual inferences, which are extremely easy for humans
- Larkin & Simon, 1987, Cognitive Science 11, pp 65-99.

How information visualization amplifies cognition.

Increased Resources

High-bandwidth hierarchical interaction

The human moving gaze system partitions limited channel capacity so that it combines high spatial resolution and wide aperture in sensing visual environments (Resnikoff, 1987).

Parallel perceptual processing

Some attributes of visualizations can be processed in parallel compared to text, which is serial.

Offload work from cognitive to perceptual system

Some cognitive inferences done symbolically can be recoded into inferences done with simple perceptual operations (Larkin and Simon, 1987).

Expanded working memory

Visualizations can expand the working memory available for solving a problem (Norman, 1993).

Expanded storage of information

Visualizations can be used to store massive amounts of information in a quickly accessible form (e.g., maps).

Reduced Search

Locality of processing

Visualizations group information used together, reducing search (Larkin and Simon, 1987).

High data density

Visualizations can often represent a large amount of data in a small space (Tufte, 1983).

Spatially indexed addressing

By grouping data about an object, visualizations can avoid symbolic labels (Larkin and Simon, 1987).

Enhanced Recognition of Patterns

Recognition instead of recall

Recognizing information generated by a visualization is easier than recalling that information by the user.

Abstraction and aggregation

Visualizations simplify and organize information, supplying higher centers with aggregated forms of information through abstraction and selective omission (Card, Robertson, and Mackinlay, 1991; Resnikoff, 1987).

Visual schemata for organization

Visually organizing data by structural relationships (e.g., by time) enhances patterns.

Value, relationship, trend

Visualizations can be constructed to enhance patterns at all three levels (Bertin, 1977/1981).

Perceptual Inference

Visual representations make some problems obvious

Visualizations can support a large number of perceptual inferences that are extremely easy for humans (Larkin and Simon, 1987).

Graphical computations

Visualizations can enable complex specialized graphical computations (Hutchins, 1996).

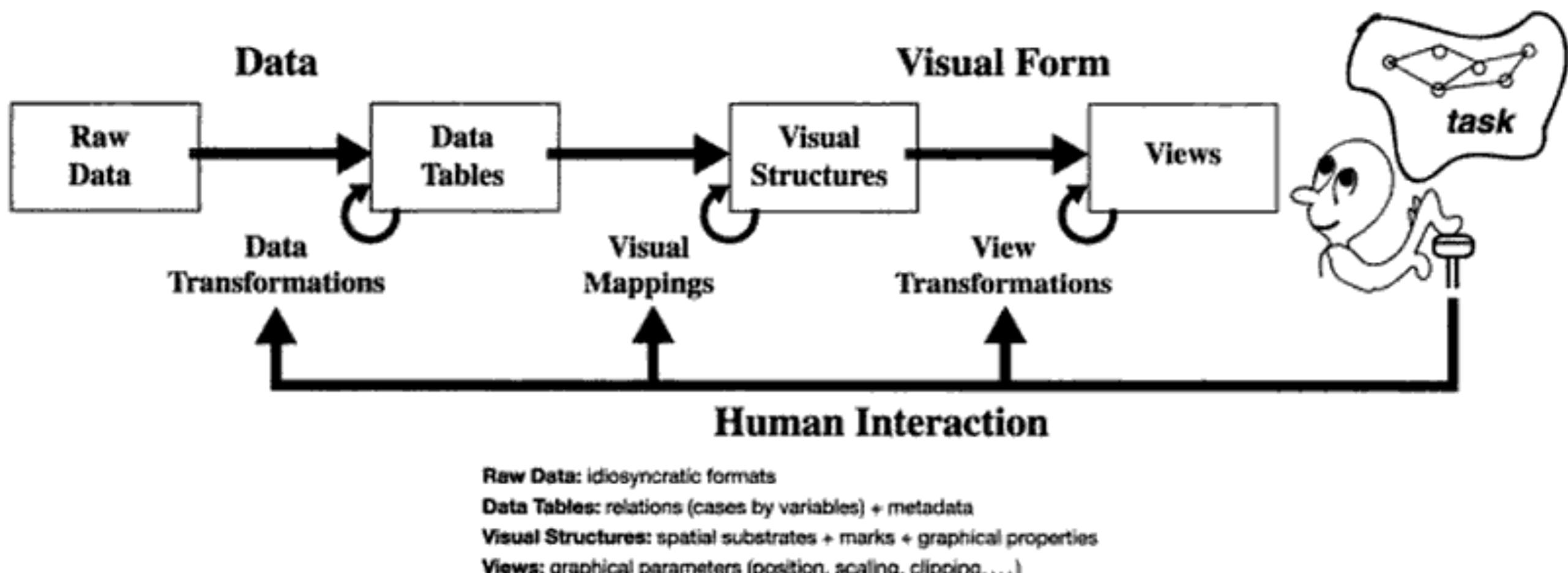
Perceptual Monitoring

Visualizations can allow for the monitoring of a large number of potential events if the display is organized so that these stand out by appearance or motion.

Manipulable Medium

Unlike static diagrams, visualizations can allow exploration of a space of parameter values and can amplify user operations.

Designing an information visualization



S.K.Card, J.D.Mackinlay, B.Shneiderman, "Information Visualization", Readings in Information Visualization: Using Vision to Think, Morgan Kaufman, Chapter 1.

Marks' graphical properties

- Quantitative (Q), Ordinal (O), Nominal (N)
- Filled circle - good; open circle - bad

Spatial		Object			
Extent	(Position)	— — —	Gray Scale	■ ■ ■ □	
	Size	● ● ● .	Color	■ ■ ■ ■	
Dif- feren- tial	Orientation	— / \	Texture	■ ■ ■ ■	
			Shape	■ ★ ● ◆	

Effectiveness of graphical properties

- Quantitative (Q), Ordinal (O), Nominal (N)
- Filled circle - good; open circle - bad

	Spatial	Q	O	N	Object	Q	O	N
Extent	(Position)	●	●	●	Grayscale	○	●	○
	Size	●	●	●				
Differential	Orientation	○	○	●	Color	○	○	●
					Texture	○	○	●
					Shape	○	○	●

Tufte's principles of graphical excellence

- show the **data**
- induce the viewer to think about the substance rather than the methodology
- avoid distorting what the data have to say
- present **many** numbers in a small space
- make large data sets **coherent**
- encourage the eye to **compare** different pieces of data
- reveal data at several levels of detail, from overview to fine structure
- serve reasonable clear **purpose**: description, exploration, tabulation, decoration

Interactive visualizations

- Users often use iterative process of making **sense** of the data
 - Answers lead to new questions
- Interactivity helps user constantly change display of information to answer new questions
- Should offer visualization that offers best view of data moment to **moment** as desired view **changes**

How software visualizations may help

- Offer information that helps developers to answer questions
- Facilitate easier navigation between artifacts containing relevant information

Key questions for software visualization design

- Do you *really* need a visualization?
 - If you know the developer's question, can you answer it more simply *without* a visualization?
- **Anti**-pattern: show all the information, let user find patterns
 - In other domains (e.g., data analytics), visualization is a tool for data exploration and understanding dataset.
 - **Not true for SE**: developers want to complete tasks, finding patterns often not relevant
- How much context do you need?
 - More context —> more information to sort through
 - Less context —> more direct

Some popular forms of software visualizations

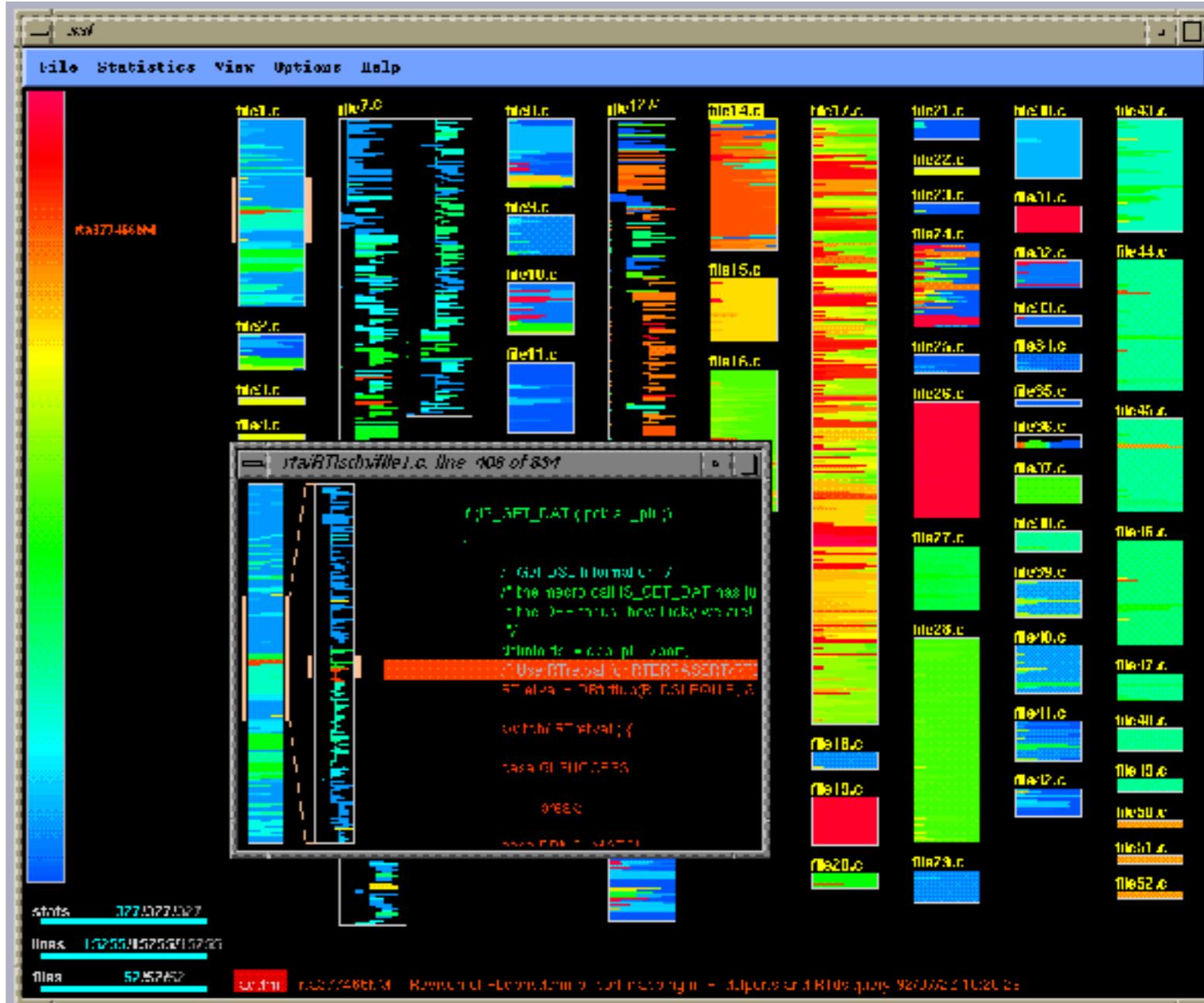
- Code
 - Iconographic representation of code text
- Algorithm & object structure visualizations
 - Depictions of data value changes over time
 - Runtime snapshots of object reference structure
- Module structure
 - Static views of module properties & dependencies (e.g., calls, references)
- Function calls
 - Dynamic and static depictions of function calls

Code visualizations

- Offer overview of source code
- Identify relevant sources lines matching some property
 - e.g., changed in a commit, passing a test, with a compiler warning
- Represent lines of iconographically
 - e.g., colored lines

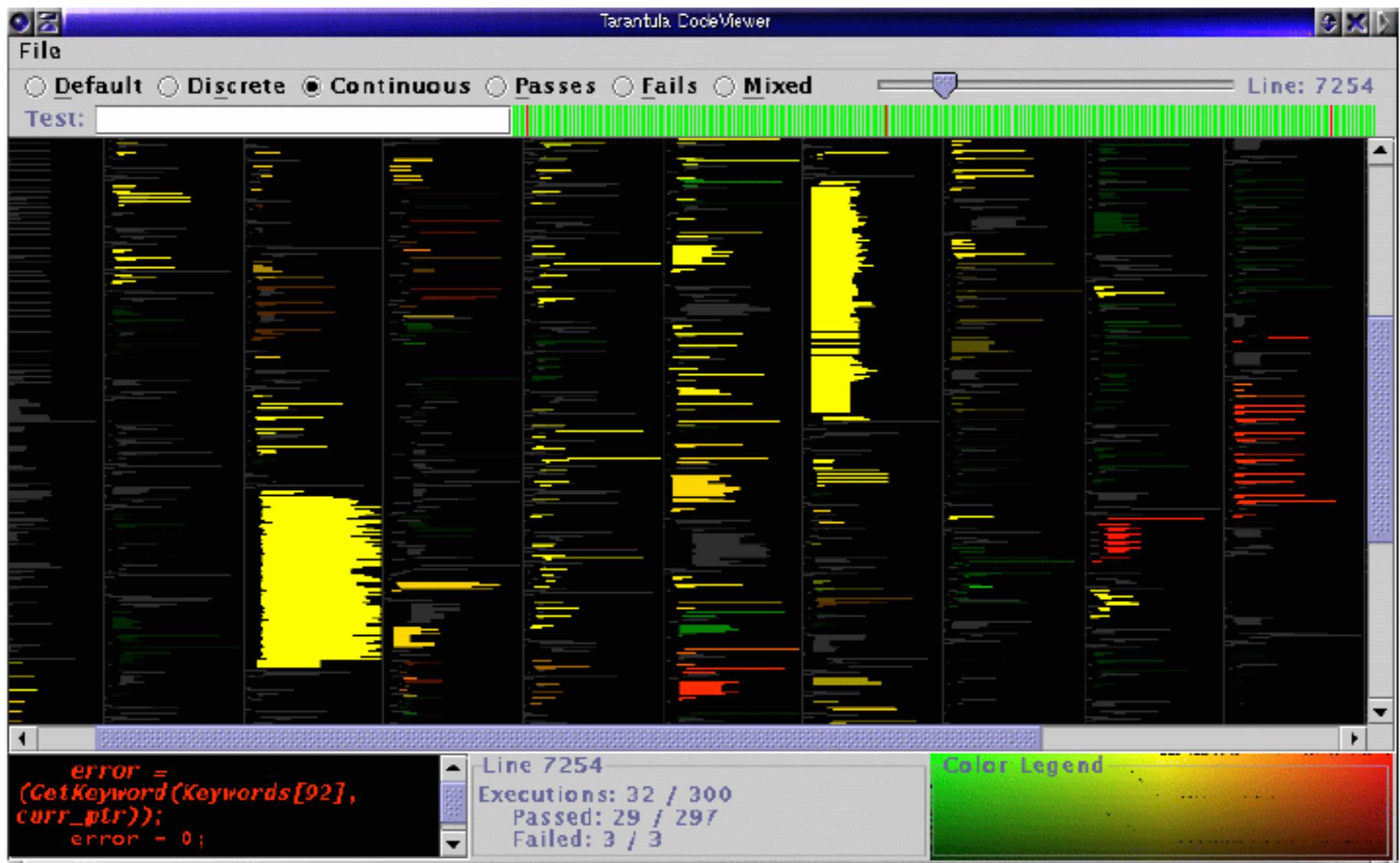
SeeSoft

AT&T Bell Labs [Eick, 1992]
Visualization for performance
“Hot spots” in red
Large volumes of code
Image is of 15,255 LOC
Up to 50,000 LOC
Can indent like original
source files
Also, recently changed,
Version control systems
Static, dynamic analyses
Interactive investigation



Stephen G. Eick, Joseph L. Steffen, and Eric E. Sumner, Jr.. 1992. Seesoft-A Tool for Visualizing Line Oriented Software Statistics. IEEE Trans. Softw. Eng. 18, 11 (November 1992), 957-968.

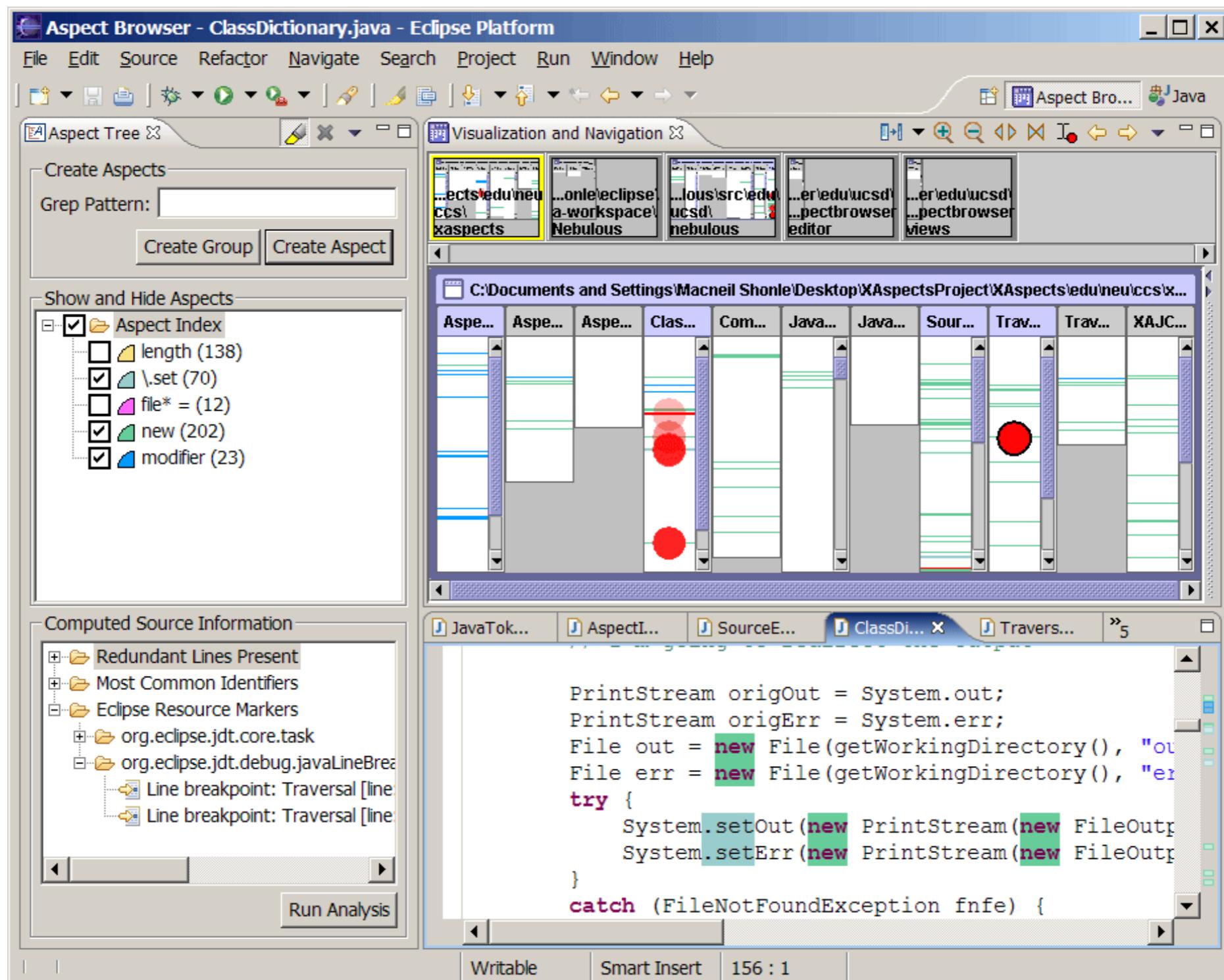
Tarantula



Color – code coverage
Red – failed test case
Green – past test case
Yellow – hue is % of test cases passing

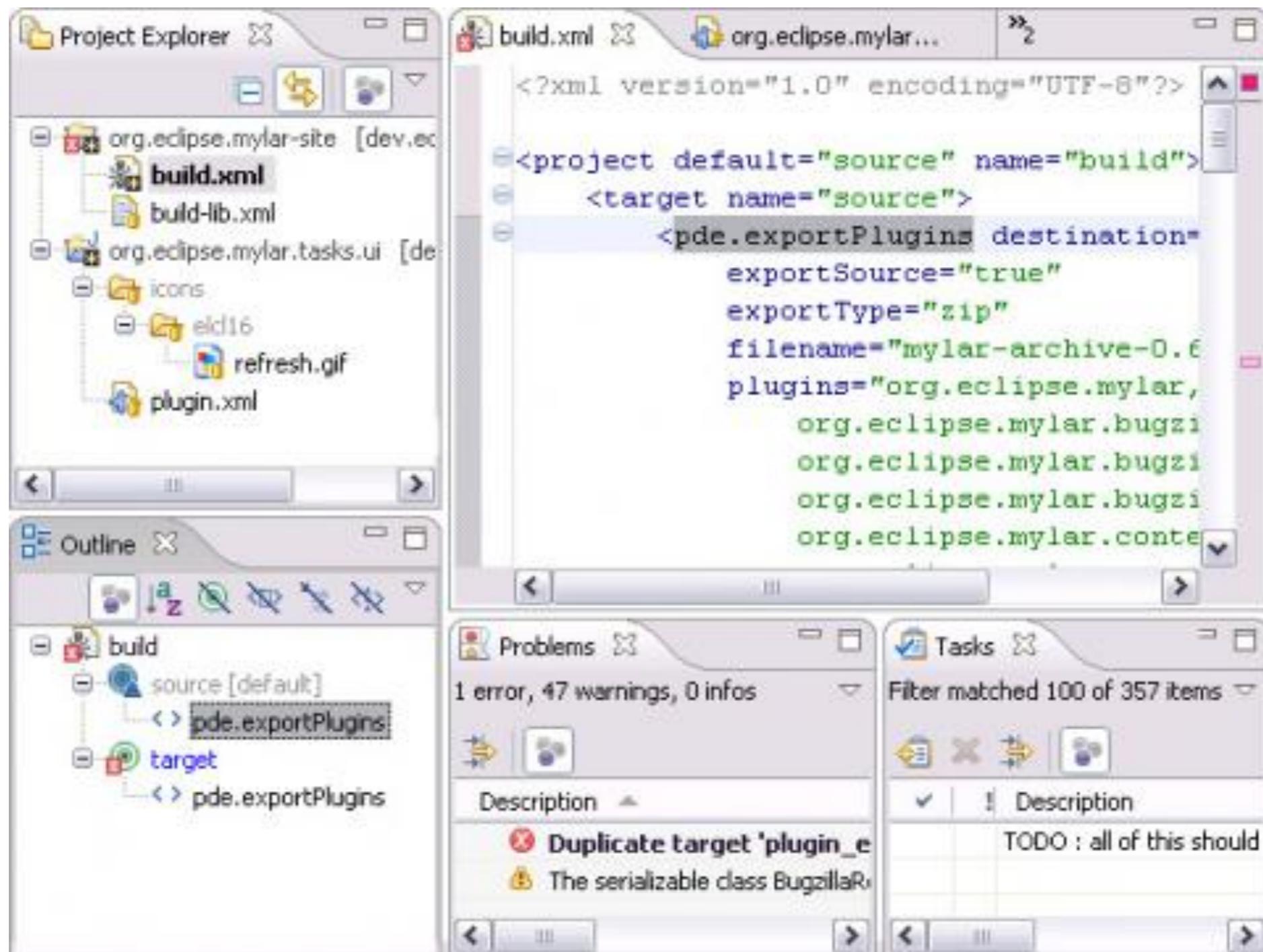
James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. International Conference on Software Engineering (ICSE '02), 467-477.

AspectBrowser

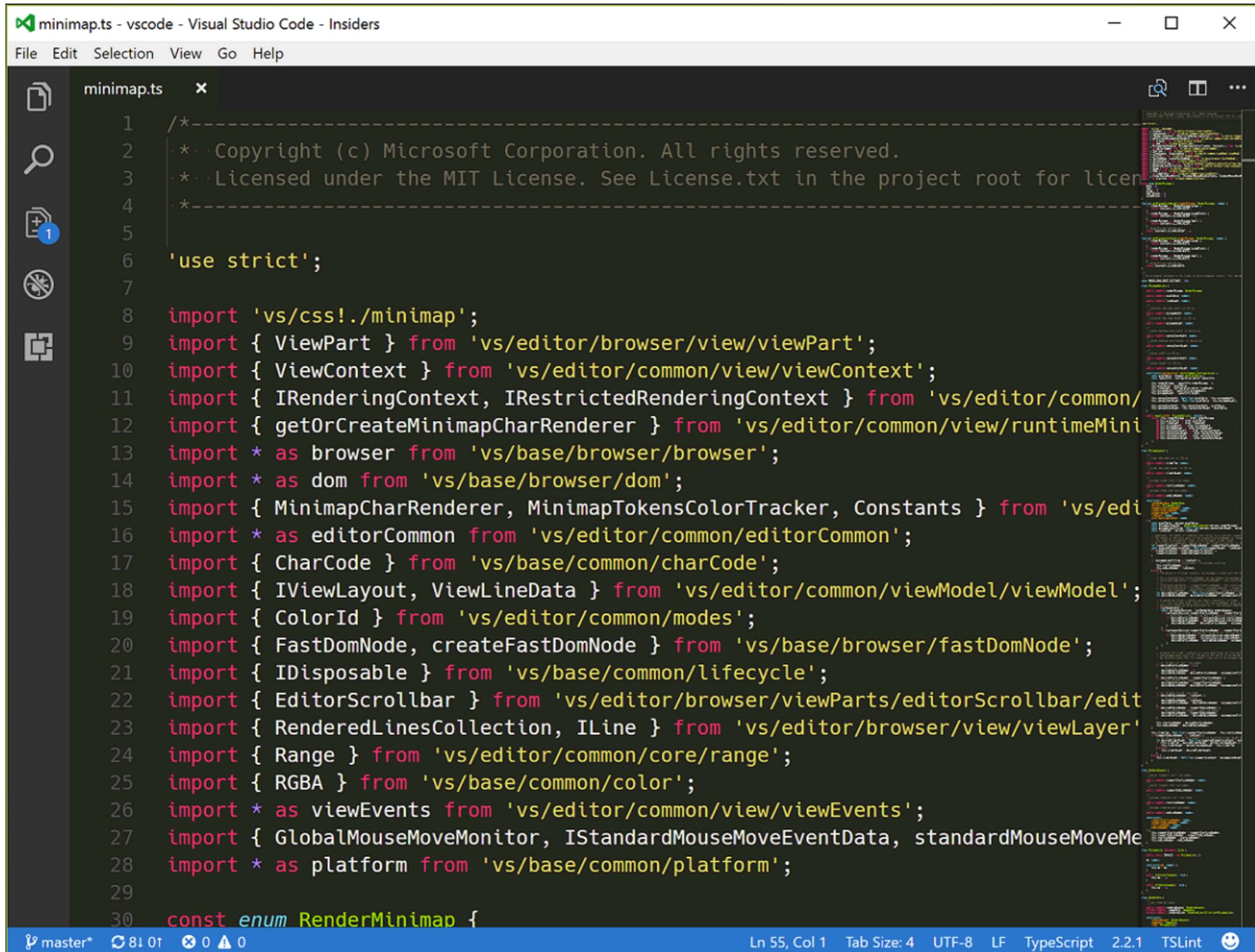


Macneil Shonle, Jonathan Neddenriep, and William Griswold. 2004. AspectBrowser for Eclipse: a case study in plug-in retargeting. In Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange (eclipse '04). ACM, New York, NY, USA, 78-82.

Industry Use: Eclipse Markers



Industry use: Visual Studio Code Minimap



A screenshot of the Visual Studio Code interface, specifically showing the Minimap feature. The main editor area displays a file named 'minimap.ts' with the following code:

```
1  /*
2   * Copyright (c) Microsoft Corporation. All rights reserved.
3   * Licensed under the MIT License. See License.txt in the project root for license
4   * terms.
5
6  'use strict';
7
8  import 'vs/css!./minimap';
9  import { ViewPart } from 'vs/editor/browser/view/viewPart';
10 import { ViewContext } from 'vs/editor/common/view/viewContext';
11 import { IRenderingContext, IRestrictedRenderingContext } from 'vs/editor/common/';
12 import { getOrCreateMinimapCharRenderer } from 'vs/editor/common/view/runtimeMini
13 import * as browser from 'vs/base/browser/browser';
14 import * as dom from 'vs/base/browser/dom';
15 import { MinimapCharRenderer, MinimapTokensColorTracker, Constants } from 'vs/edi
16 import * as editorCommon from 'vs/editor/common/editorCommon';
17 import { CharCode } from 'vs/base/common/charCode';
18 import { IViewLayout, ViewLineData } from 'vs/editor/common/viewModel/viewModel';
19 import { ColorId } from 'vs/editor/common/modes';
20 import { FastDomNode, createFastDomNode } from 'vs/base/browser/fastDomNode';
21 import { IDisposable } from 'vs/base/common/lifecycle';
22 import { EditorScrollbar } from 'vs/editor/browser/viewParts/editorScrollbar/edit
23 import { RenderedLinesCollection, ILine } from 'vs/editor/browser/view/viewLayer'
24 import { Range } from 'vs/editor/common/core/range';
25 import { RGBA } from 'vs/base/common/color';
26 import * as viewEvents from 'vs/editor/common/view/viewEvents';
27 import { GlobalMouseMoveMonitor, IStandardMouseMoveEventData, standardMouseMoveMe
28 import * as platform from 'vs/base/common/platform';
29
30 const enum RenderMinimap {
```

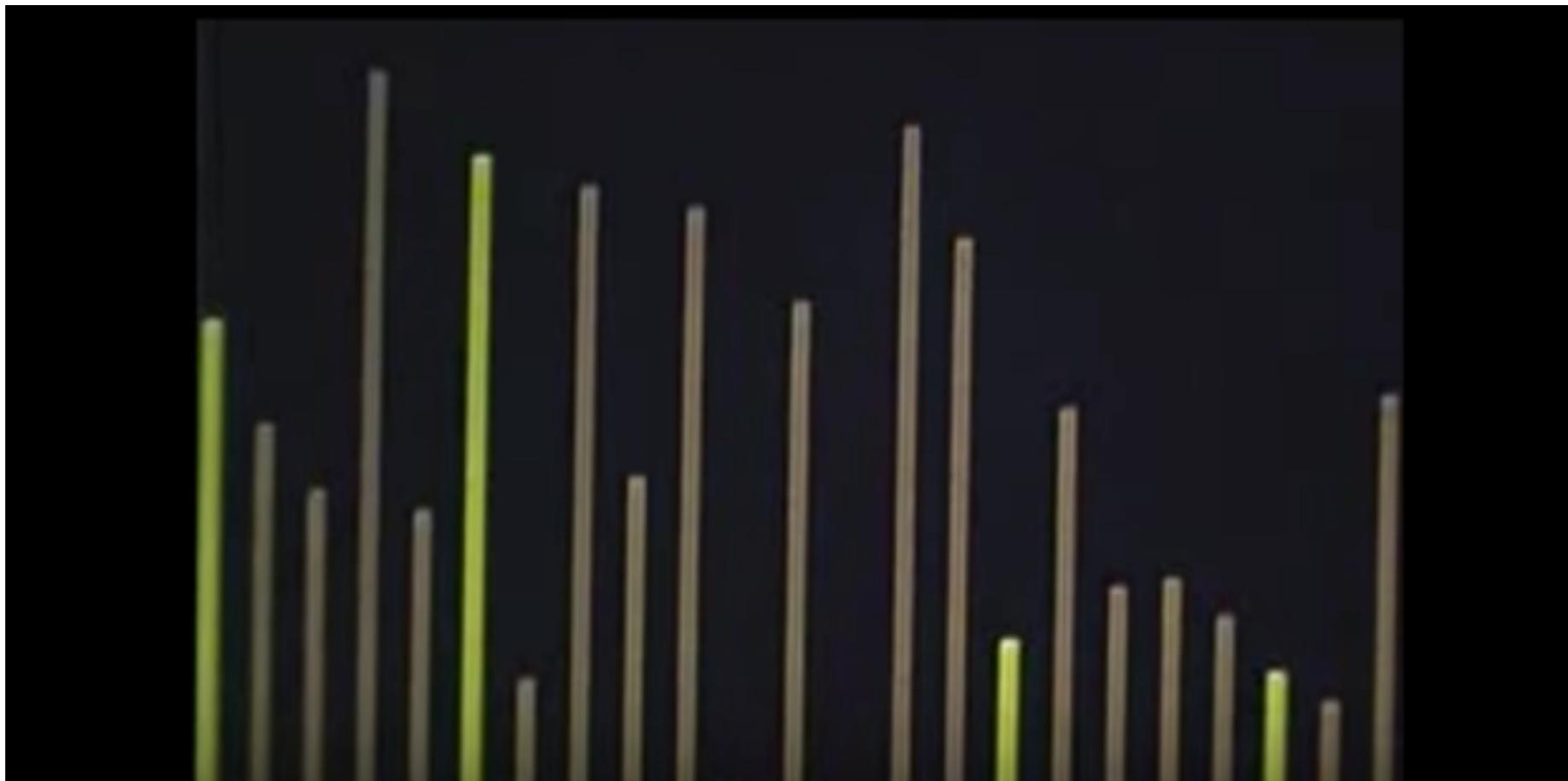
The Minimap is visible on the right side of the editor, showing a thumbnail representation of the code file. The status bar at the bottom shows the file is on 'master' branch, has 810 lines of code, 0 errors, 0 warnings, and 0 info messages. The status bar also indicates the file is in 'Ln 55, Col 1' with 'Tab Size: 4', 'UTF-8', 'LF', 'TypeScript 2.2.1', and 'TSLint' enabled.

Algorithm & object structure visualizations

- Depict runtime state at a snapshot or over time
 - e.g., elements in a collection, numeric values
- Often focused on teaching basic algorithms (e.g., sorting algorithms, linked list insertion)

(Section adapted from Software Visualization, Lecture by Brad A. Myers, Spring 2011)

Sorting out Sorting



<https://www.youtube.com/watch?v=SJwEwA5gOkM>

Incense

First to automatically
create viz. of data
structures

Produce pictures
“like you
might draw them
on a blackboard”

Goal: help with
debugging

Figure 14.
ARRAY [1..4] OF POINTER with two POINTERS
referring to the same value.

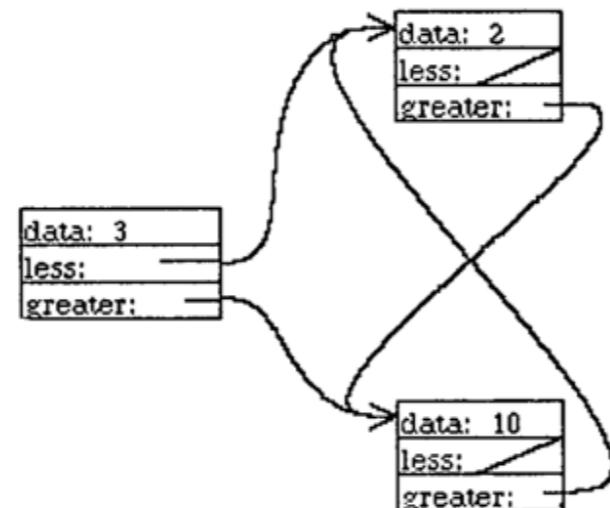


Figure 15.
This erroneous tree structure demonstrates that a pointer
to previously displayed object does not generate a new
copy. The second arrow is drawn to the first occurrence.

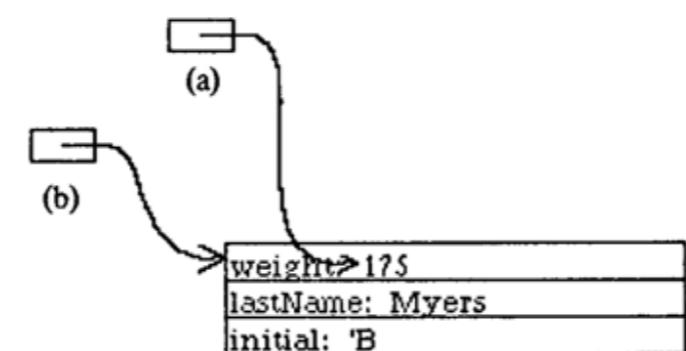


Figure 16.
Pointer to value inside a record (a) does not get confused
with a pointer to the record itself (b).

Figure 17.
Incense display for
RECORD [int: INTEGER, p1: POINTER TO CARDINAL].

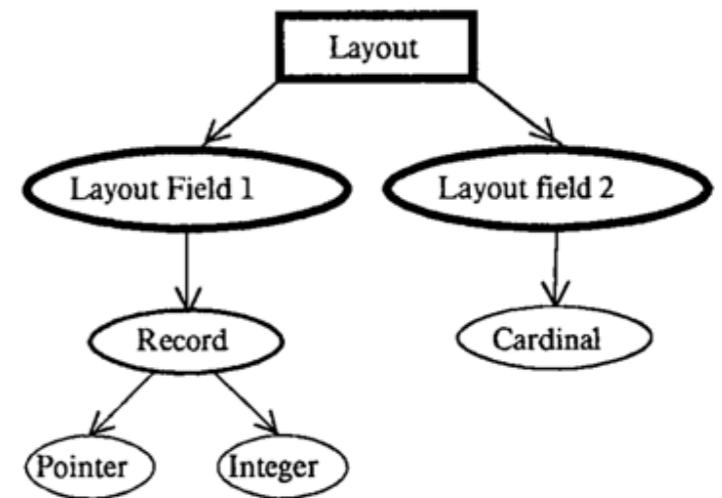


Figure 18.
Artist hierarchy that would be created for:
rec: RECORD [p1: POINTER TO CARDINAL, int: INTEGER];
(This figure was not created by Incense).

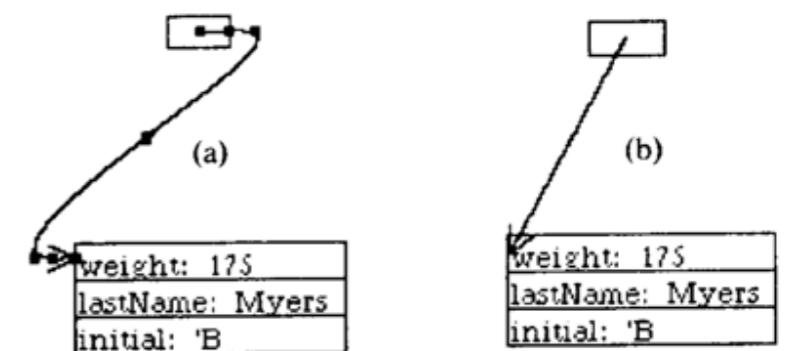


Figure 19.
Demonstration of the advantage of curved lines used in
Incense (a) over straight lines (b). The control points used
to specify the spline are shown as black squares in (a).

Brown University Algorithm Simulator and Animator (BALSA)

Major interactive integrated system

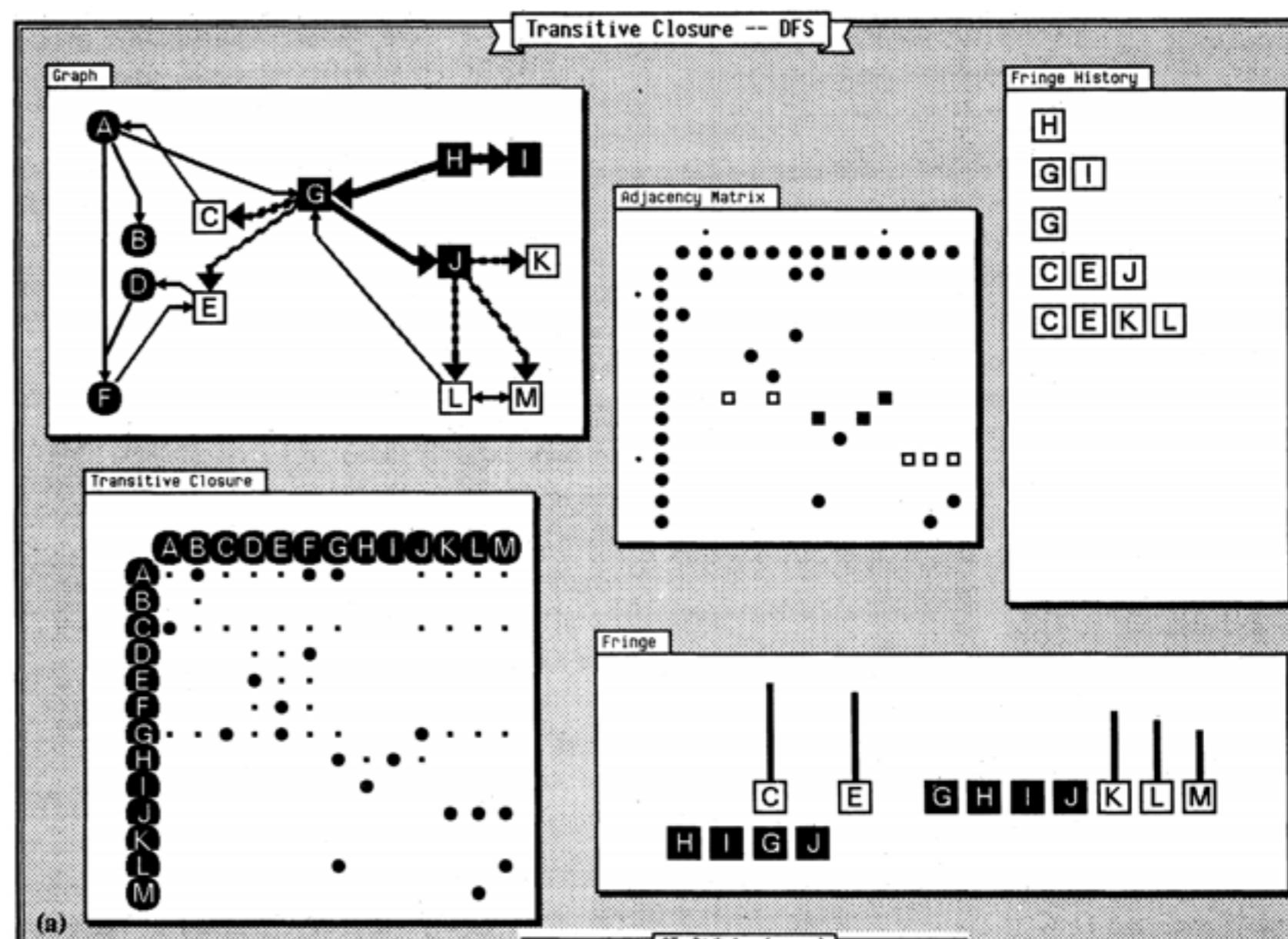
Extensively used for teaching at
Brown Univ.

Lots of algorithms visualized

Architecture for attaching the graphics

Still required significant
programming for each
viz.

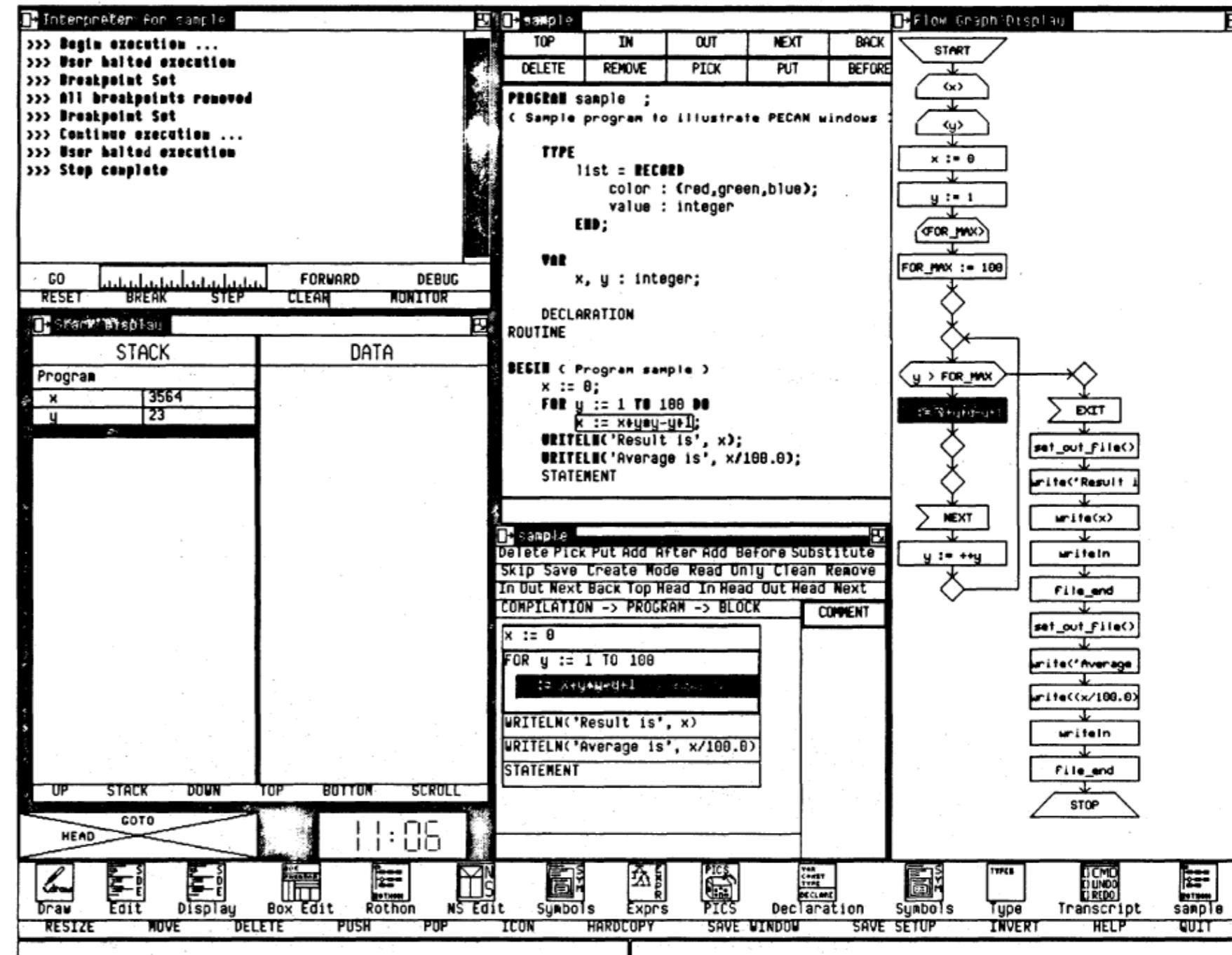
Marc followed up with
Zeus ('91) at DEC SRC



Marc H. Brown and Robert Sedgewick. Techniques for Algorithm Animation. *IEEE Software*, 1985.

PECAN

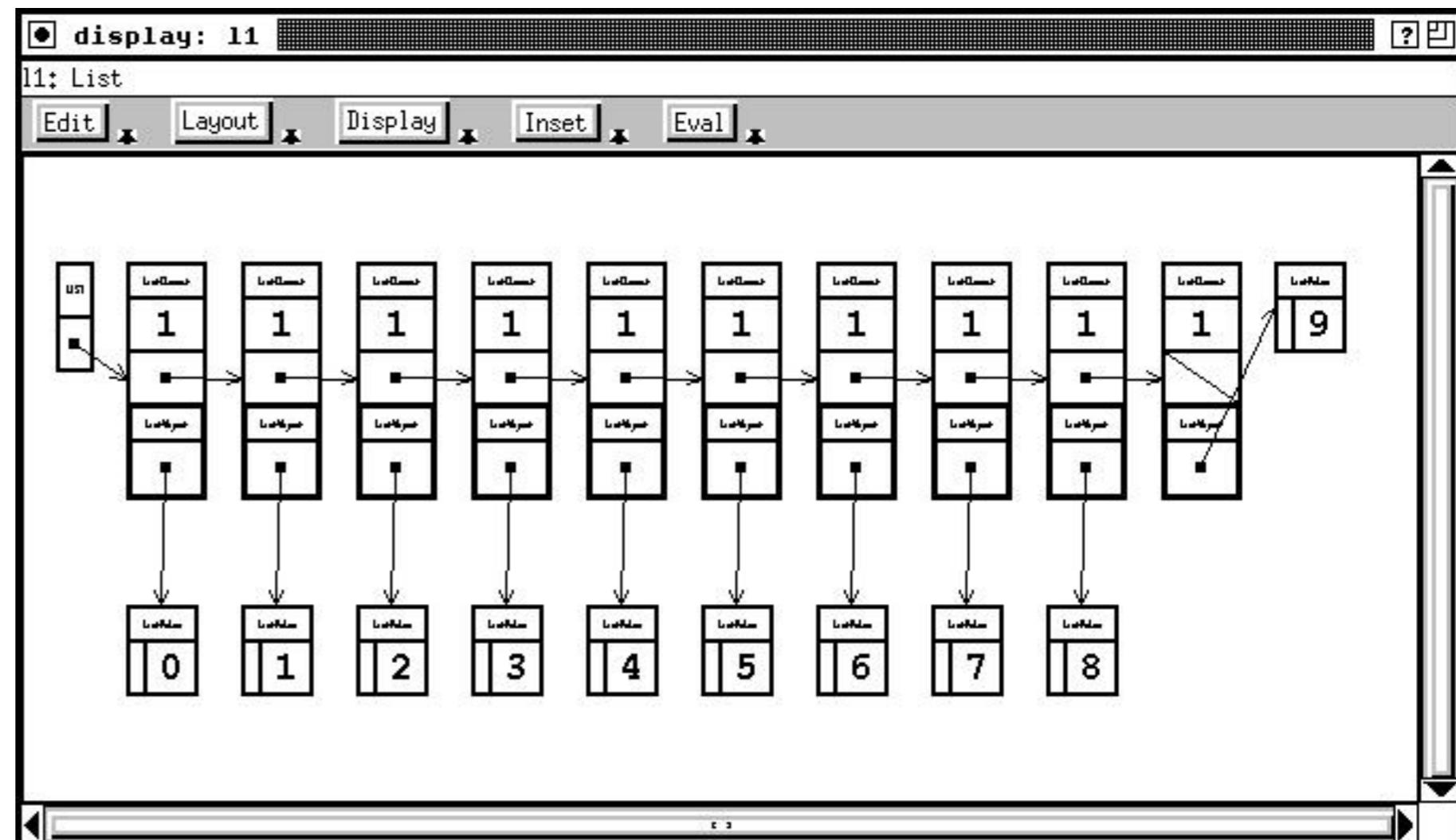
Steven Reiss at Brown's code & data visualization systems
 Take advantage of new Apollo workstation capabilities
PECAN (1985) – automatic graphics about the program
 Multiple views
 Integrates Balsa data visualization
 Syntax directed editing
 Drag and drop
 Flowcharts of code
 Code highlighting while executing
 Data viz. like Incense
 Incremental compilation
 Could handle up to 1000 LOC



Steven P. Reiss. 1984. Graphical program development with PECAN program development systems. In Proceedings of the first ACM SIGSOFT/SIGPLAN software engineering symposium on Practical software development environments (SDE 1), 30-41.

Friendly Integrated Environment for Learning and Development (FIELD)

Field (1990) – IDE,
wrappers for Unix tools
Code and data viz.
Message-based (control)
integration
Basis for most other Unix
IDEs
Widely used
Followed by
DESERT, ...



Steven P. Reiss: Interacting with the FIELD environment. Softw., Pract. Exper. 20(S1): S1 (1990)

Transition-based Animation Generation (TANGO)

John Stasko PhD thesis at Brown Univ. (1990)
Smooth animations between states
Paths & transitions
Make it easier to author algorithm visualizations
Events inserted into the code tied to animations

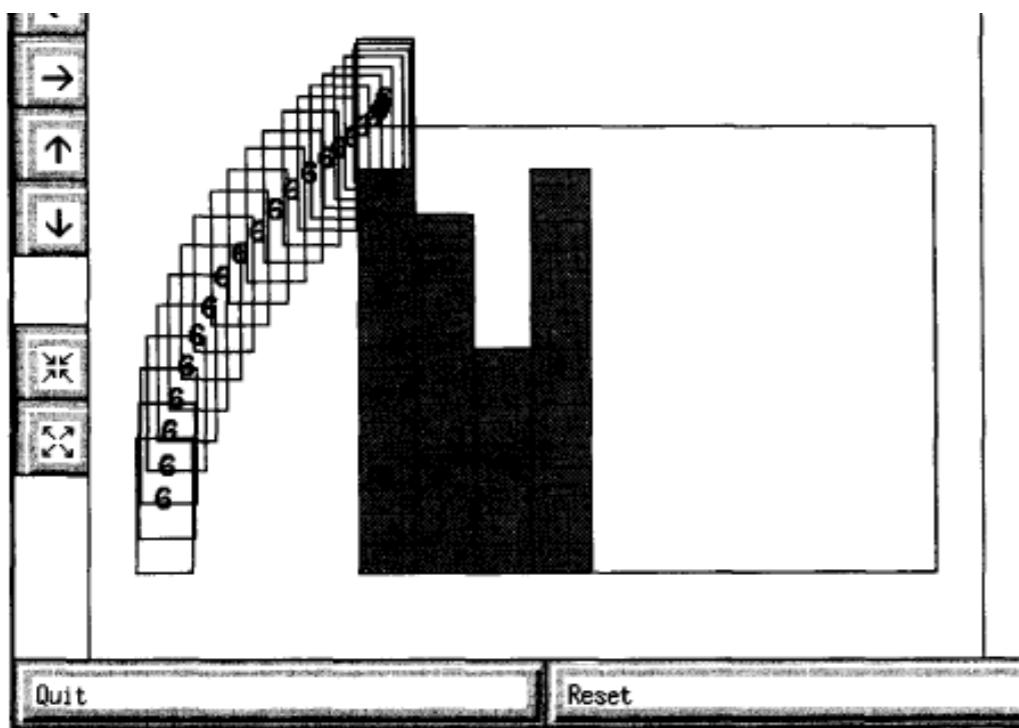


Figure 9. Superimposed sequence of frames from the bin-packing animation.

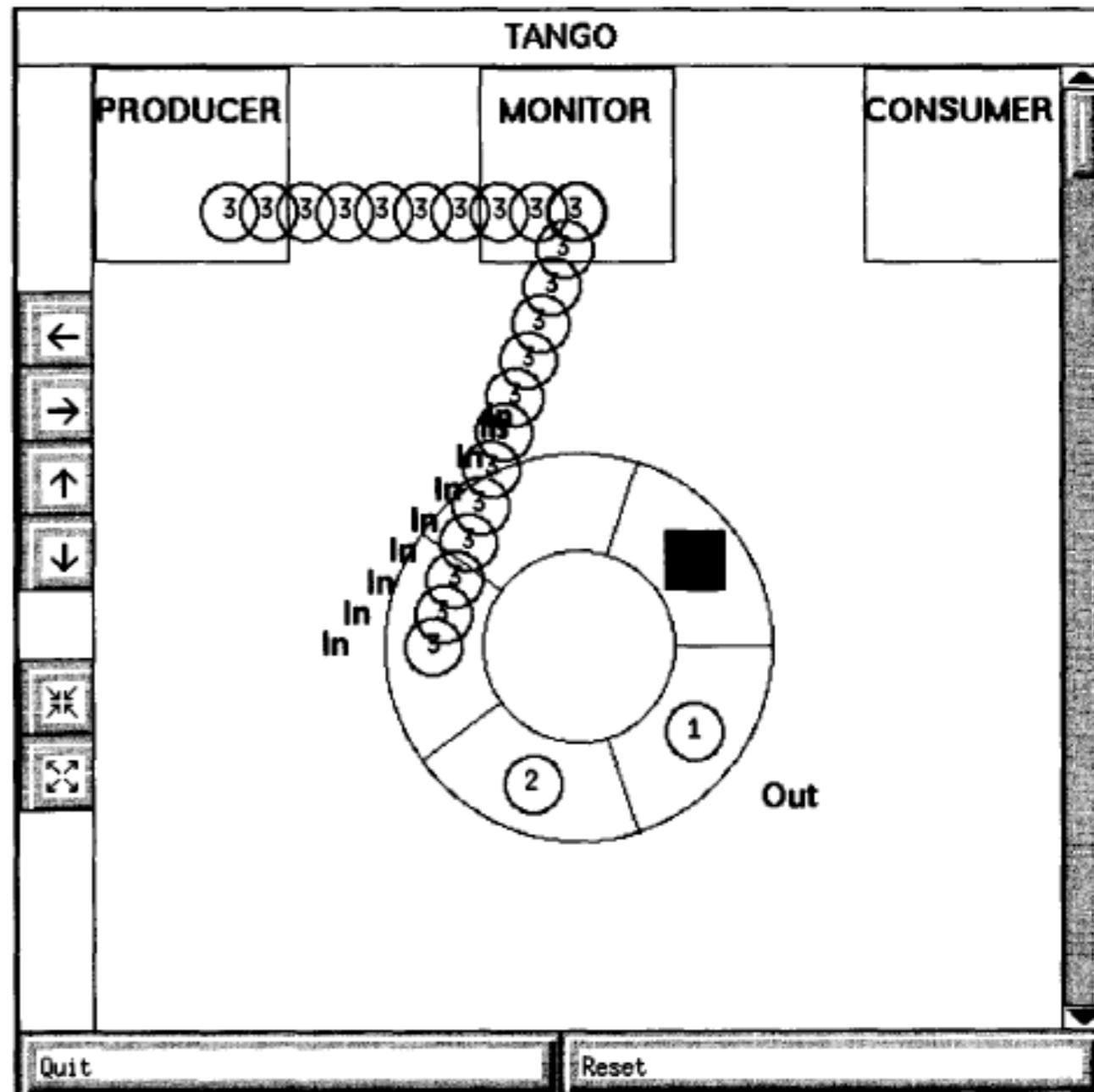
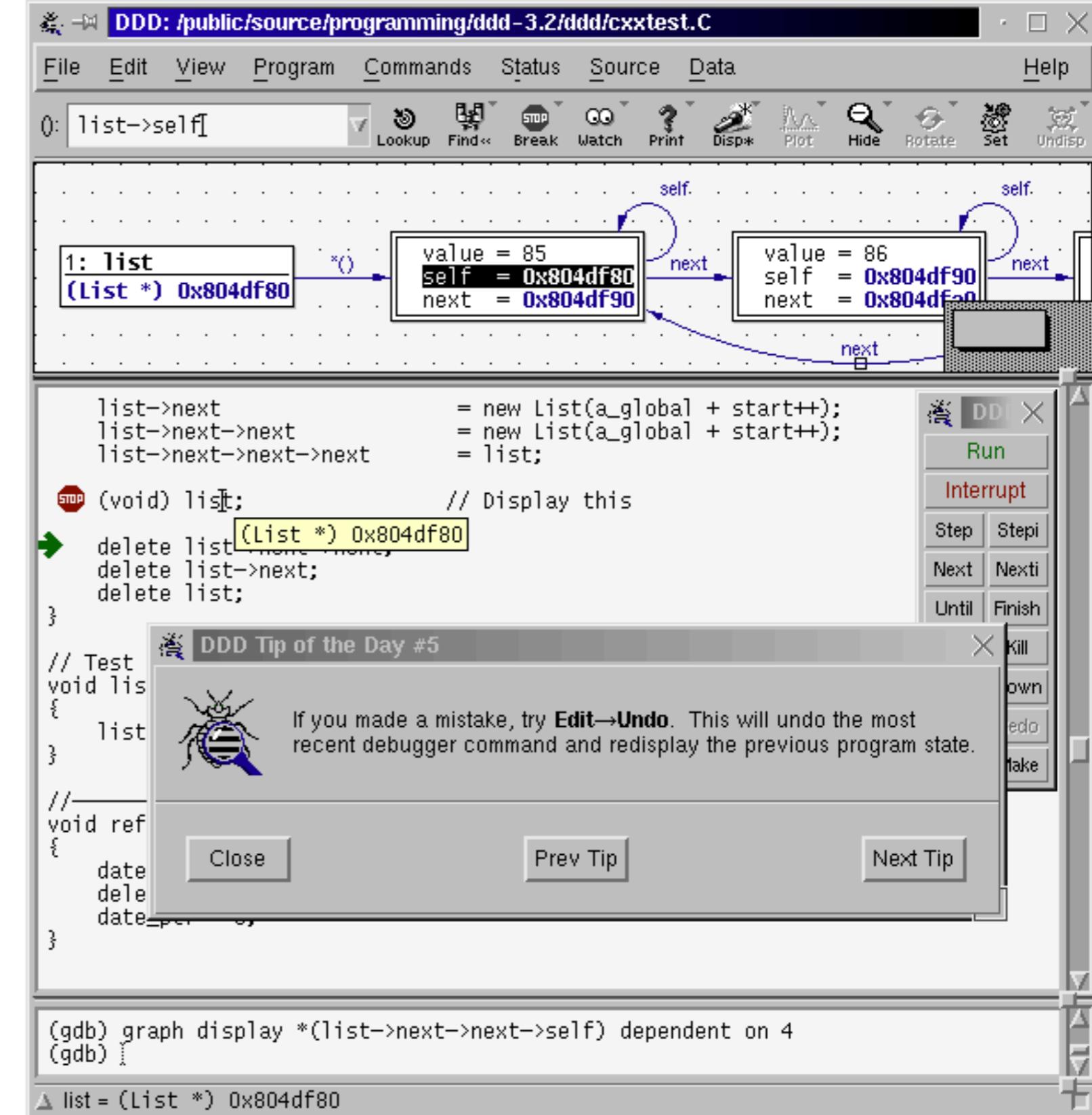


Figure 2. Tango animation of a producer-consumer ring buffer.

J. T. Stasko, "Tango: a framework and system for algorithm animation," in Computer, vol. 23, no. 9, pp. 27-39, Sept. 1990.

Data Display Debugger



<https://www.gnu.org/software/ddd/>

Andreas Zeller and Dorothea Lütkehaus. 1996. DDD—a free graphical front-end for UNIX debuggers. SIGPLAN Not. 31, 1 (January 1996), 22-27.

PythonTutor

<http://pythontutor.com/>

Python 2.7

```
1 def listSum(numbers):
2     if not numbers:
3         return 0
4     else:
5         (f, rest) = numbers
6         return f + listSum(rest)
7
8 myList = (1, (2, (3, None)))
9 total = listSum(myList)
```

[Edit code](#)

line that has just executed
next line to execute

< Back Step 11 of 22 Forward >

Visualized using [Python Tutor](#) by [Philip Guo](#)

Frames Objects

Global frame

listSum

myList

listSum

numbers

f

rest

listSum

numbers

f

rest

function listSum(numbers)

tuple

tuple

tuple

0 1

0 1

0 1 None

1

2

3

None

Over 2.5 million people in over 180 countries have used Python Tutor to visualize over 20 million pieces of code

Philip J. Guo. Online Python Tutor: Embeddable Web-Based Program Visualization for CS Education. In Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE), March 2013.

Module Views

- Depict static structure of modules (e.g., files, folders, packages)
- Often depicts dependencies between modules
- Focus on reverse engineering tasks, refactoring tasks, other architecture related tasks

SHriMP

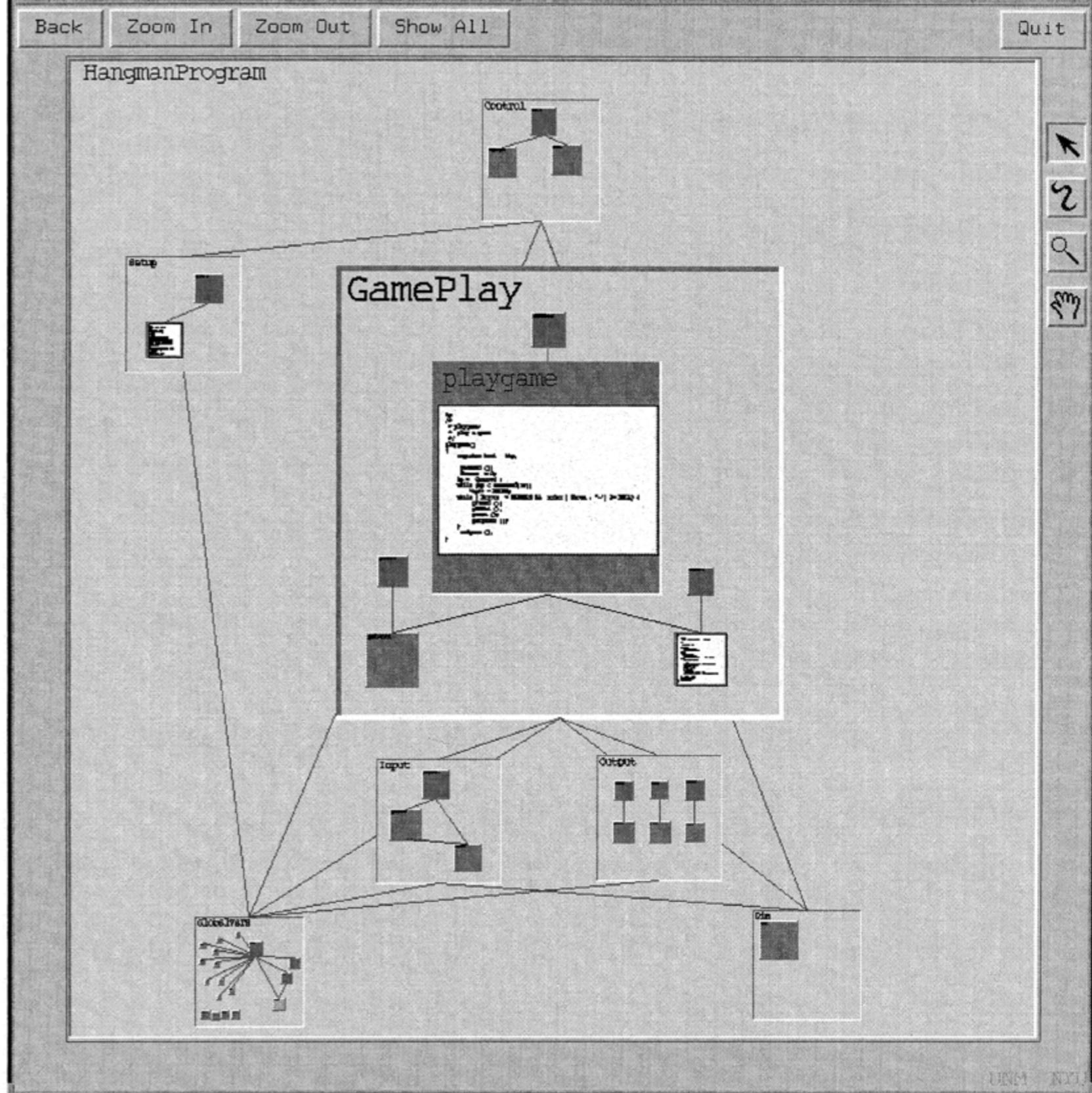
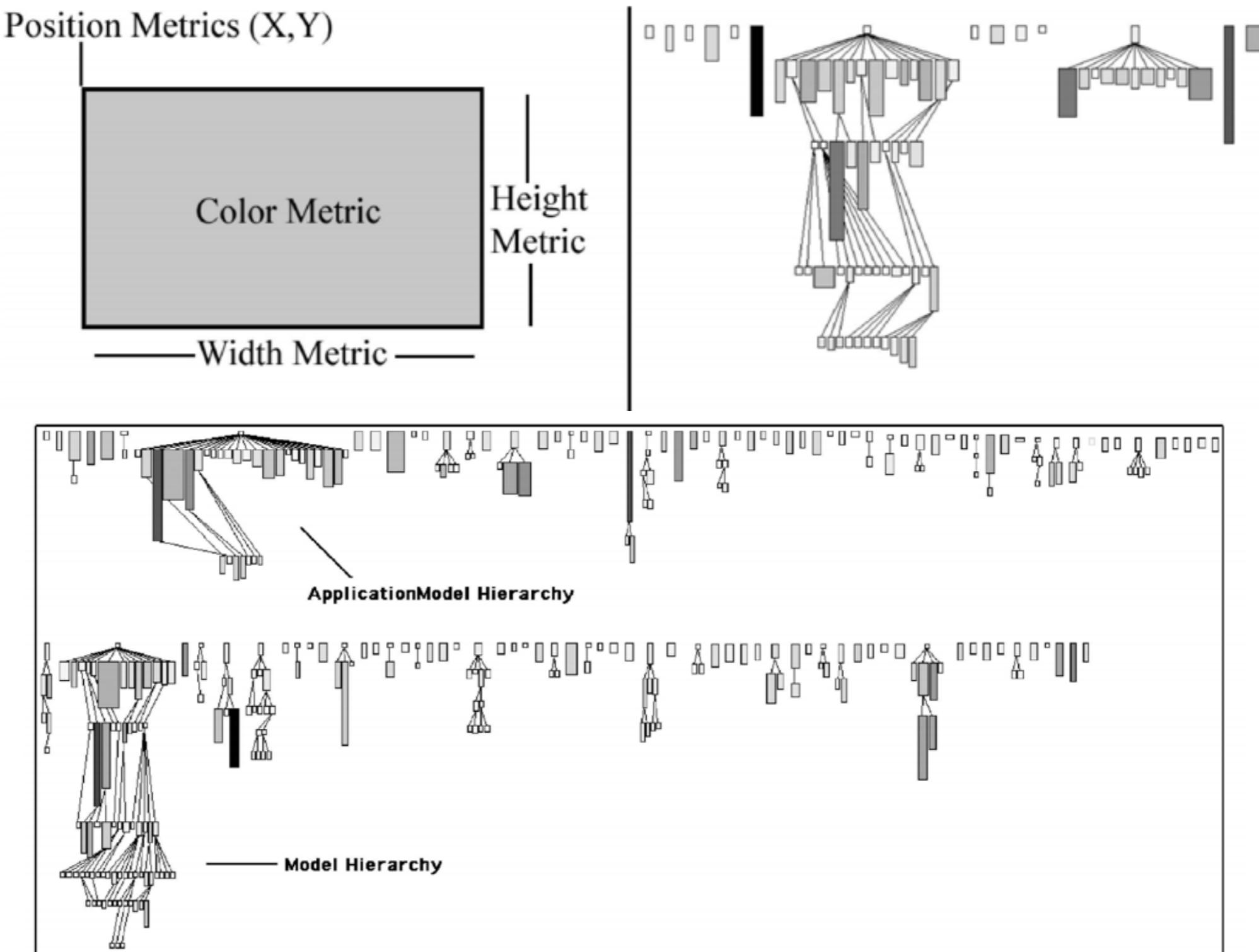


Fig. 3. A SHriMP View of a program which implements a Hangman game. The main subsystems (Control, Setup, GamePlay, Input, Output, GlobalVars and Die) are shown in this view. A fisheye view of the GamePlay subsystem provides more detail since it is shown larger than the other subsystems. The maintainer can browse the source code by following hyperlinks within an architectural view of the entire program.

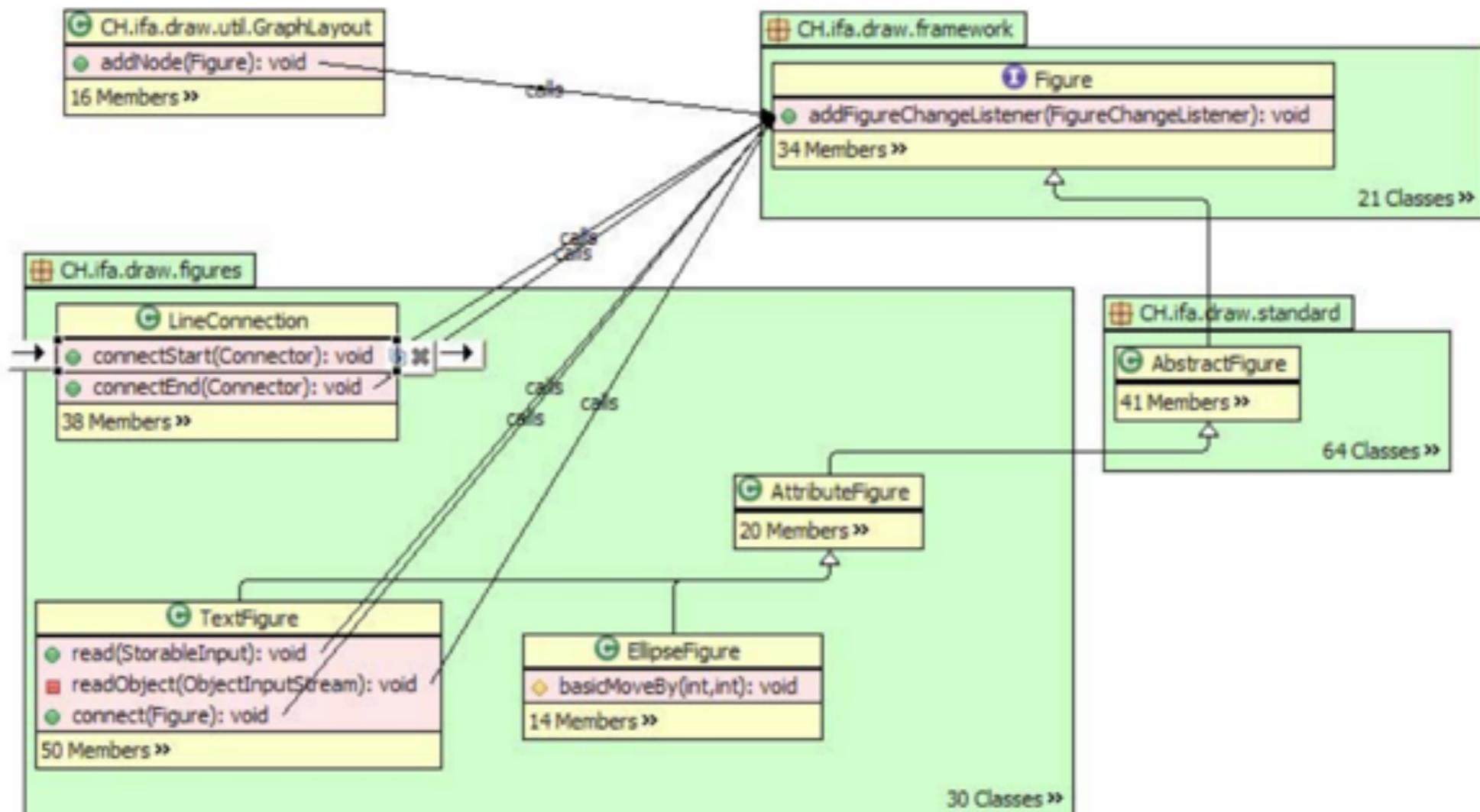
M.-A.D Storey, F.D Fracchia, H.A Müller, Cognitive design elements to support the construction of a mental model during software exploration, Journal of Systems and Software, Volume 44, Issue 3, January 1999, Pages 171-185.

Code Crawler (Polymetric Views)



Michele Lanza and Stéphane Ducasse. 2003. Polymetric Views-A Lightweight Visual Approach to Reverse Engineering. IEEE Trans. Softw. Eng. 29, 9 (September 2003), 782-795.

Relo



Vineet Sinha, David Karger, and Rob Miller. 2006. Relo: Helping Users Manage Context during Interactive Exploratory Visualization of Large Codebases. In Proceedings of the Visual Languages and Human-Centric Computing (VLHCC '06), 187-194.

Lattix (Design Structure Matrices)

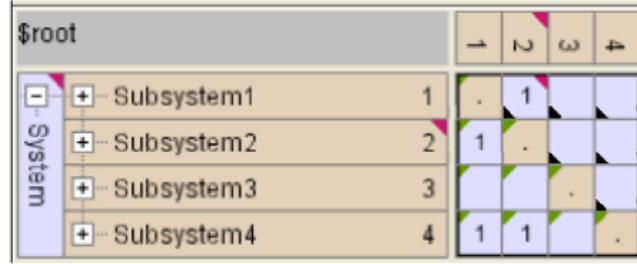


Figure 12: DSM with Rule View

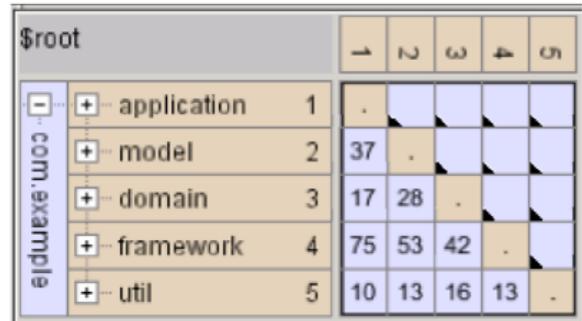


Figure 13: Design Rules for a Layered System

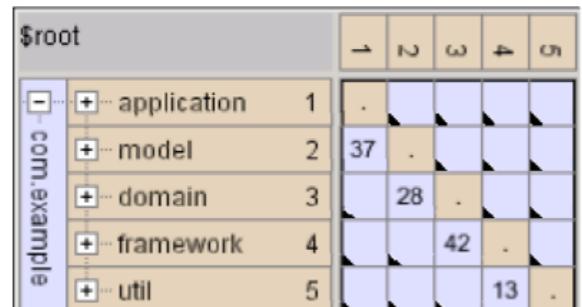


Figure 14: Design Rules for a Strictly Layered System

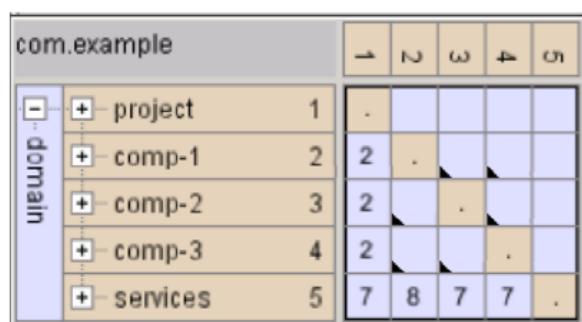
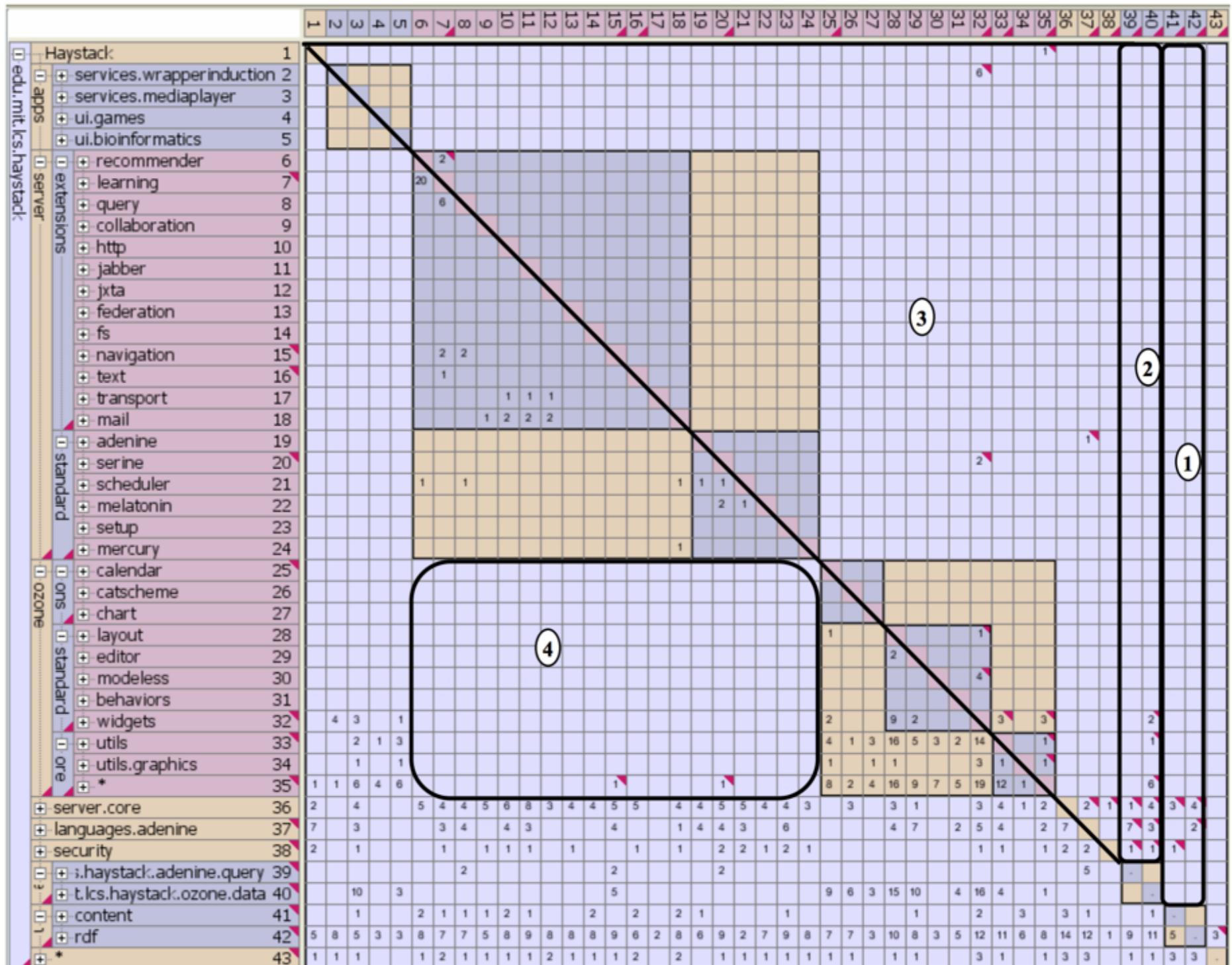


Figure 15: Design Rules for Independent Components

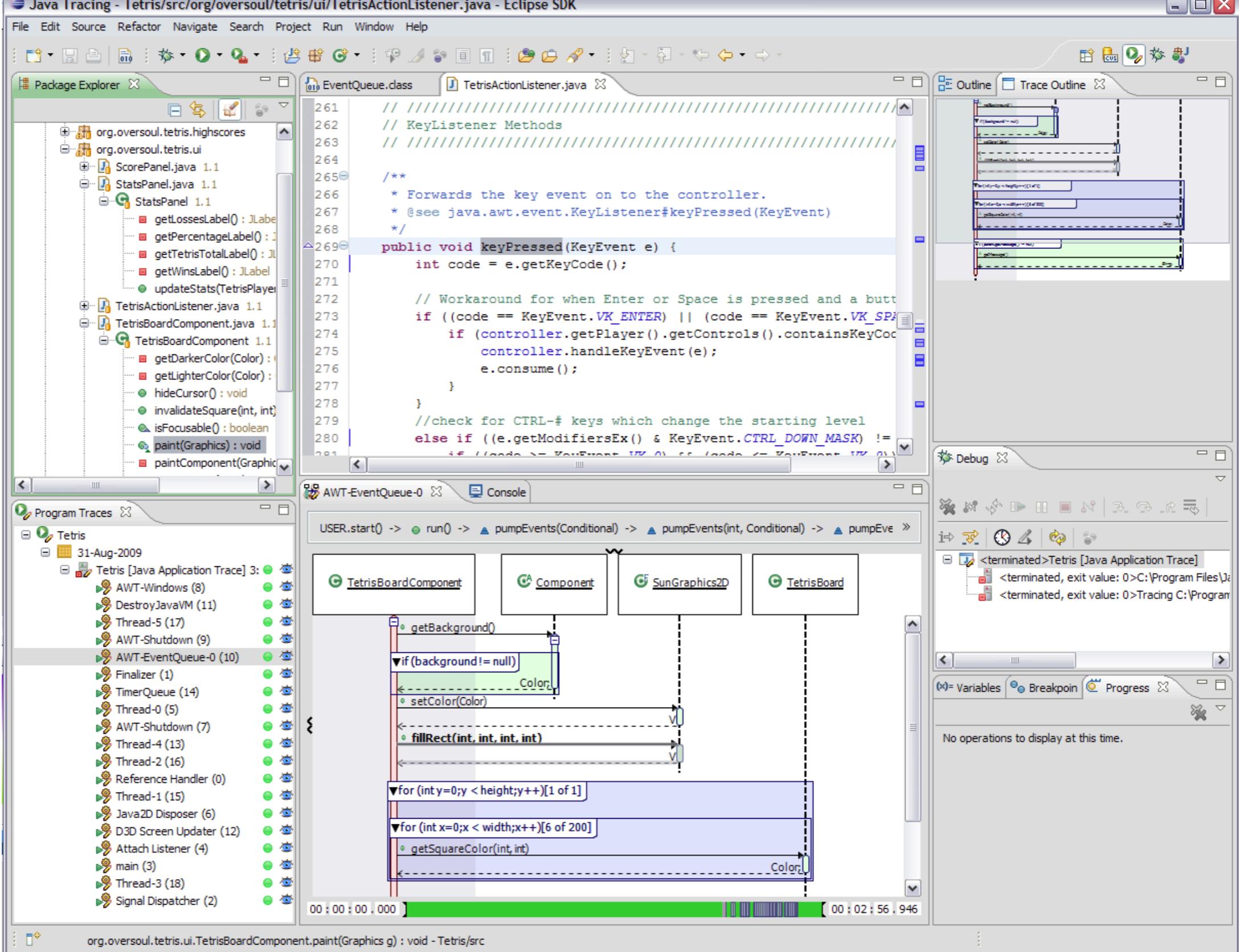


Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. 2005. Using dependency models to manage complex software architecture. Conference on Object-oriented programming, systems, languages, and applications (OOPSLA '05), 167-176.

Function calls

- Depict function invocations
- Could be runtime view (specific execution) or static view (all possible executions)
- Many decisions about what to show & how to show it
 - Code centric? Timeline centric?
 - Show all functions? Show some functions? Which ones?
 - What information about functions to depict? Order, time, asynchronicity, ...

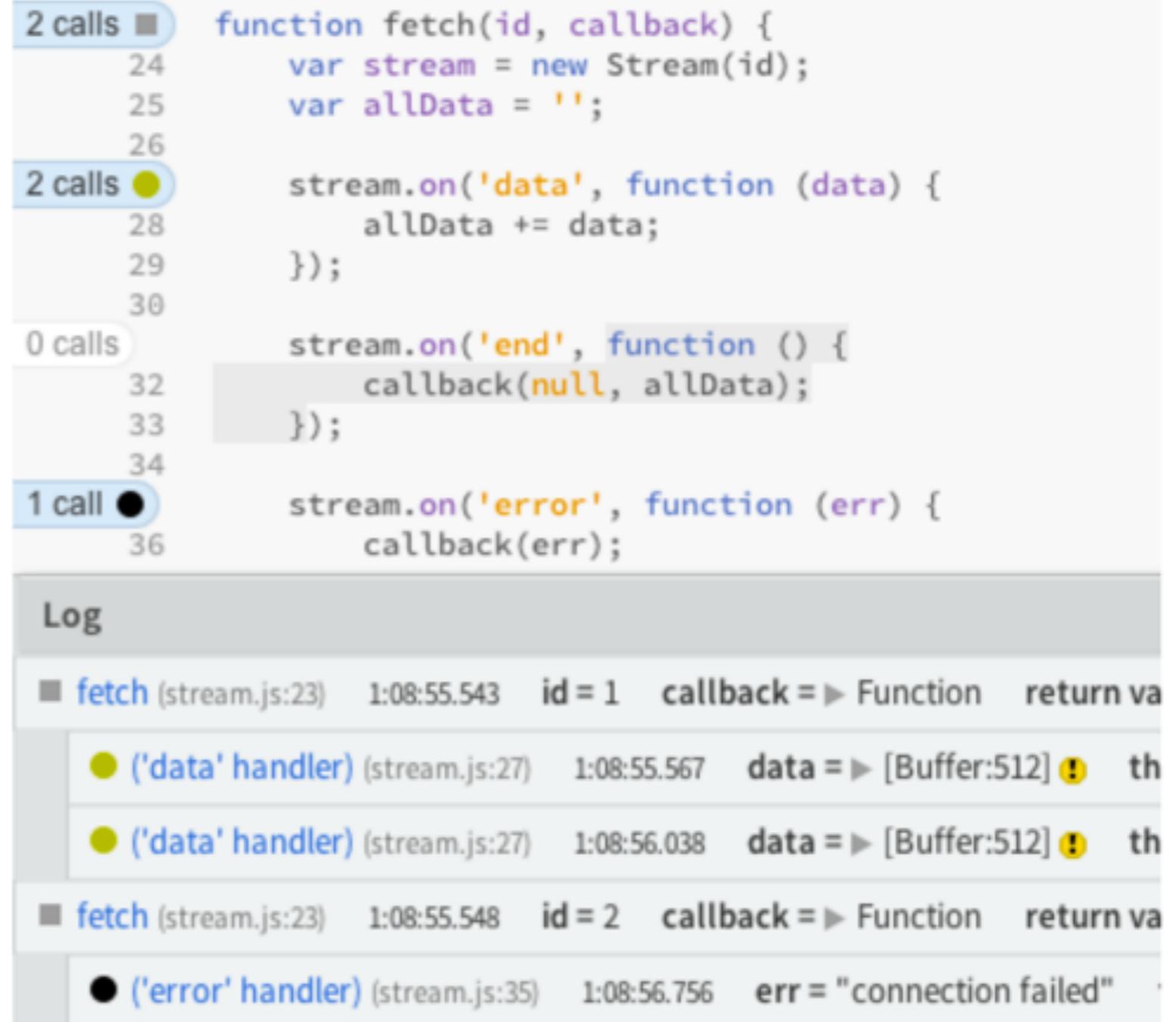
Diver



<https://www.youtube.com/watch?v=FzMI4Zu2tps>

Del Myers and Margaret-Anne Storey. 2010. Using dynamic analysis to create trace-focused user interfaces for IDEs. In Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering (FSE '10). ACM, New York, NY, USA, 367-368.

Theseus



The image shows a screenshot of the Theseus interface. At the top, there is a code editor with the following JavaScript code:

```
2 calls ■ function fetch(id, callback) {  
24     var stream = new Stream(id);  
25     var allData = '';  
26  
2 calls ● 28     stream.on('data', function (data) {  
29         allData += data;  
30     });  
0 calls 32     stream.on('end', function () {  
33         callback(null, allData);  
34     });  
1 call ● 36     stream.on('error', function (err) {  
37         callback(err);  
38     });
```

Below the code editor is a 'Log' section. It contains the following entries:

- **fetch** (stream.js:23) 1:08:55.543 id = 1 callback => Function return value
 ● ('data' handler) (stream.js:27) 1:08:55.567 data => [Buffer:512] ! th
 ● ('data' handler) (stream.js:27) 1:08:56.038 data => [Buffer:512] ! th
- **fetch** (stream.js:23) 1:08:55.548 id = 2 callback => Function return value
 ● ('error' handler) (stream.js:35) 1:08:56.756 err = "connection failed" .

Figure 1. Theseus shows call counts for every function, and an asynchronous call tree allows the user to see how functions interact. In the log below the code, users can see which call to `fetch` corresponds to the failure without adding any debugging-specific code or re-executing their program.

<https://www.youtube.com/watch?v=qnwXX510E2Q>

Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. Conference on Human Factors in Computing Systems, 2481-2490.

WhyLine

The screenshot illustrates the WhyLine interface for Java code analysis. The main window displays a Java file, `PaintWindow.java`, with annotations and a timeline.

Annotations:

- a**: A yellow box highlights the value `Color #19,941` in the code.
- b**: A callout box shows the question `(↑) why did this execute?` and the answers `(1) why did color = rgb(0,0,0)? (producer)` and `(2) why did this = PencilPaint #25,299? (producer)`.
- c**: A green box highlights the `Color()` constructor in the timeline.
- d**: A callout box shows the question `(↑) why did this execute?` and the answers `(1) why did getValue() return 0? (producer)`, `(2) why did getValue() return 0? (producer)`, and `(3) why did getValue() return 0? (producer)`.
- g**: A callout box shows the question `(↑) why did this execute?` and the answers `(1) why did getValue() return 0? (producer)`, `(2) why did getValue() return 0? (producer)`, and `(3) why did getValue() return 0? (producer)`.

Code Editor (PaintWindow.java):

```

source      thread EventQueue0-5
private PaintObjectConstructor objectConstructor;
private ChangeListener colorChangeListener;
public void stateChanged(ChangeEvent changeEvent) {
    objectConstructor.setColor(
        new Color(
            rSlider.getValue(),
            gSlider.getValue(),
            bSlider.getValue()));
    repaint();
}

```

Timeline:

The timeline shows the execution flow from the `start of program` to the `Color()` constructor, with annotations **c** and **g** pointing to specific events.

Call Stack:

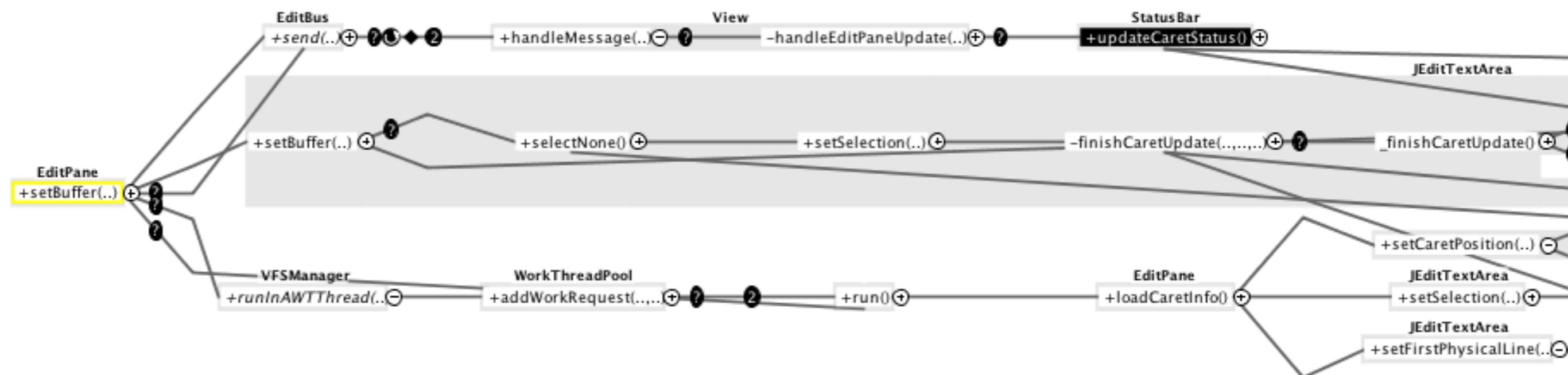
The call stack shows the execution path from the `start of program` to the `Color()` constructor, with annotations **e** and **g** pointing to specific frames.

Search and Help:

Annotations **f** and **g** point to the search bar and help text respectively.

Andrew J. Ko and Brad A. Myers. 2009. Finding causes of program output with the Java Whyline. In Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '09). ACM, New York, NY, USA, 1569-1578.

Reacher



T. D. LaToza and B. A. Myers, "Visualizing call graphs," 2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC), 2011, pp. 117-124.

In Class Activity

- Form groups of 2
 - Sketch a software visualization
 - You should decide
 - What is the task you are supporting
 - What information do developers need for this task
 - How does your visualization help developers to obtain this information more easily
 - What context is (or is not) visualized? Why is the specific visualization chosen?
 - Illustrate your visualization with two or more examples of its output