Software Visualization

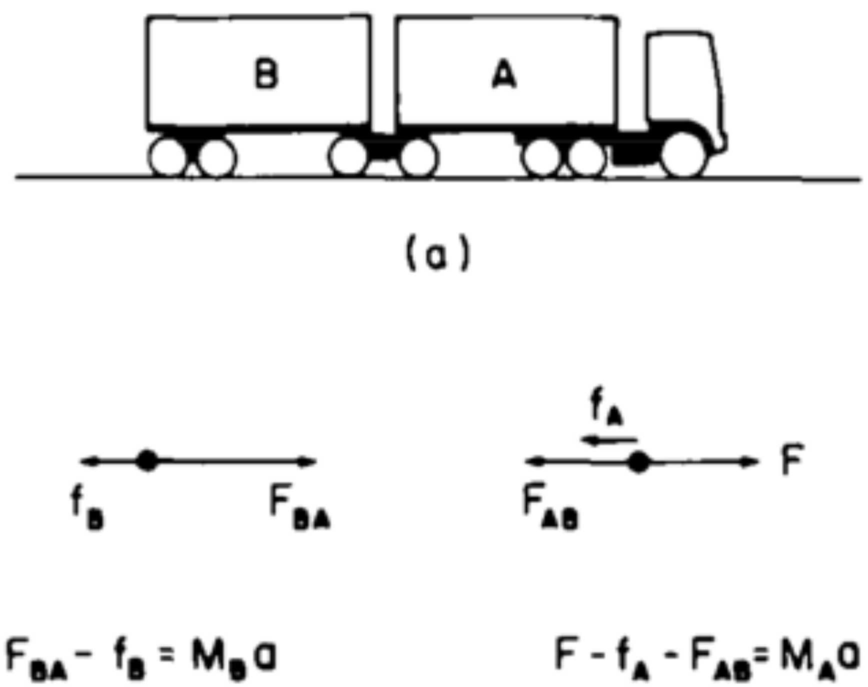
SWE 795, Fall 2019

Software Engineering Environments

Today

- Part 1 (Lecture)(~60 mins)
    - Software visualization

Why a diagram is (sometimes) worth ten thousand words



- 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

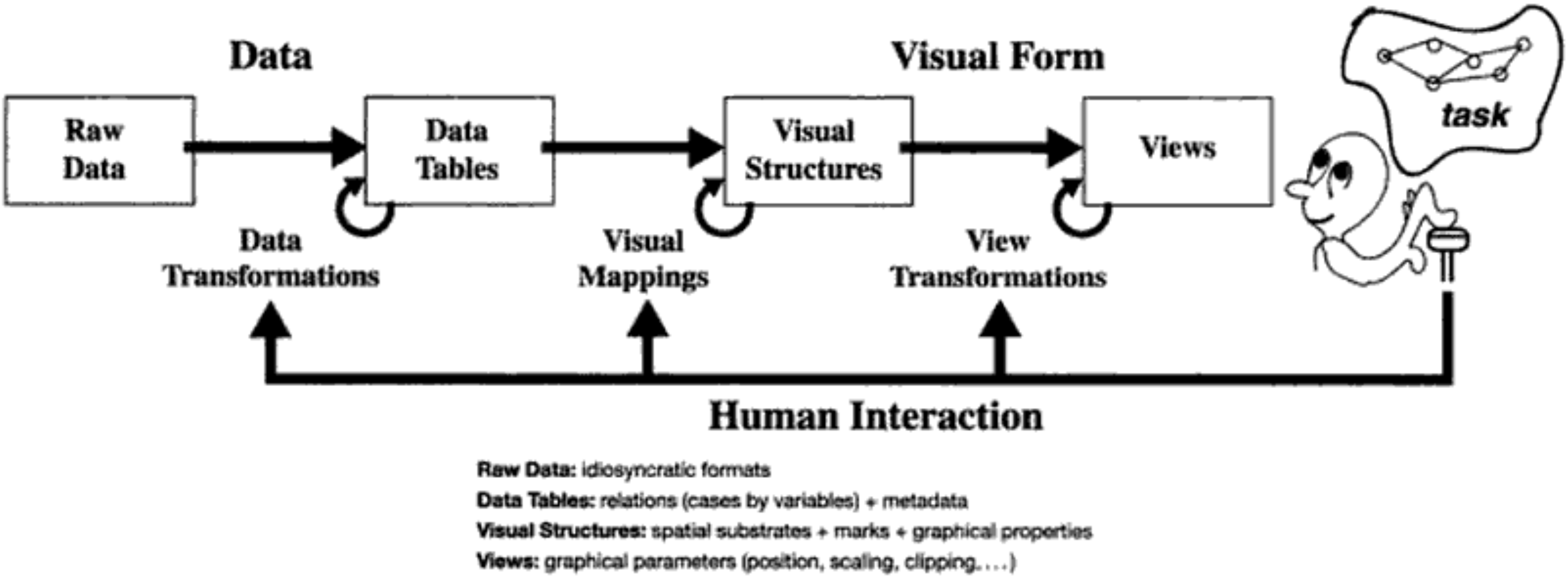
- Larkin & Simon, 1987, Cognitive Science 11, pp 65-99.

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 aerial.
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.

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

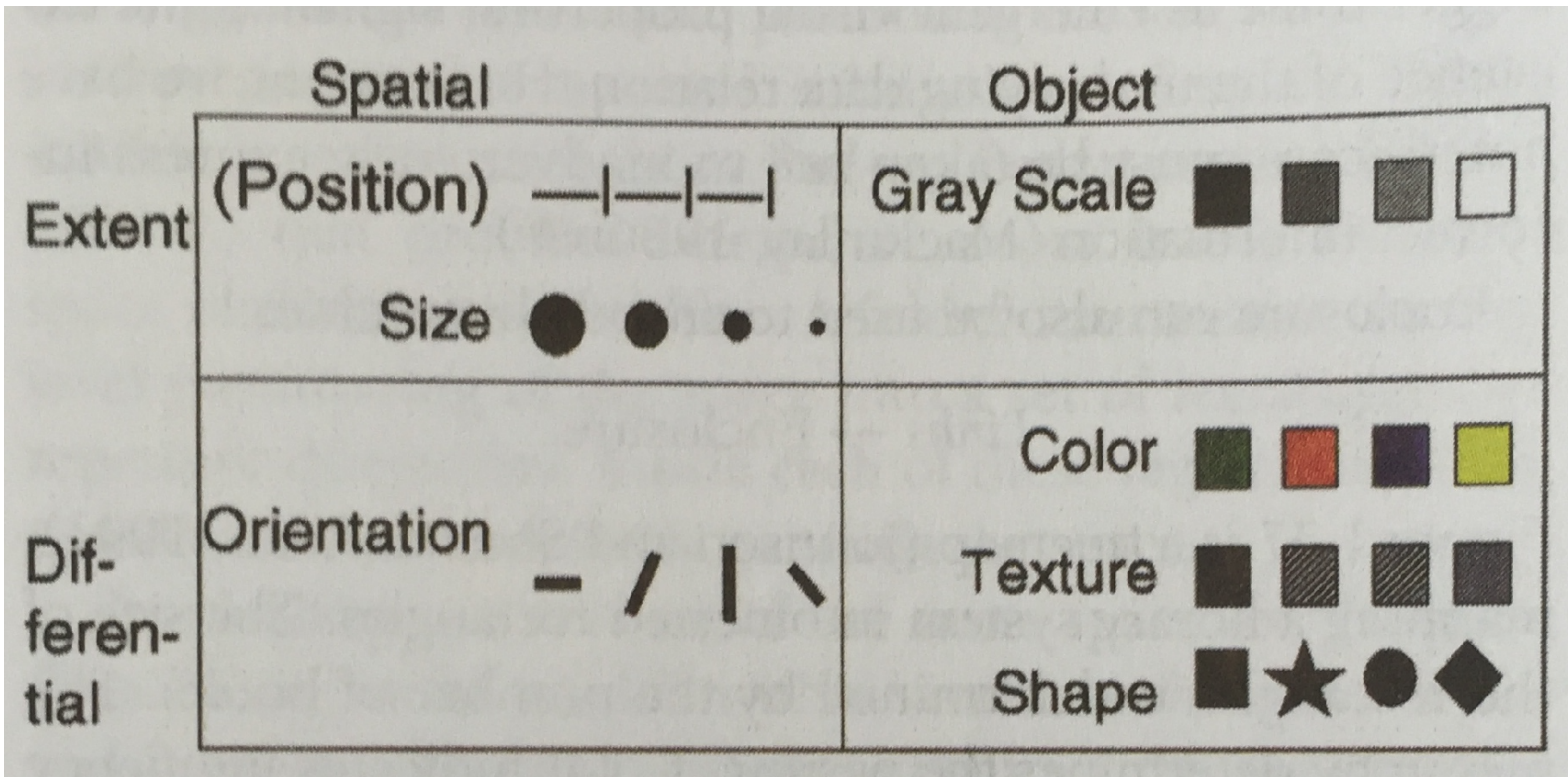
Designing an information visualization



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

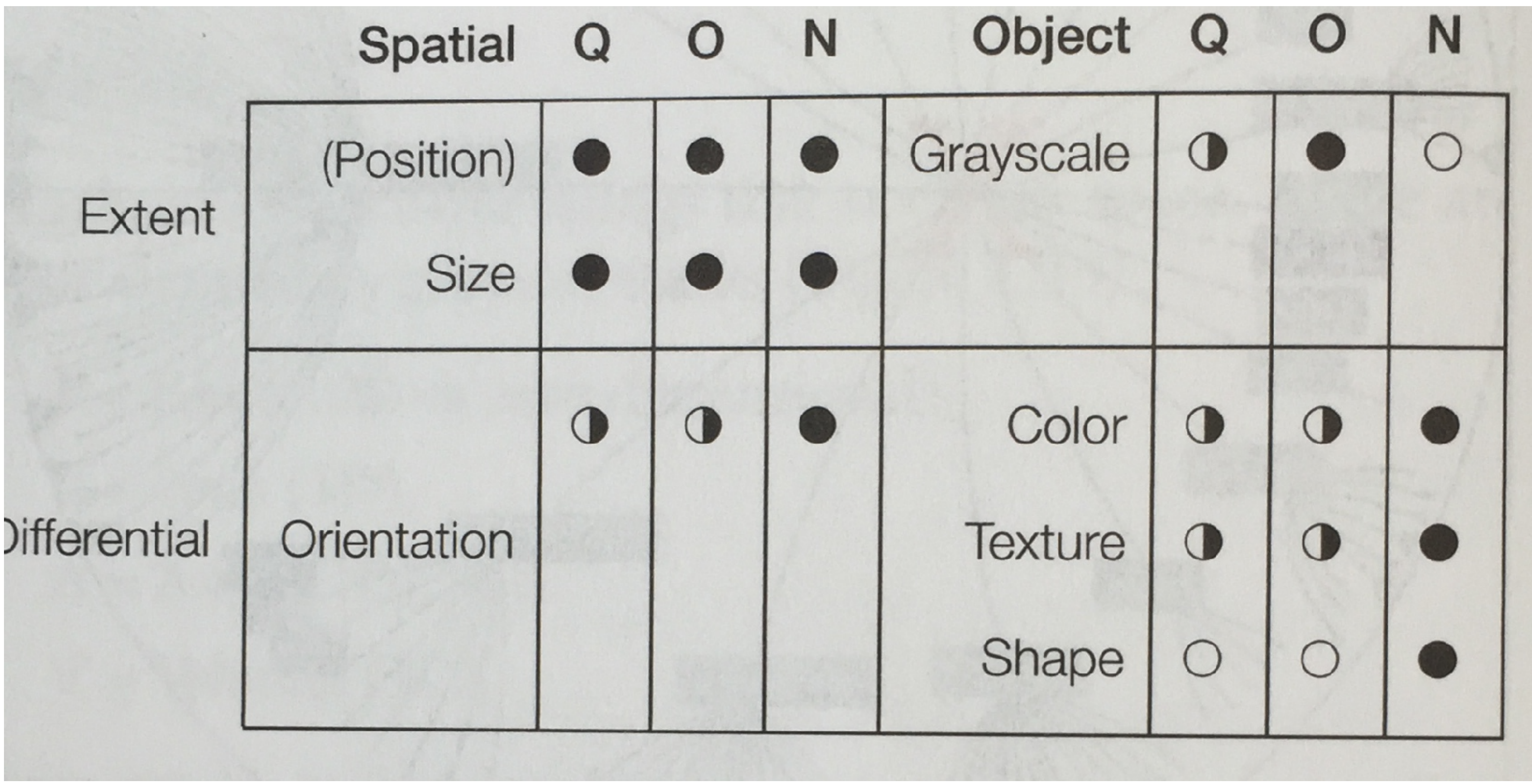
Marks' graphical properties

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



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)

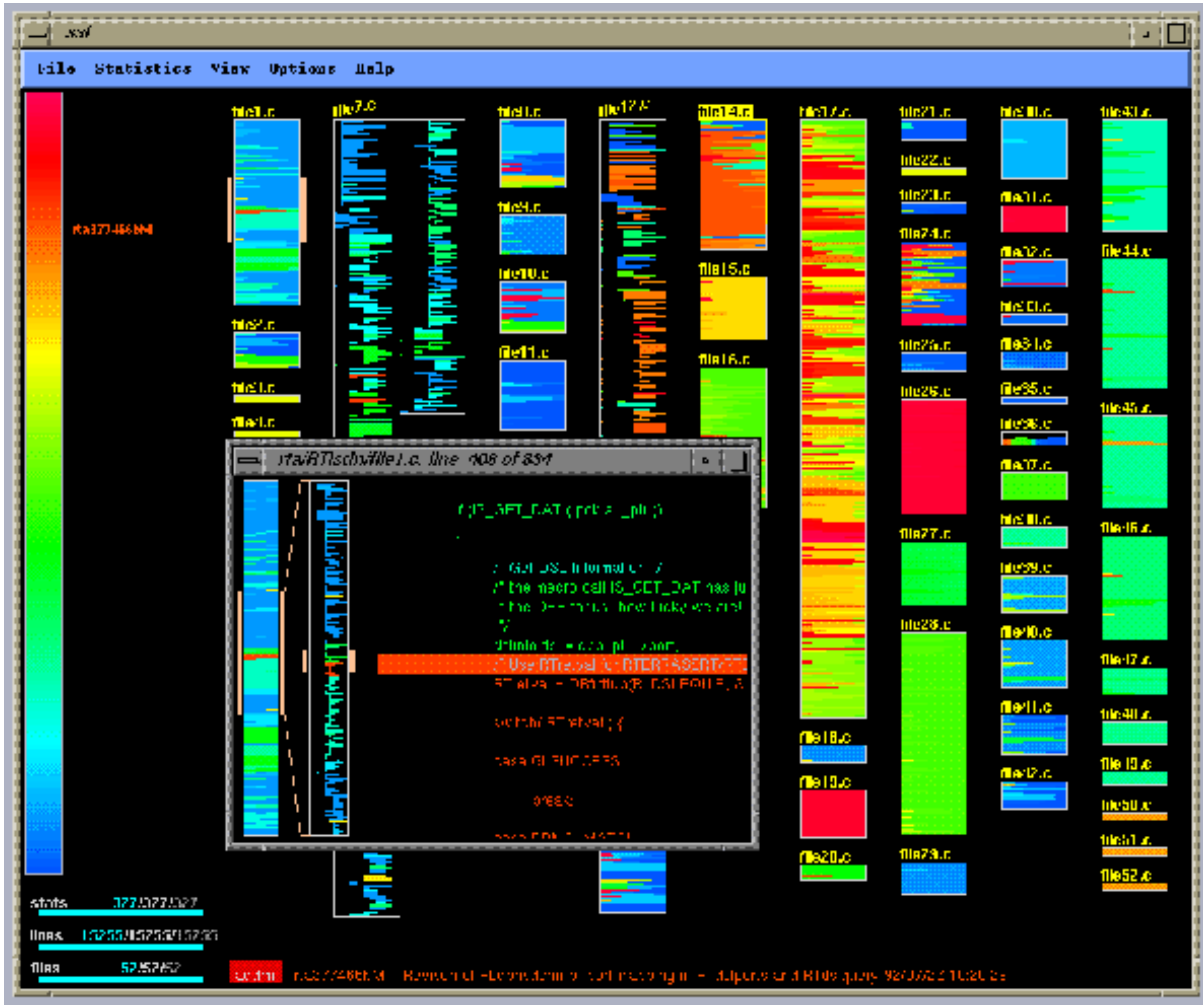
- Function calls
  - Dynamic and static depictions of function calls

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 iconagraphically
  - e.g., colored lines

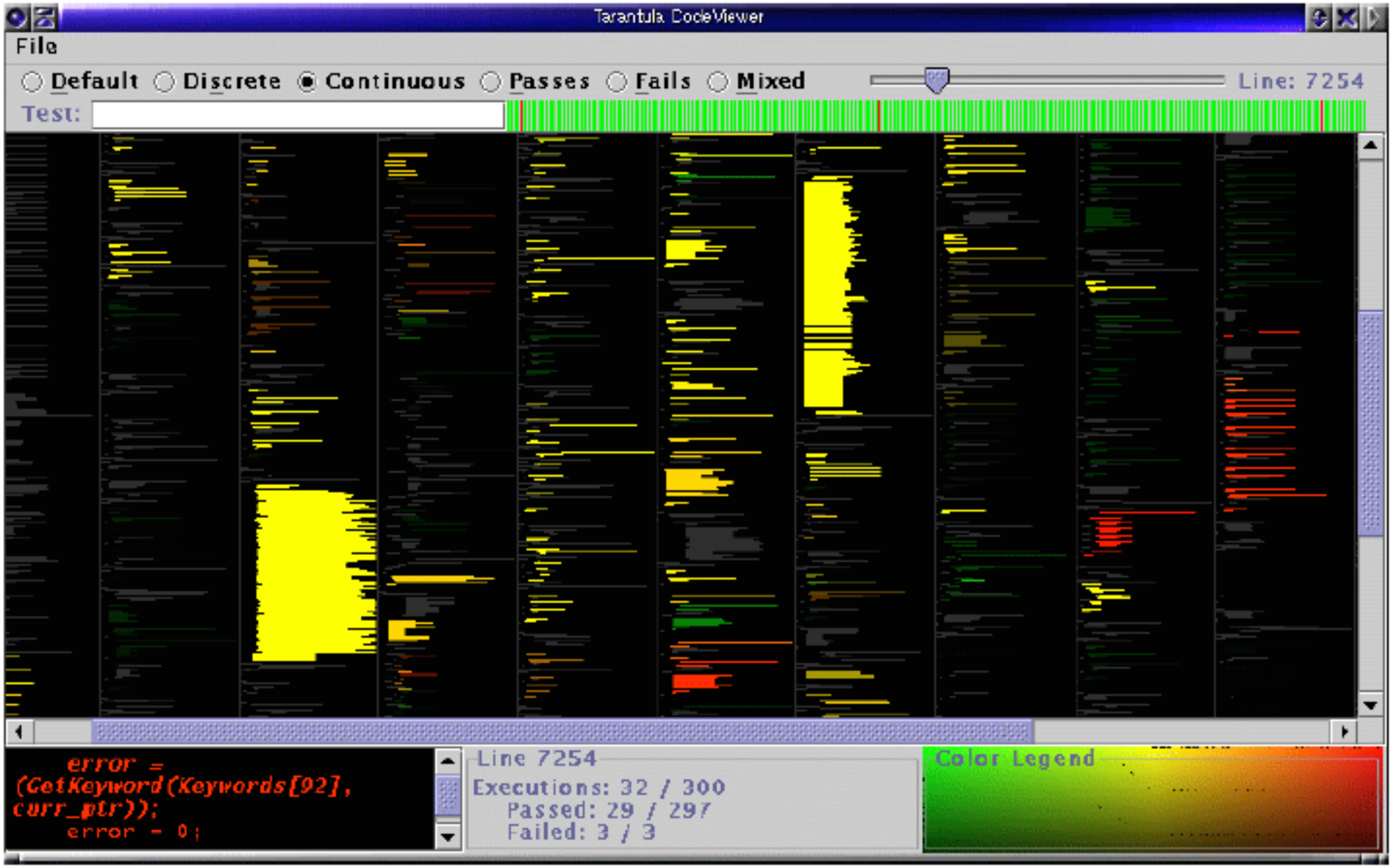
SeeSoft

- 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.

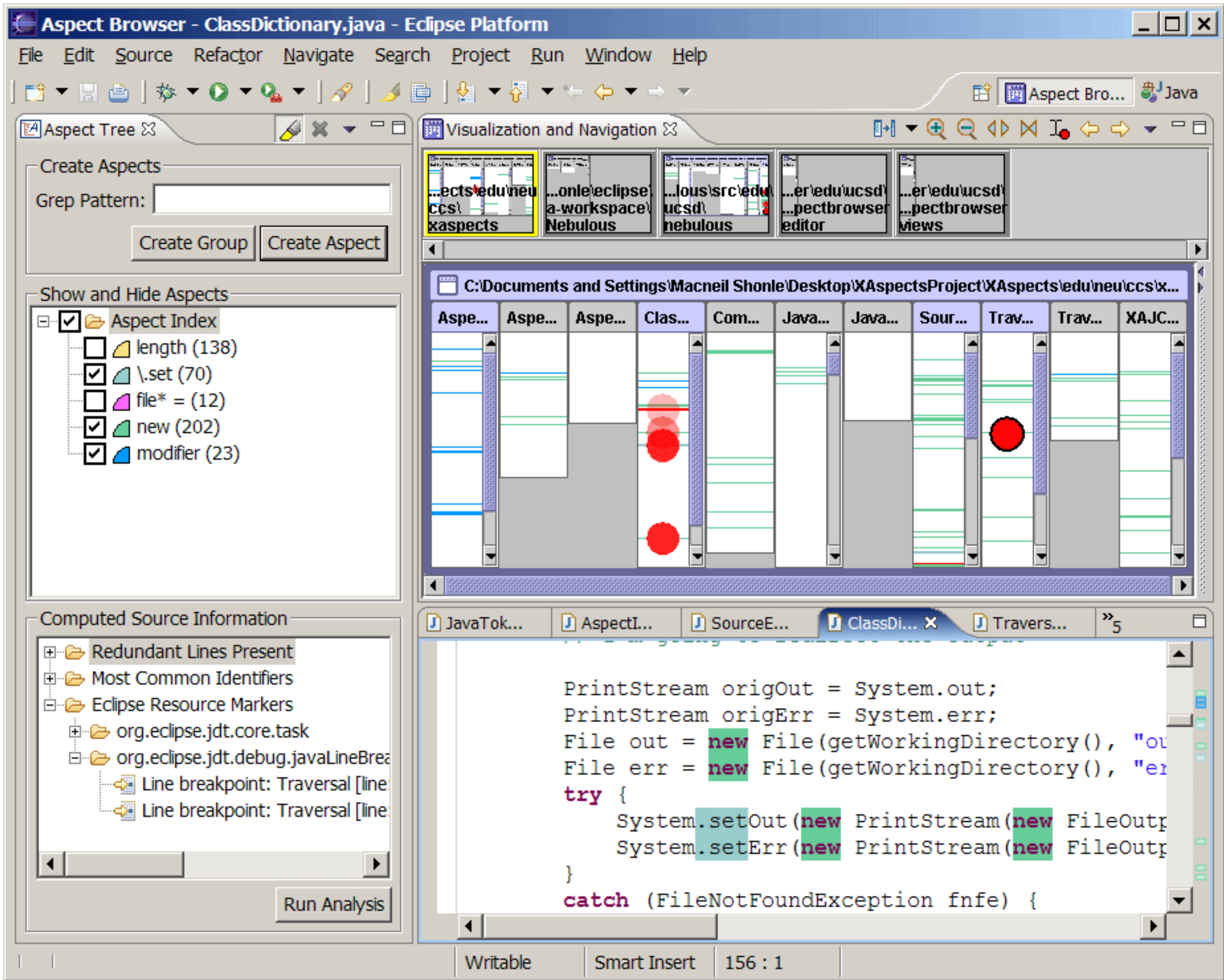
Tarantula



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

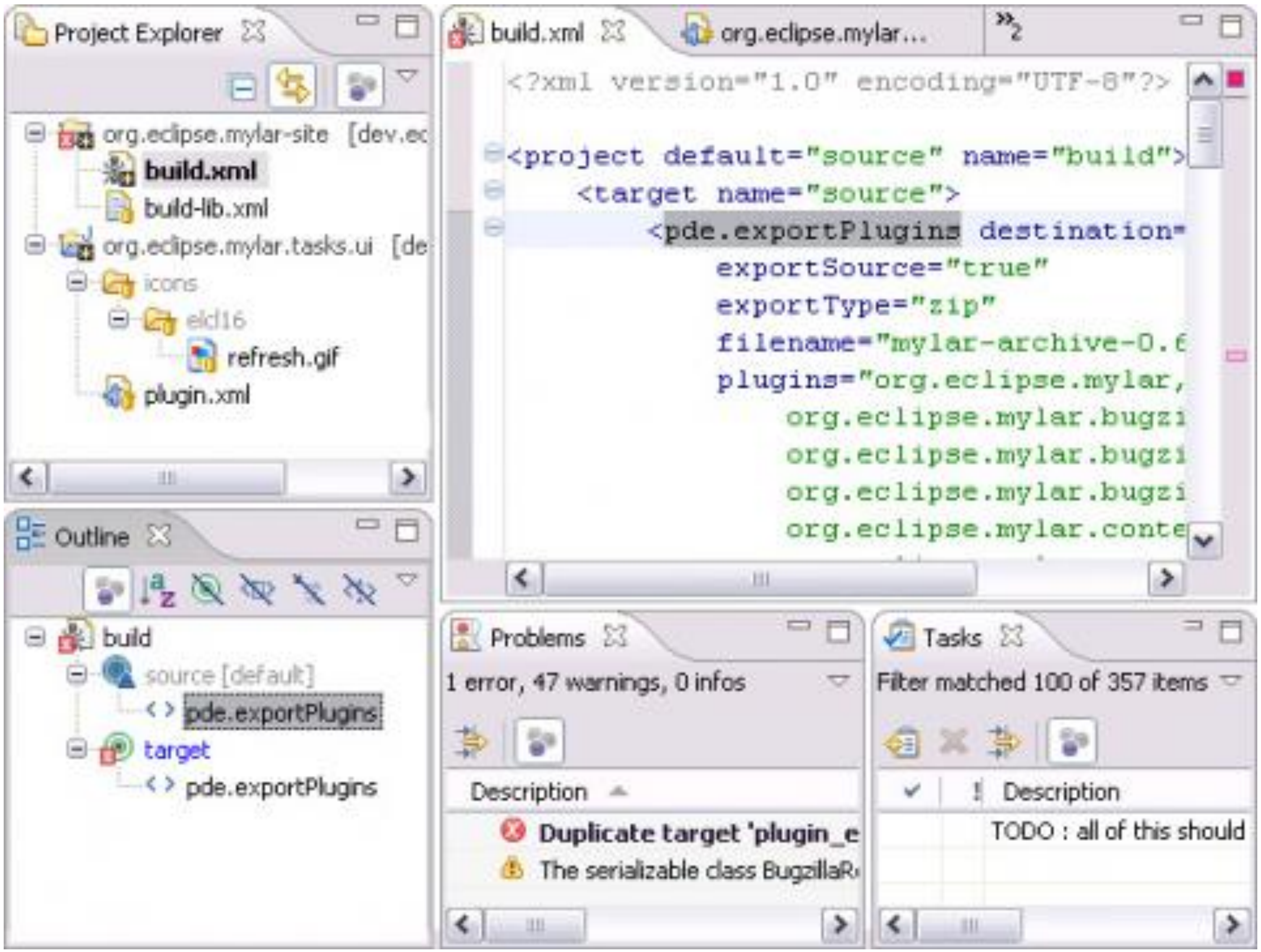
James A. Jones, Mary Jean Harrold, and John Stasko. 2002. Visualization of test information to assist fault localization. International Conference on Software Engineering (ICSE '02), 467-477.

AspectBrowser

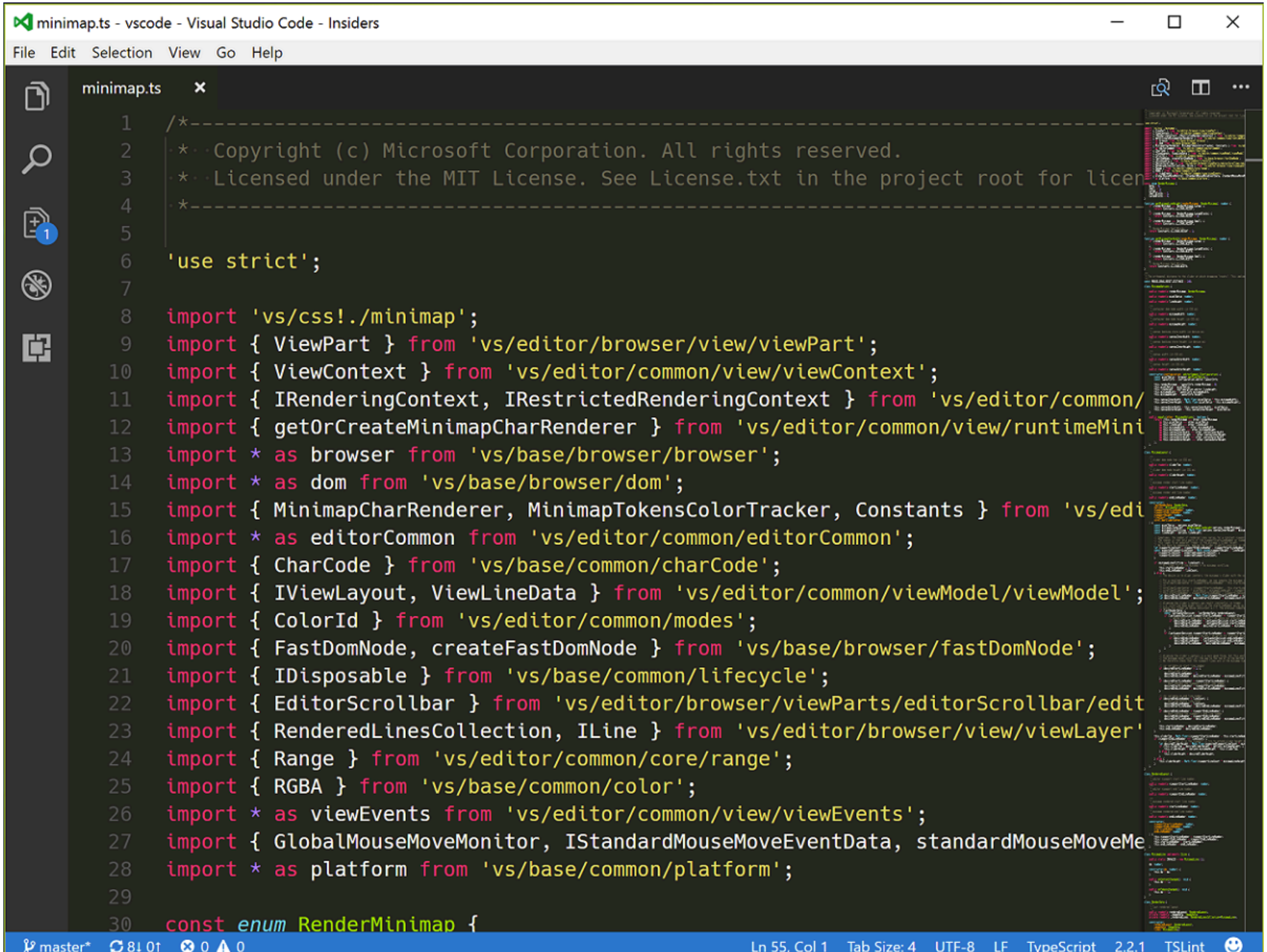


Macneil Shonle, Jonathan Neddenriep, and William Griswold. 2004. AspectBrowser for Eclipse: a case study in plug-in retargeting. In Proceedings of the 2004 OOPSLA workshop on eclipse technology eXchange (eclipse '04). ACM, New York, NY, USA, 78-82.

Industry Use: Eclipse Markers



Industry use: Visual Studio Code Minimap



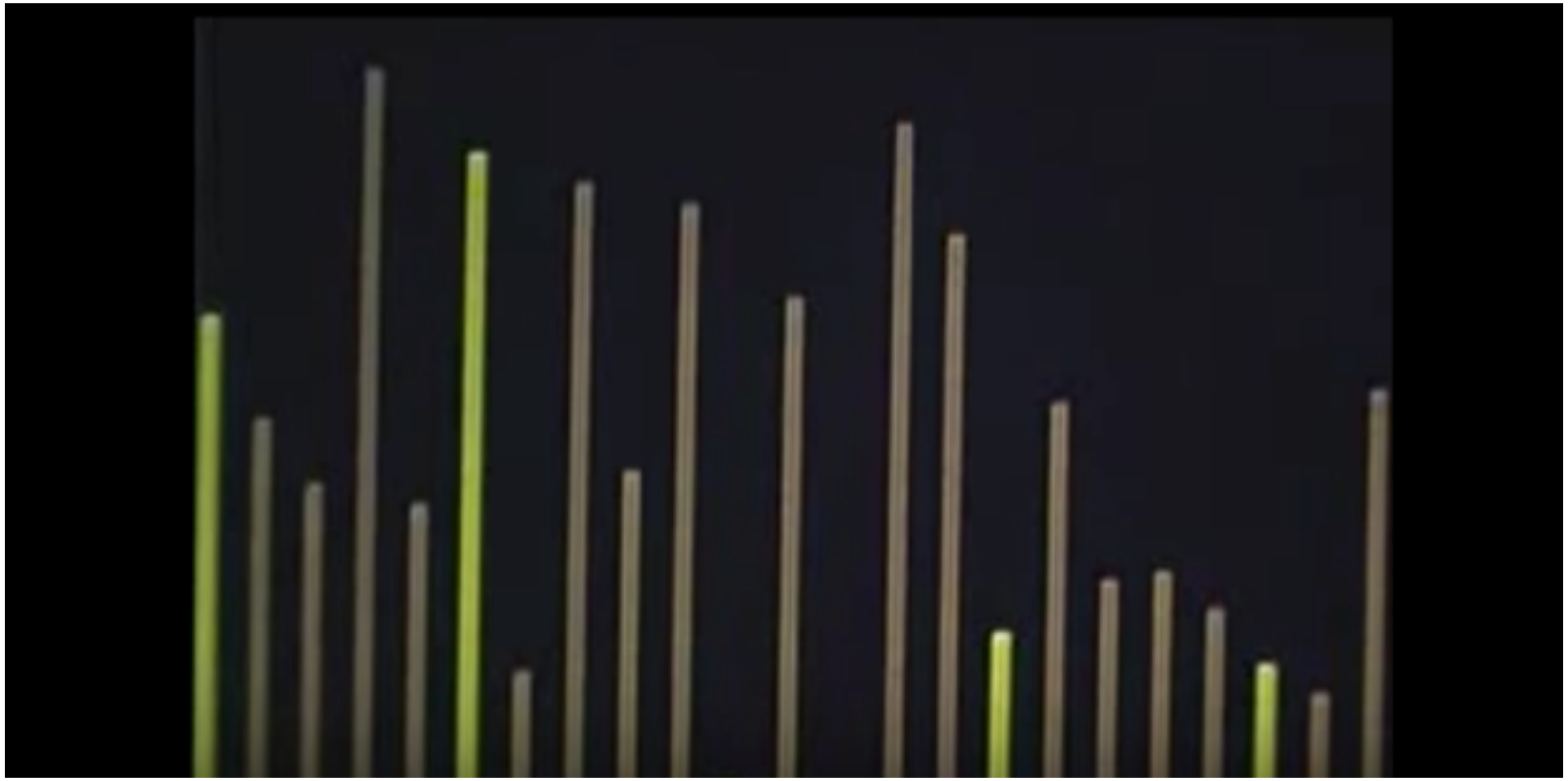
Algorithm & object structure visualizations

- 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 drawn them on a blackboard"

Goal: help with debugging

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

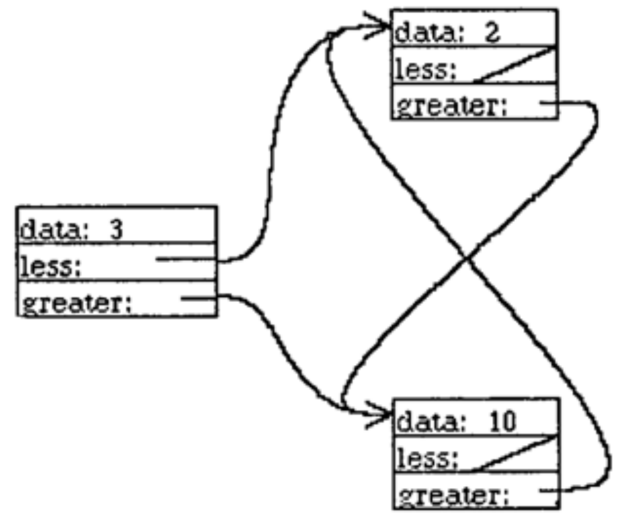


Figure 17.
Incense display for
RECORD [int: INTEGER, p1: POINTER TO CARDINAL].

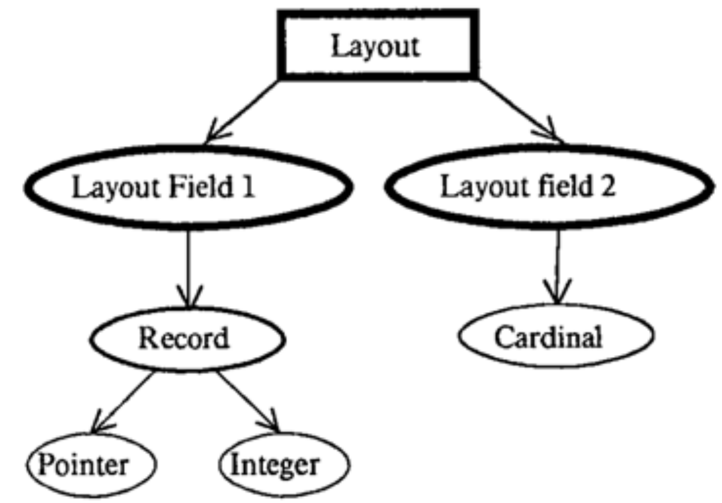


Figure 18.
Artist hierarchy that would be created for:
rec: RECORD [p1: POINTER TO CARDINAL, int: INTEGER];
(This figure was not created by Incense).

Figure 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.

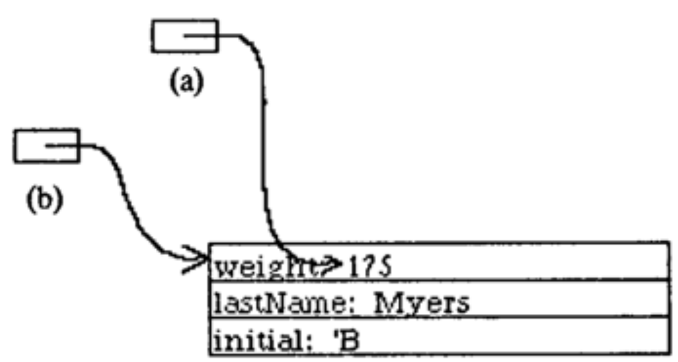


Figure 16.
Pointer to value inside a record (a) does not get confused with a pointer to the record itself (b).

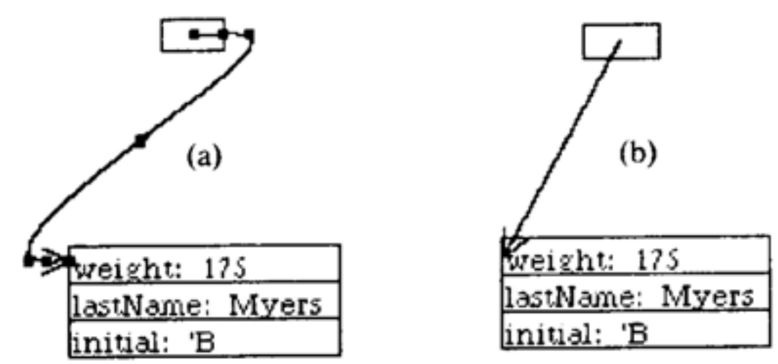
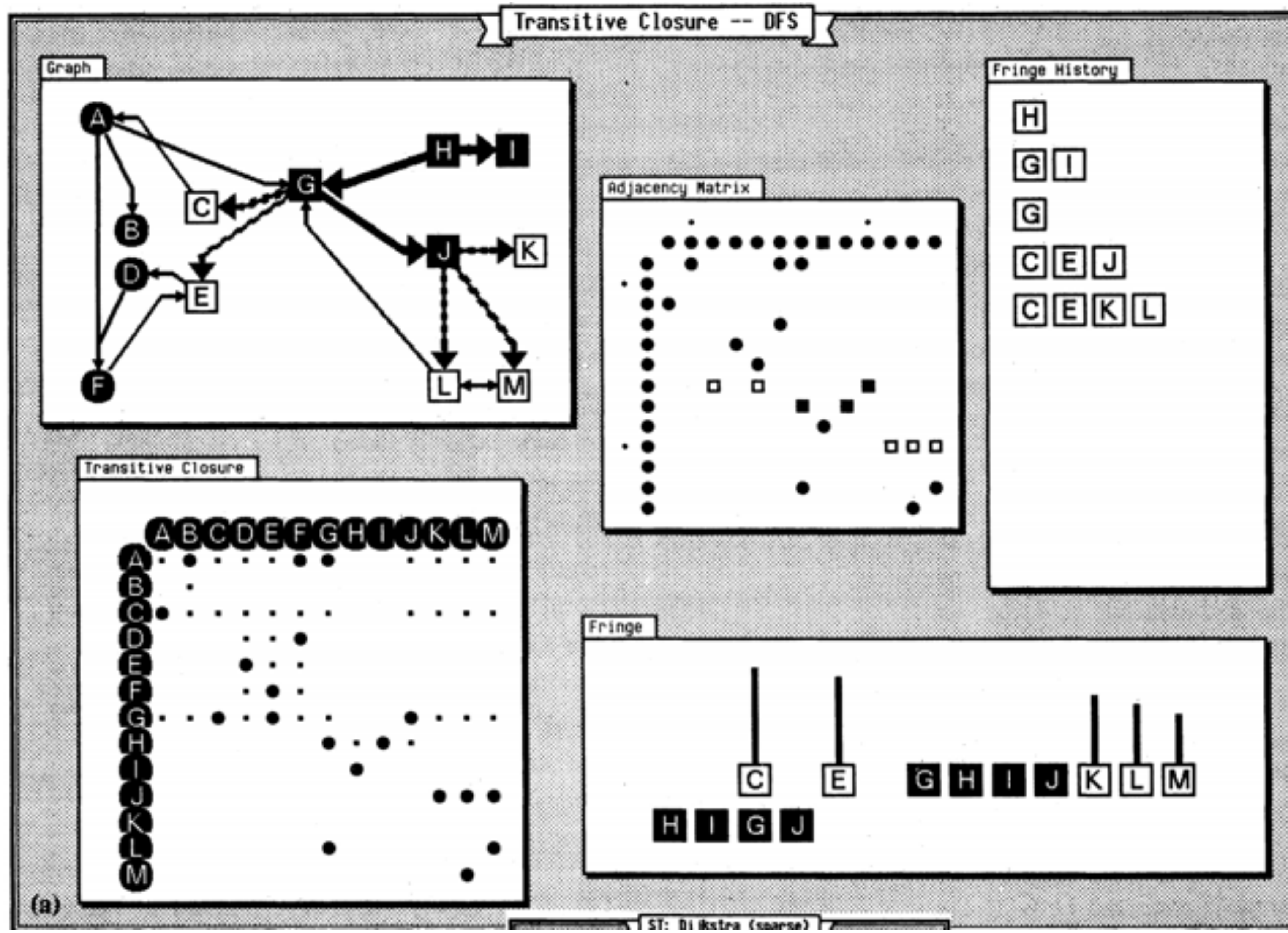


Figure 19.
Demonstration of the advantage of curved lines used in Incense (a) over straight lines (b). The control points used to specify the spline are shown as black squares in (a).

Brad A. Myers. 1983. INCENSE: A system for displaying data structures. Conference on Computer graphics and interactive techniques (SIGGRAPH '83),115-125.

Brown University Algorithm Simulator and Animator (BALSA)

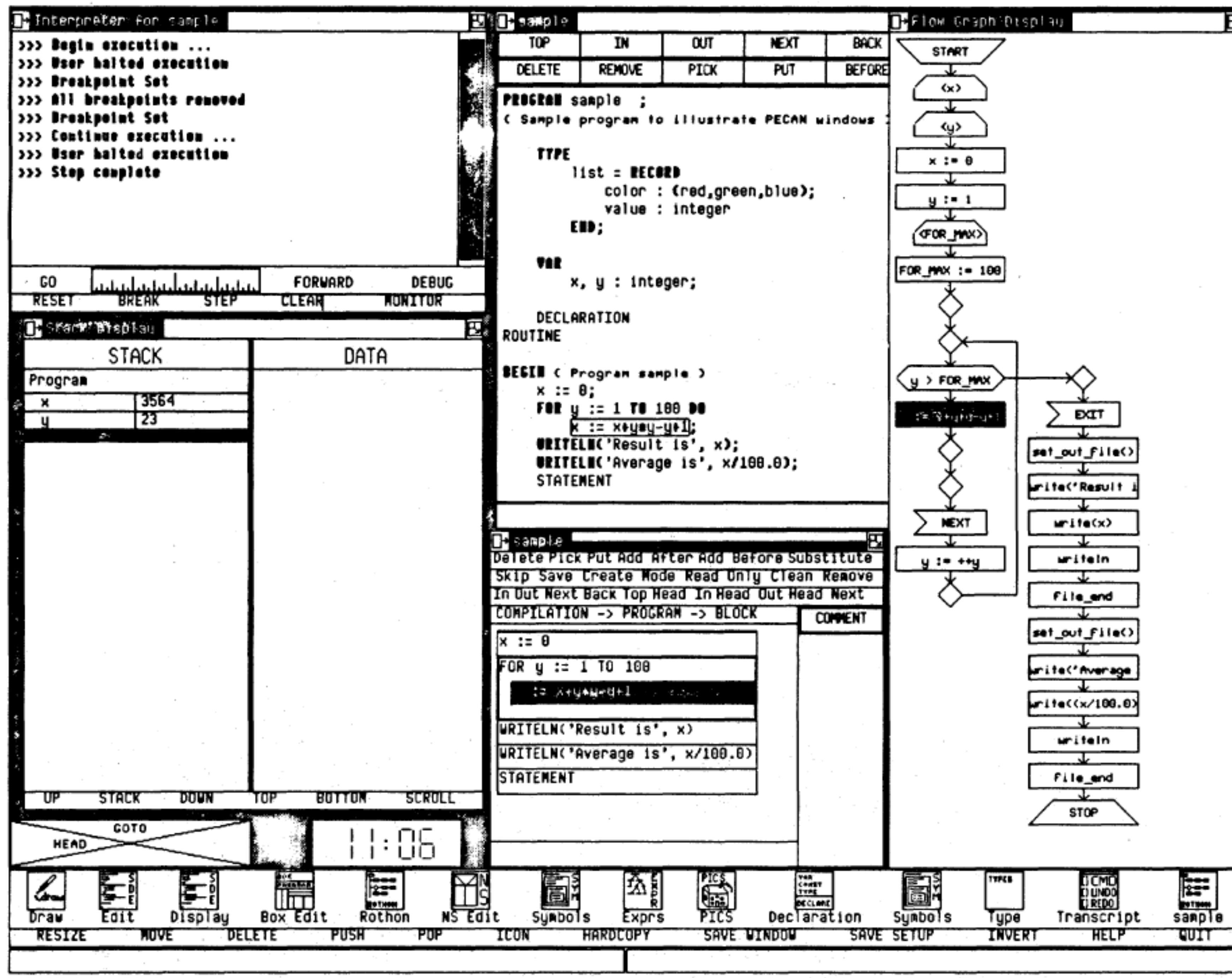
- Major interactive integrated system
  - Extensively used for teaching at Brown Univ.
  - Lots of algorithms visualized
  - Architecture for attaching the graphics with code
    - Still required significant programming for each viz.
  - Marc followed up with Zeus ('91) at DEC SRC



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

PECAN

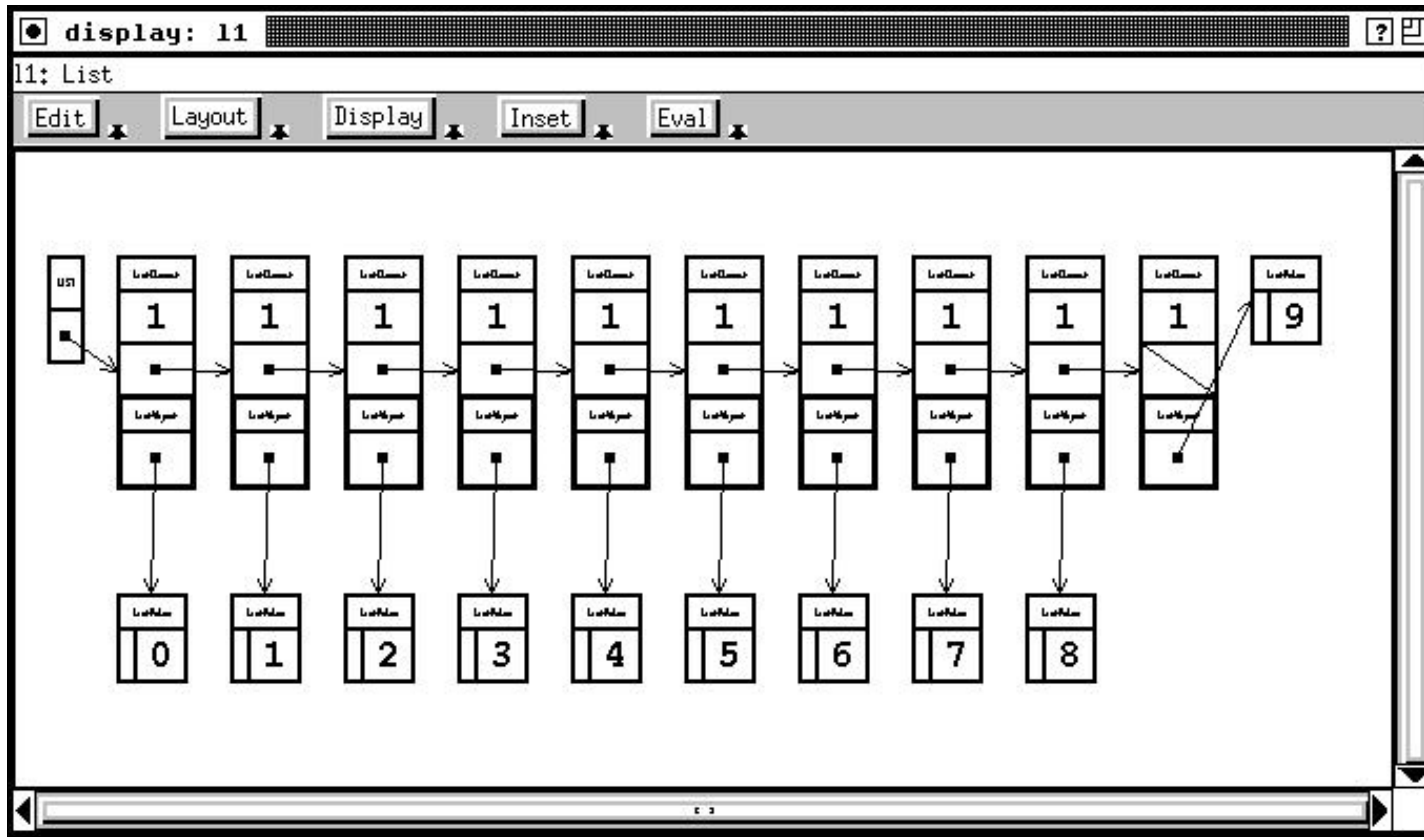
- Steven Reiss at Brown's code & data visualization systems
  - Take advantage of new Apollo workstation capabilities
- PECAN (1985) – automatic graphics about the program
  - Multiple views
  - Integrates Balsa data visualization
  - Syntax directed editing
    - Drag and drop
  - Flowcharts of code
  - 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

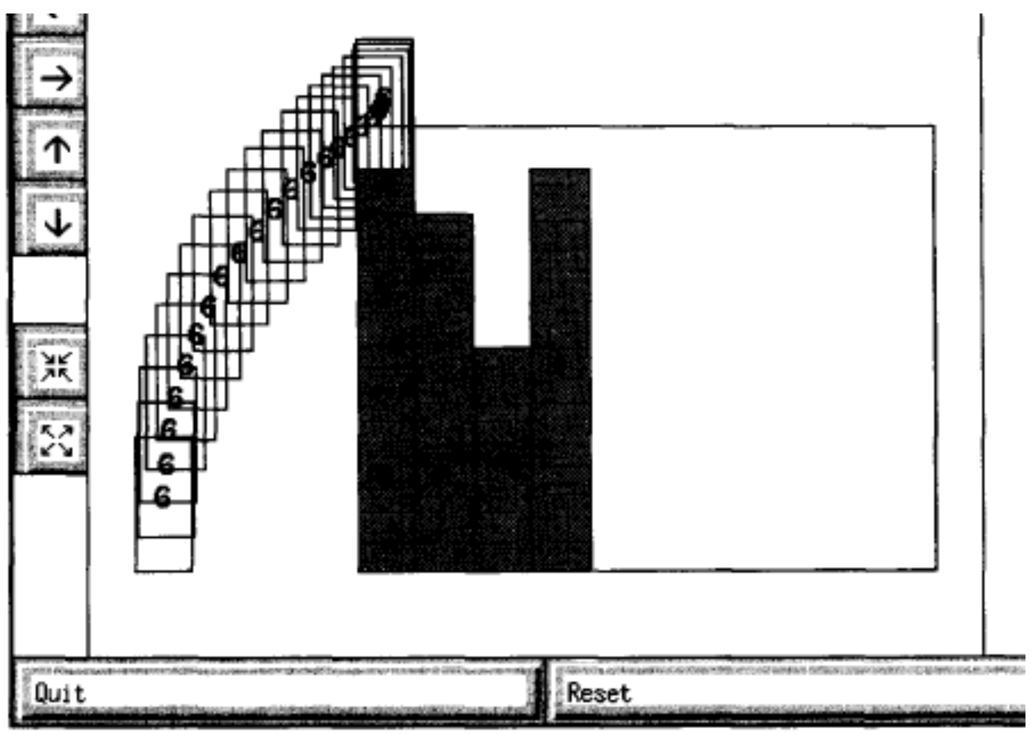


Figure 9. Superimposed sequence of frames from the bin-packing animation.

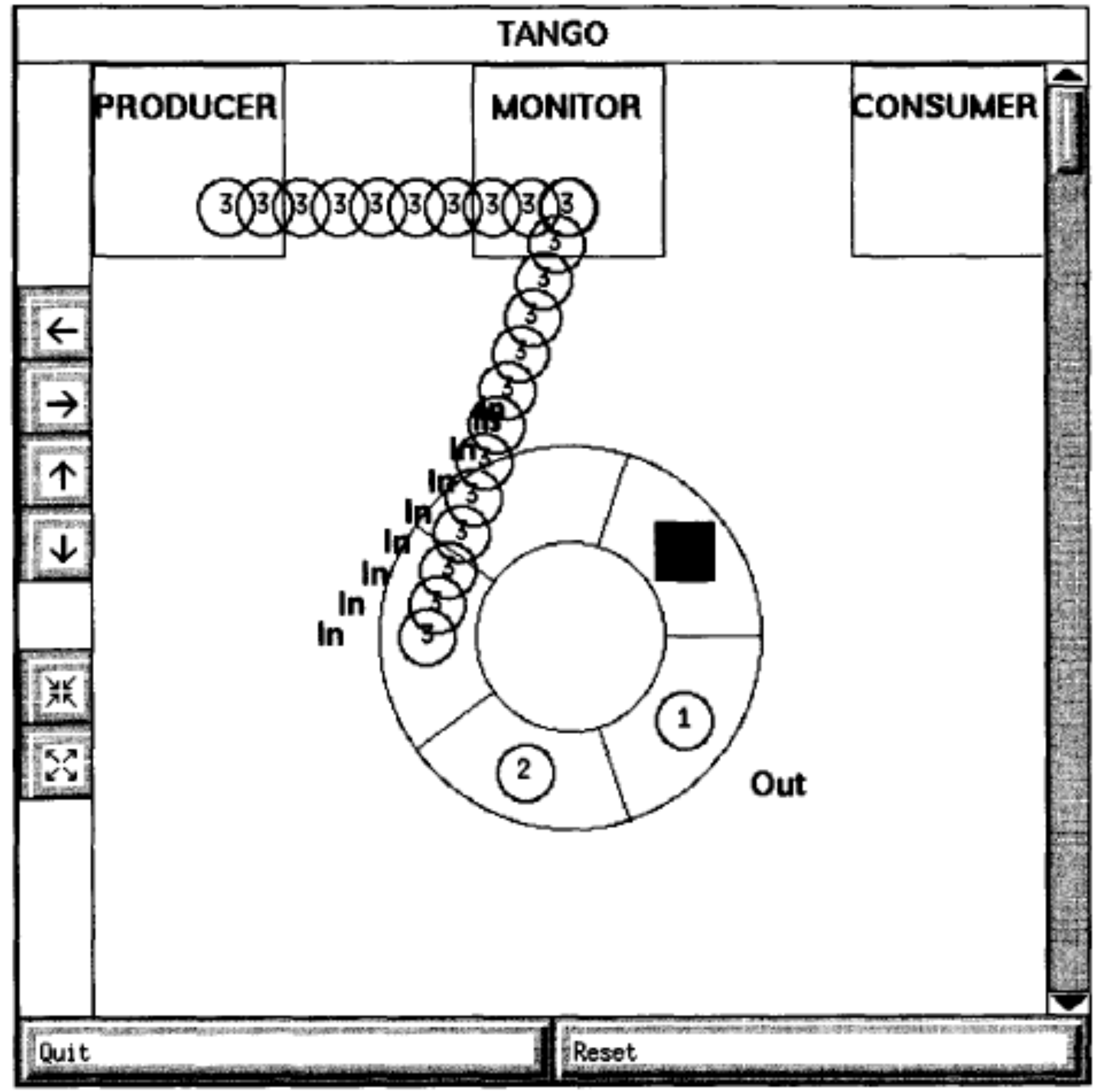
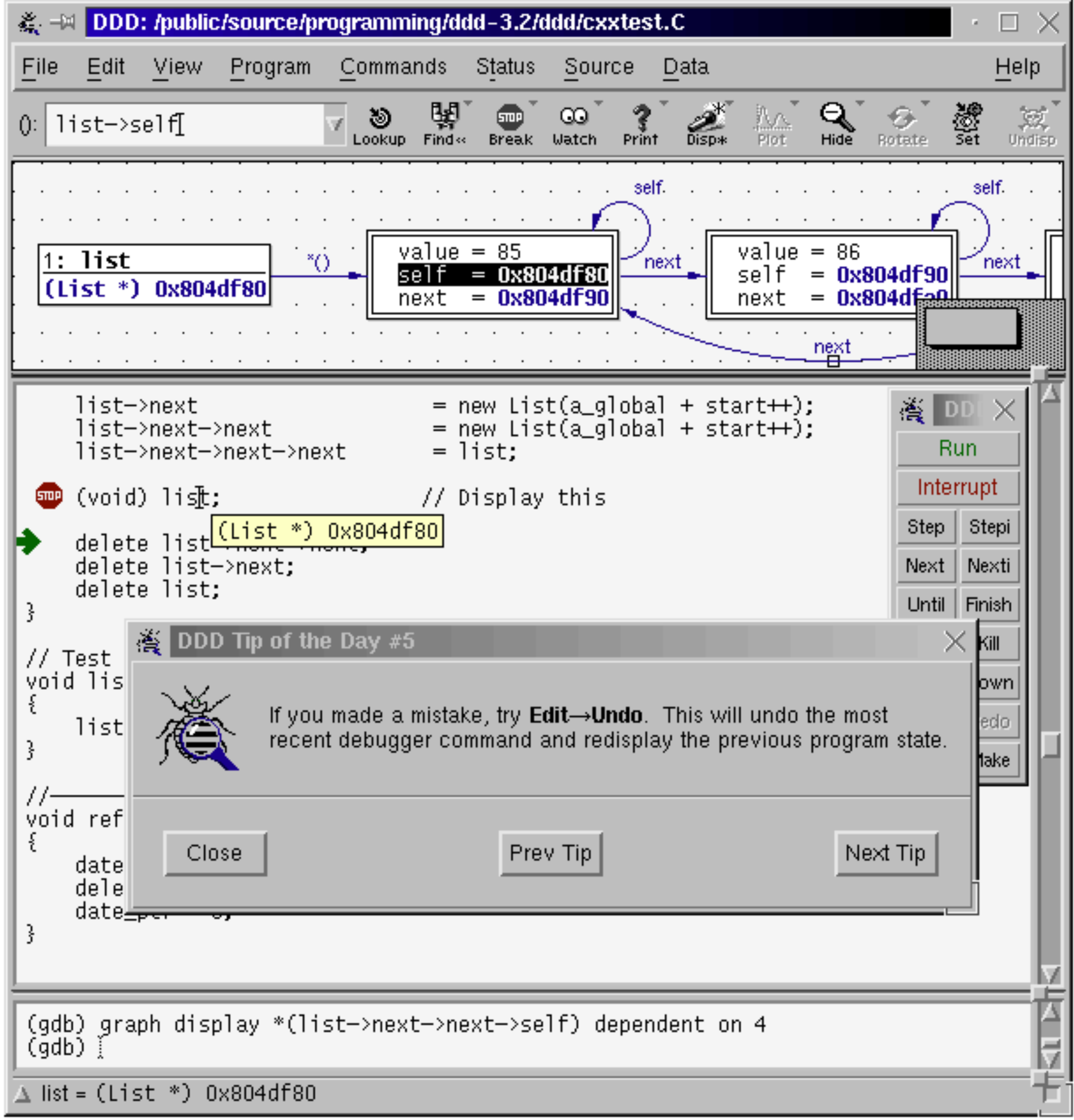


Figure 2. Tango animation of a producer-consumer ring buffer.

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

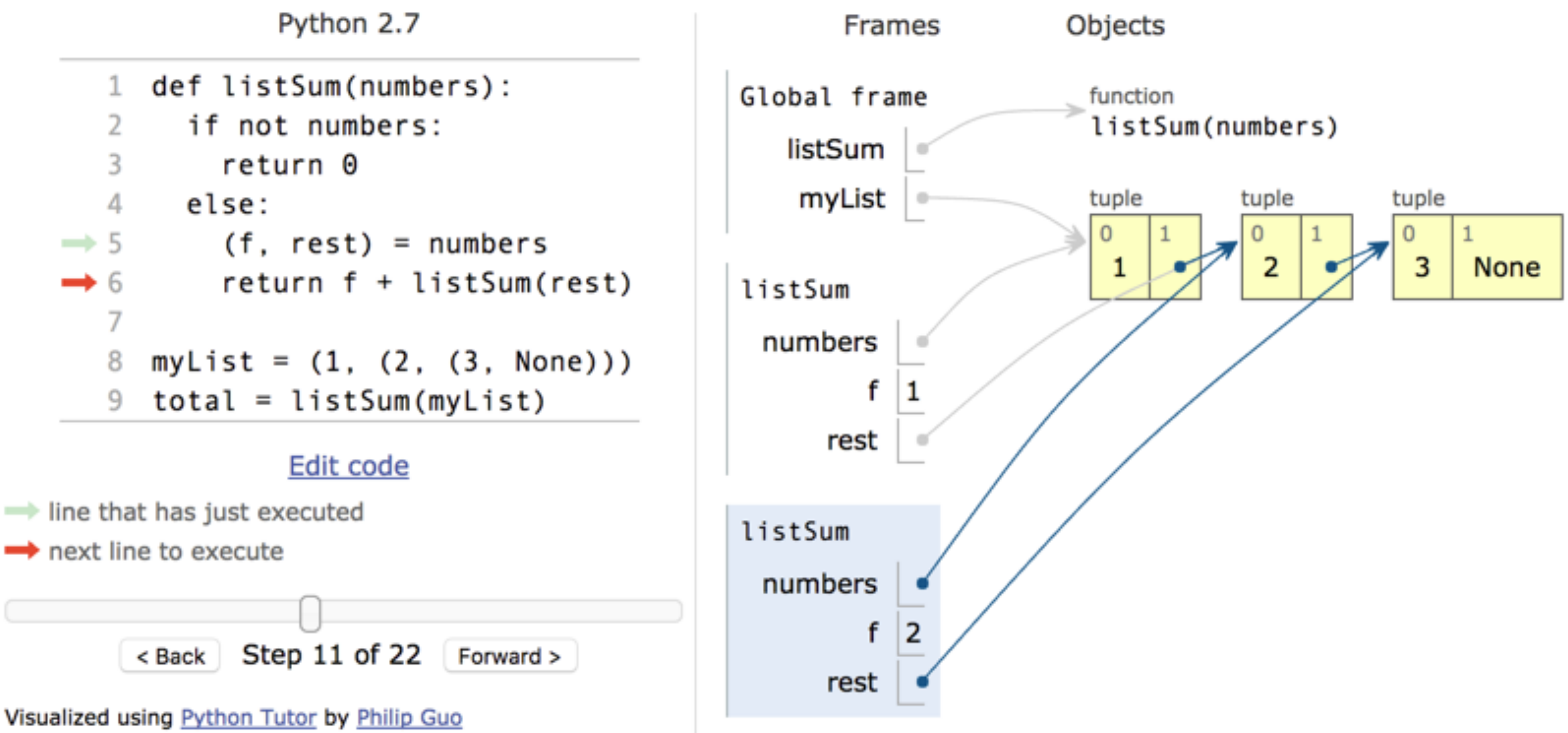


https://www.gnu.org/software/ddd/

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/



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

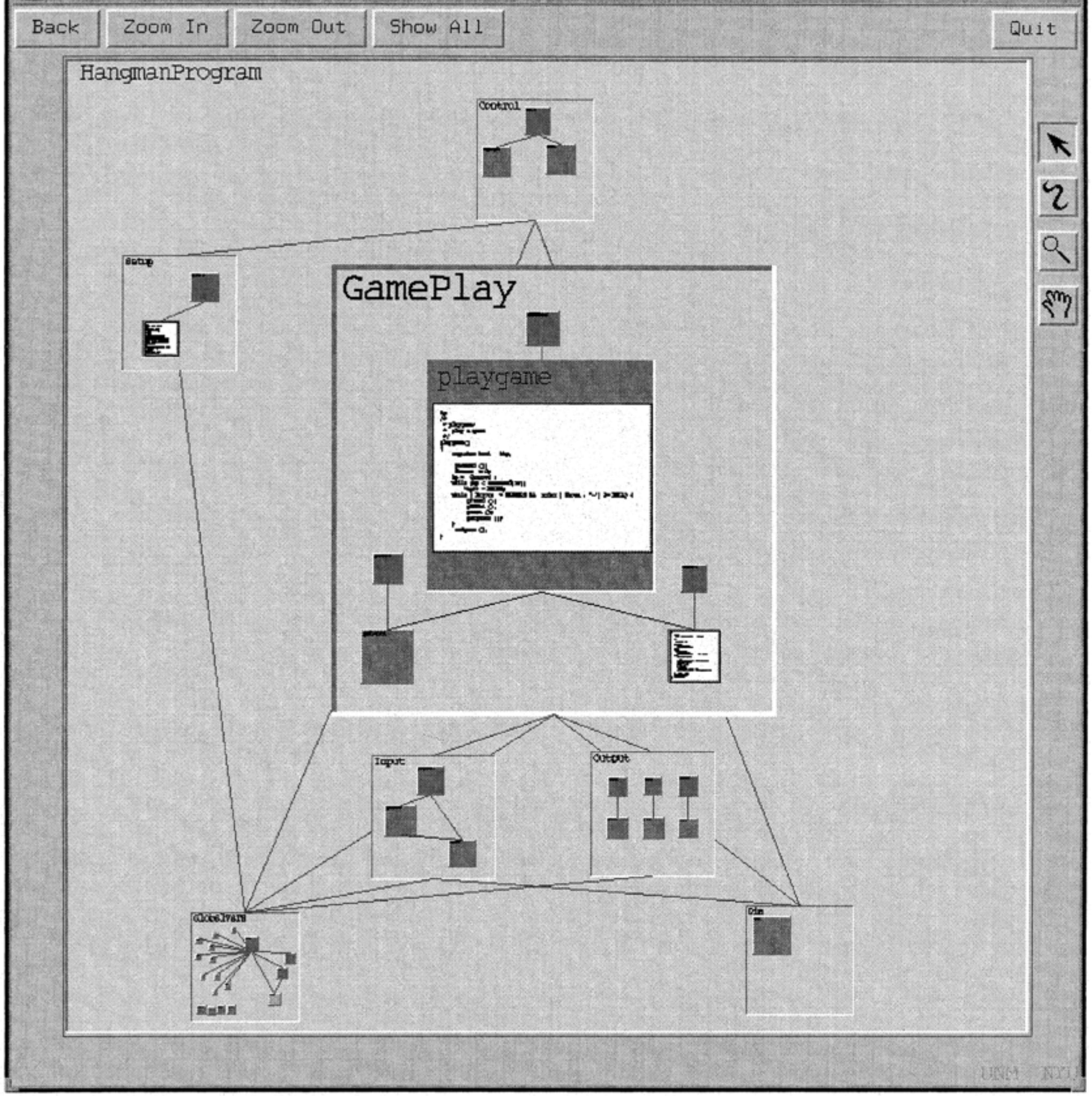
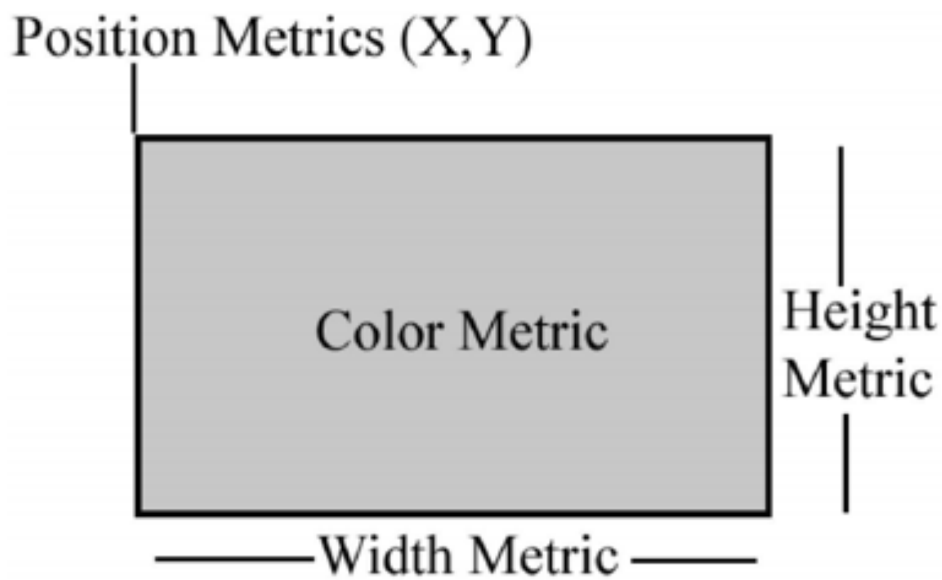
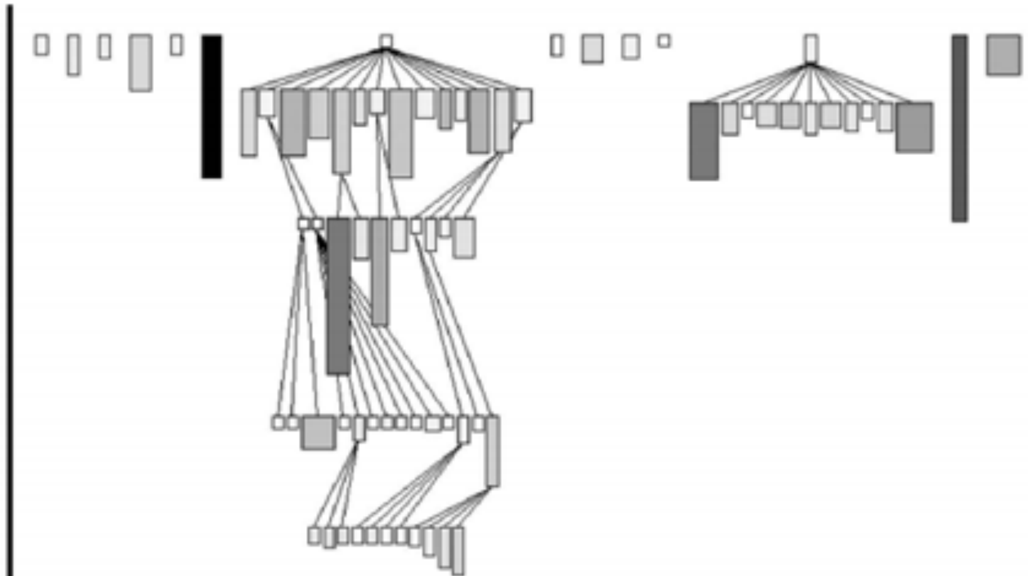


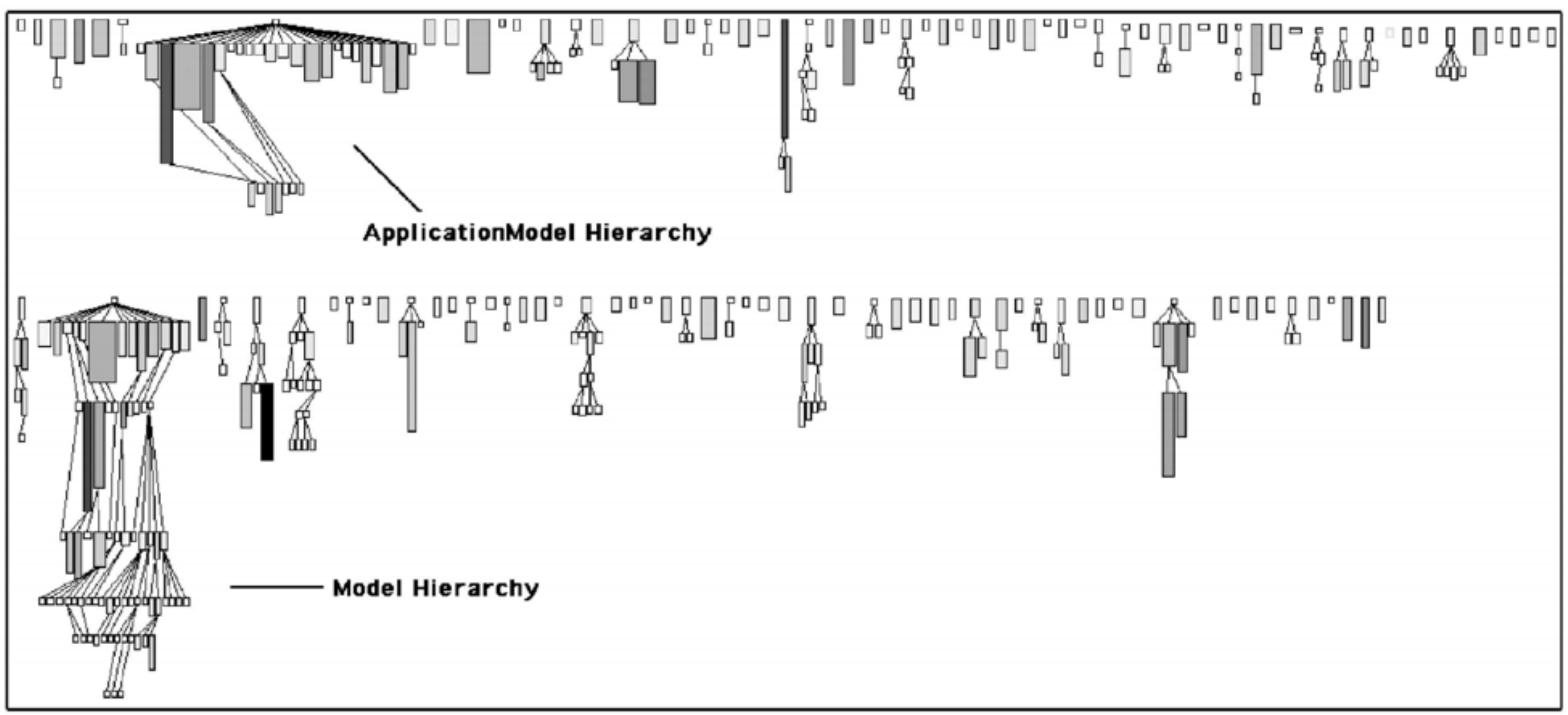
Fig. 3. A SHriMP View of a program which implements a Hangman game. The main subsystems (Control, Setup, GamePlay, Input, Output, GlobalVars and Die) are shown in this view. A fisheye view of the GamePlay subsystem provides more detail since it is shown larger than the other subsystems. The maintainer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

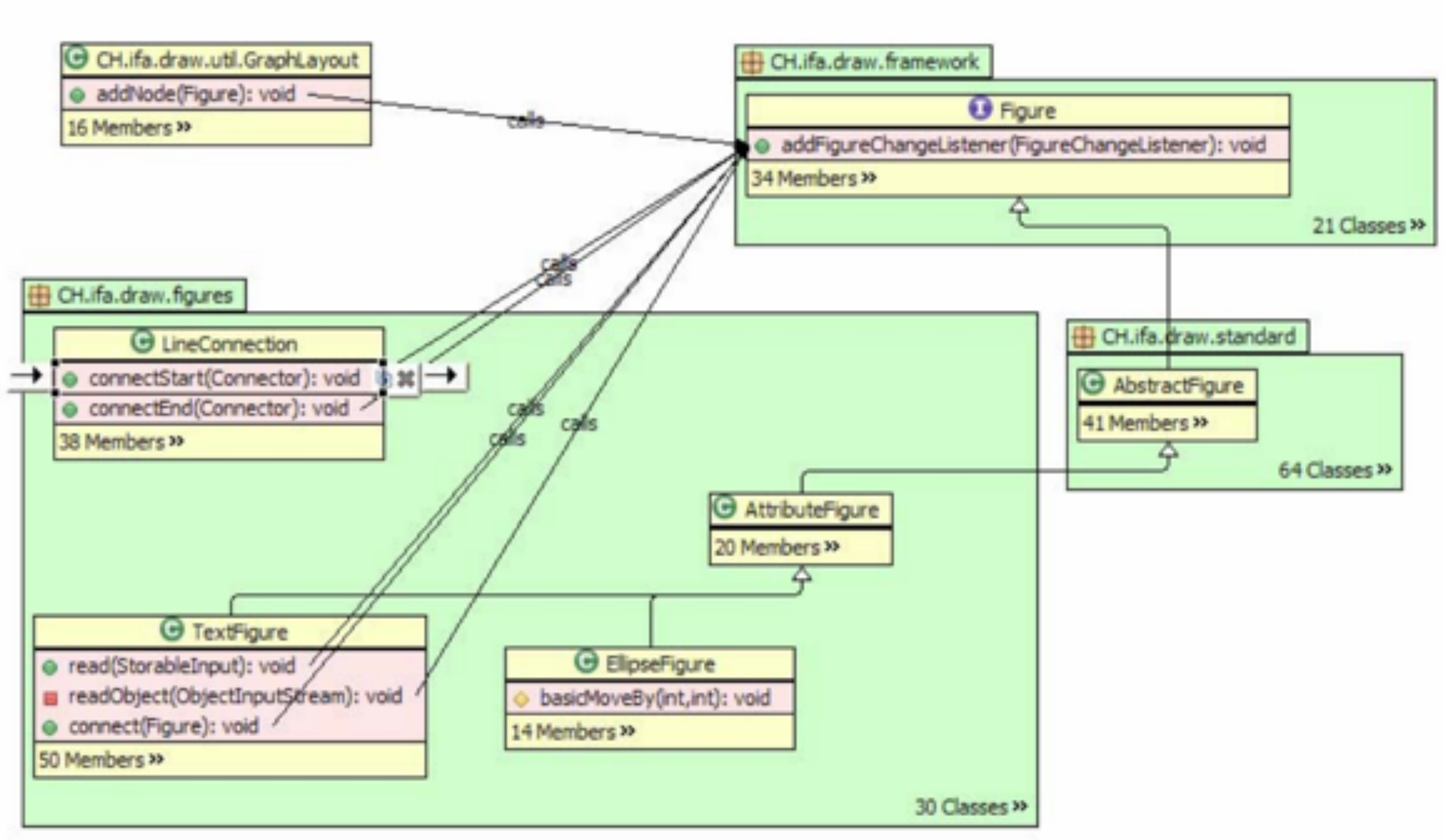






Michele Lanza and Stéphane Ducasse. 2003. Polymetric Views-A Lightweight Visual Approach to Reverse Engineering. IEEE Trans. Softw. Eng. 29, 9 (September 2003), 782-795.

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)

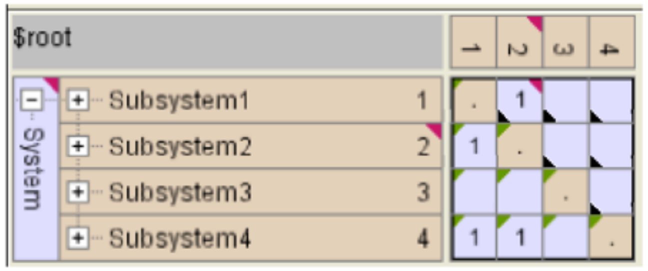


Figure 12: DSM with Rule View

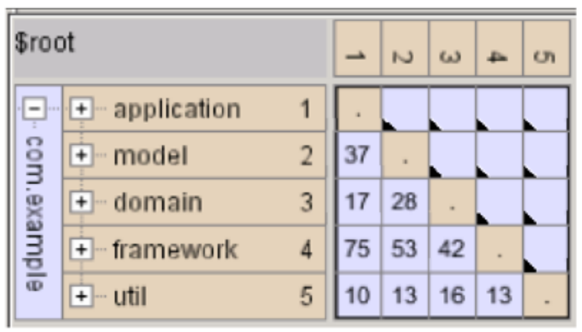


Figure 13: Design Rules for a Layered System

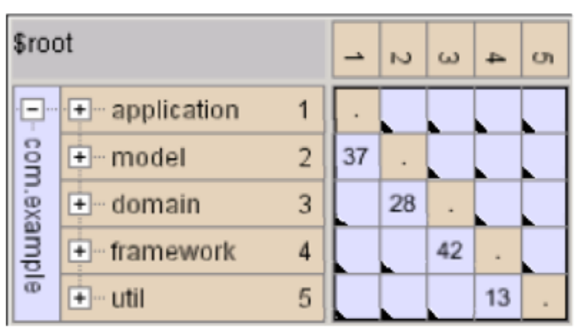


Figure 14: Design Rules for a Strictly Layered System

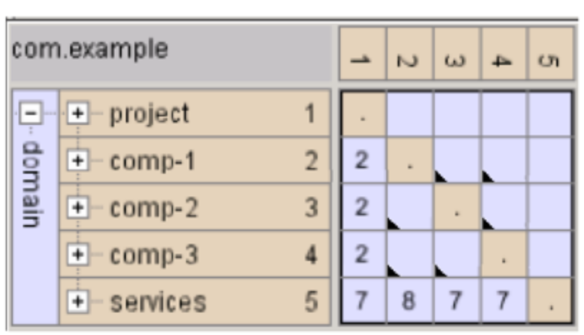
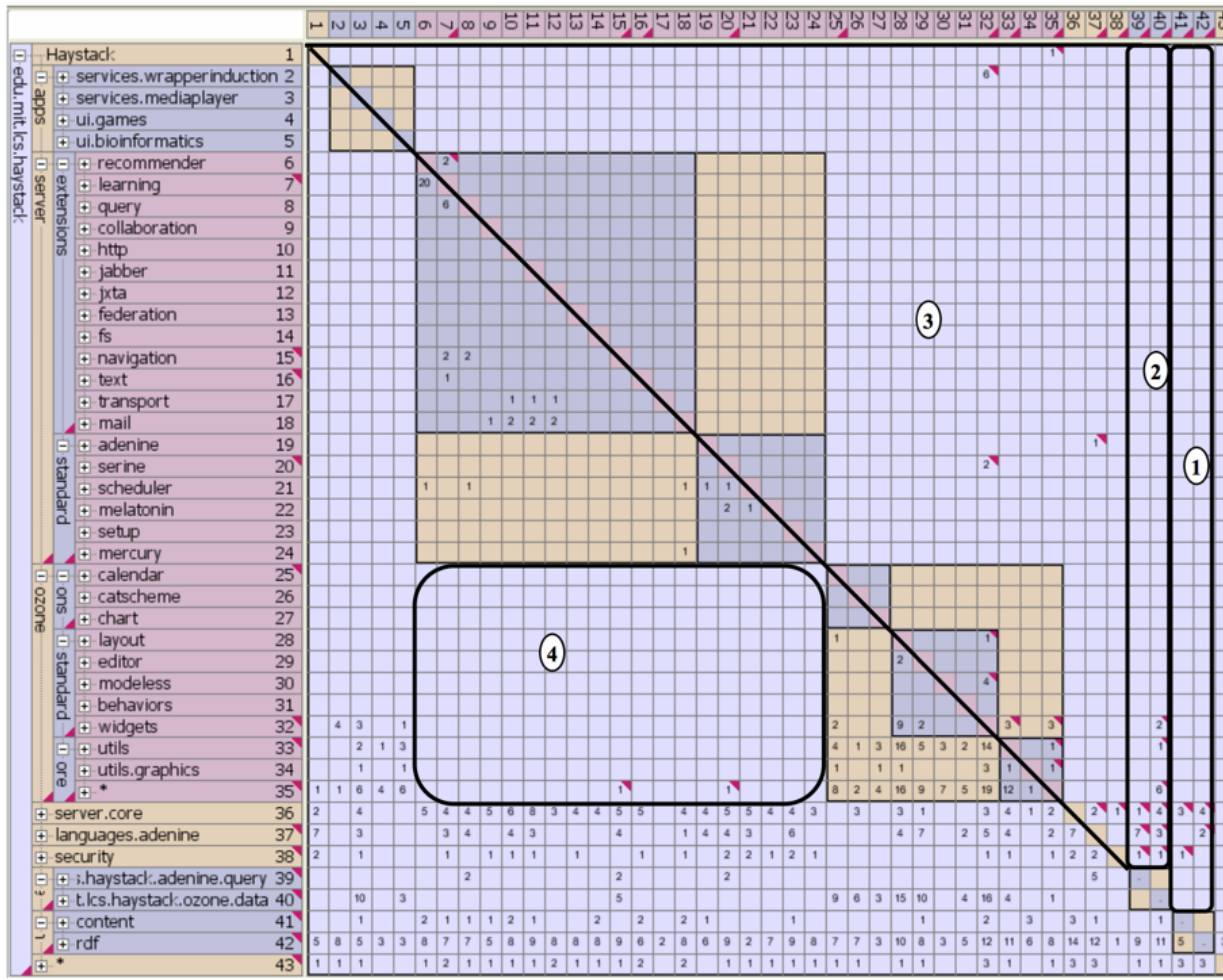


Figure 15: Design Rules for Independent Components

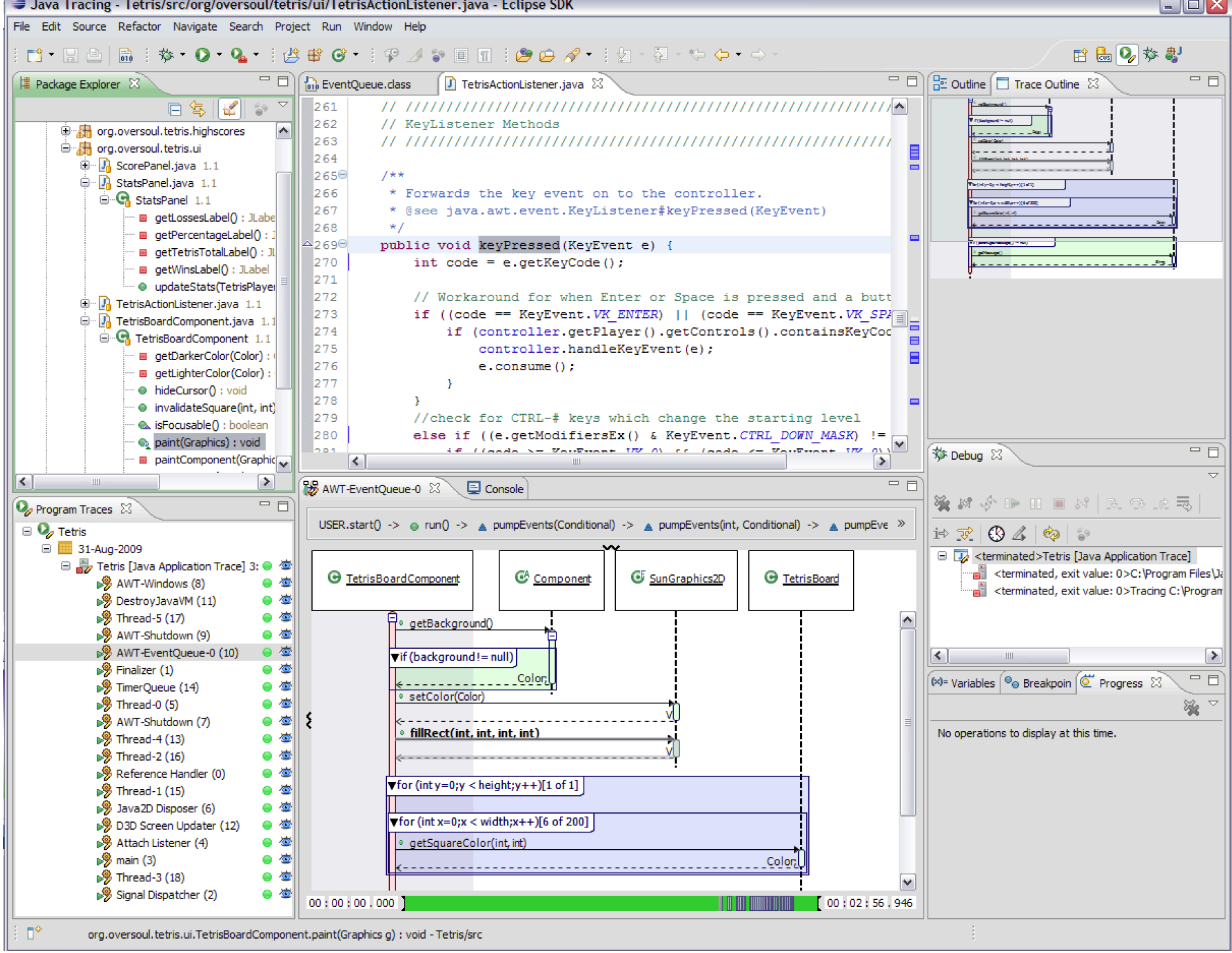


Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. 2005. Using dependency models to manage complex software architecture. Conference on Object-oriented programming, systems, languages, and applications (OOPSLA ’05), 167-176.

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, asycnonicity, …

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

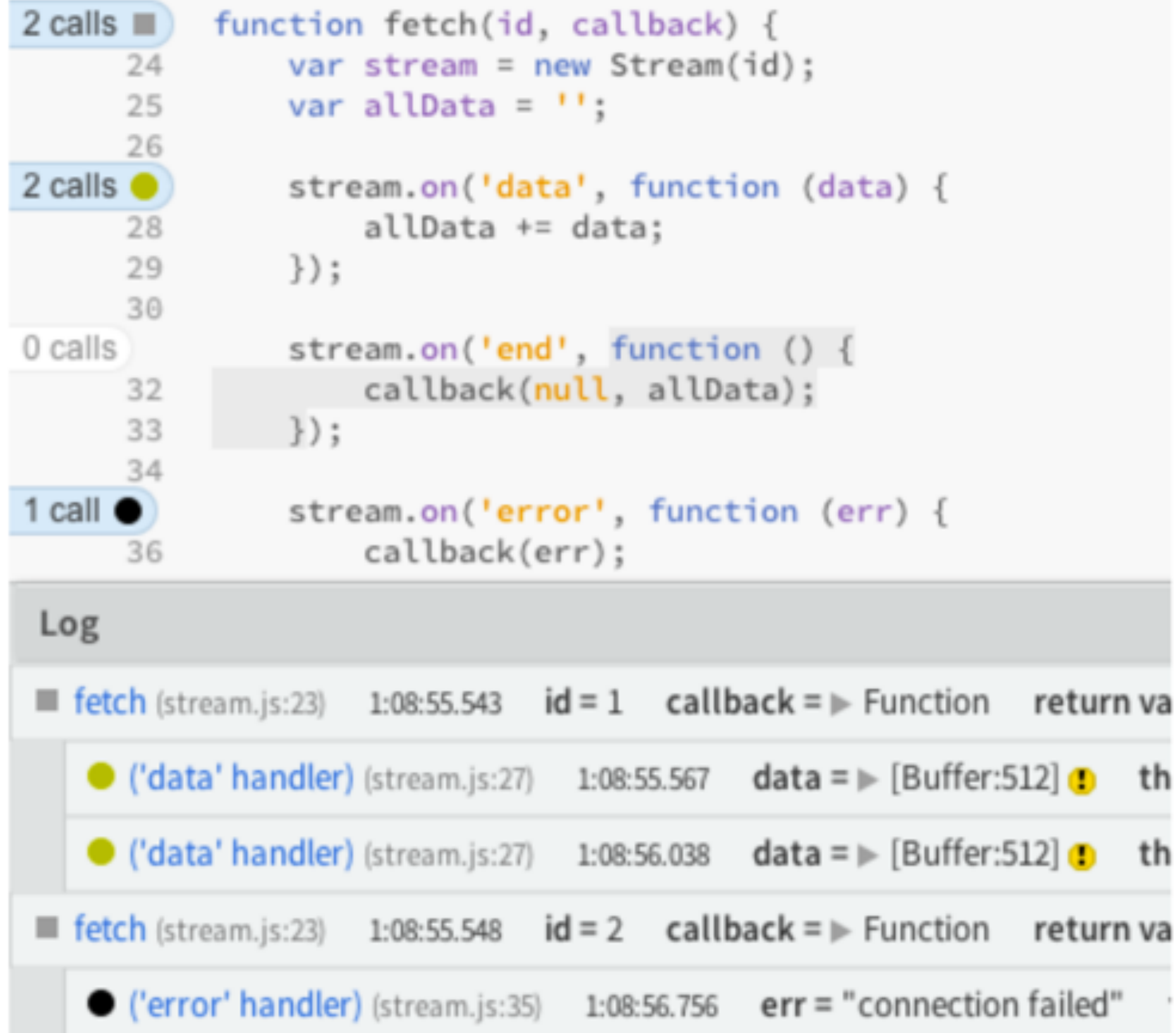


Figure 1. Theseus shows call counts for every function, and an asynchronous call tree allows the user to see how functions interact. In the log below the code, users can see which call to `fetch` corresponds to the failure without adding any debugging-specific code or re-executing their program.

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.

Visual Programming Languages

Definitions

"Programming"

"The process of transforming a mental plan of desired actions for a computer into a representation that can be understood by the computer"

– Jean-Michel Hoc and Anh Nguyen-Xuan

"Single-dimensional characteristics"

The compilers or interpreters programs as long, one-dimensional streams.

Definitions

"Visual Programming"

"Programming in which more than one dimension is used to convey semantics." *- Myers, 1990*

"Token"

"A collection of one or more multi-dimensional objects".

Examples:

Multi-dimensional graphical objects

Spatial relationships

Use of the time dimension to specify "before-after" semantic relationships.

"Visual Expression"

"A collection of one or more tokens"

Definitions

"Visual Programming Language"

"Any system where the user writes a program using two or more dimensions" [Myers, 1990]

"A visual language manipulates visual information or supports visual interaction, or allows programming with visual expressions" [Golin , 1990]

"A programming language that lets users create programs by manipulating program elements graphically rather than by specifying them textually".

"A set of spatial arrangements of text-graphic symbols with a semantic interpretation that is used in carrying out communication actions in the world". [Lakin, 1989]

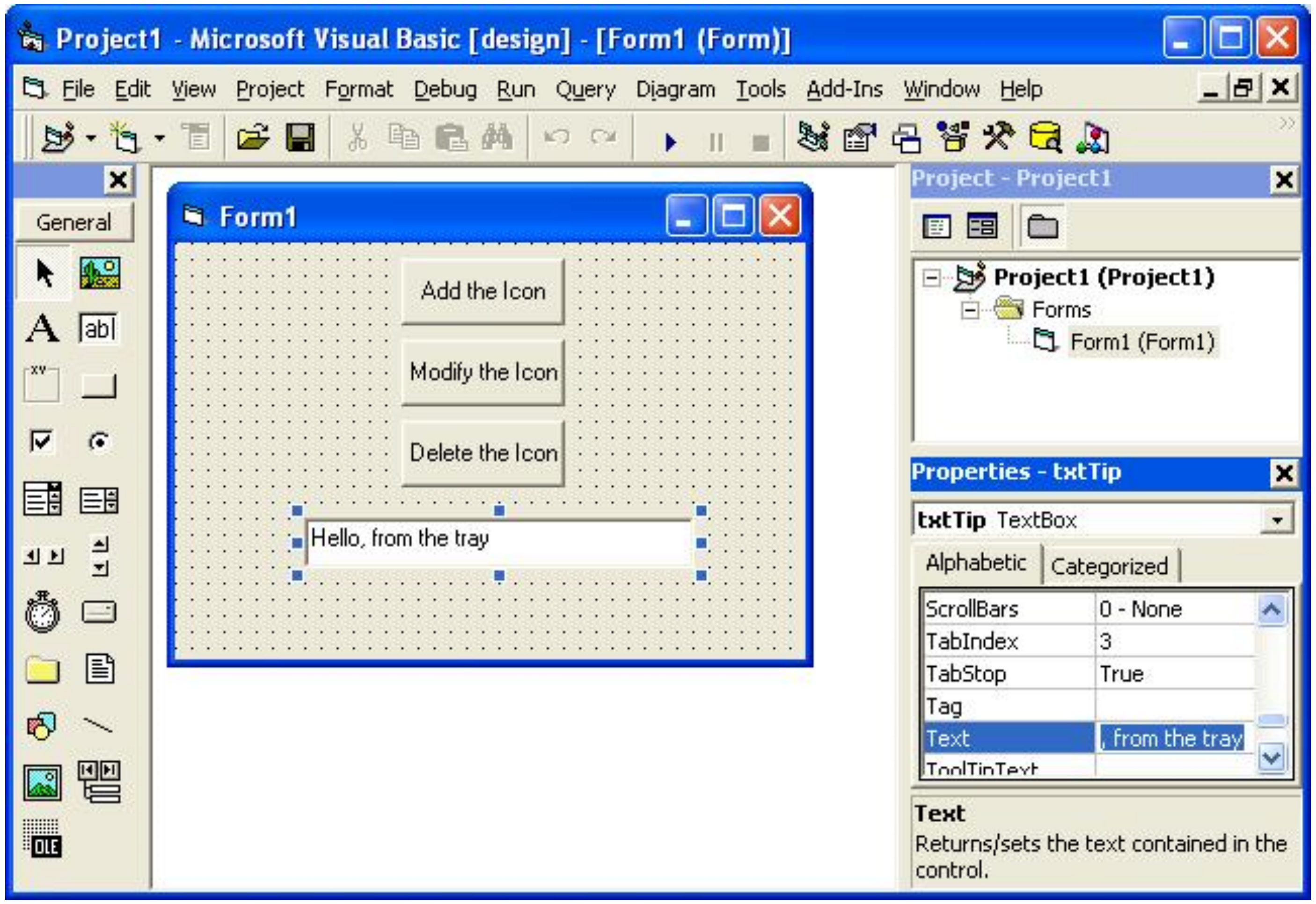
What is not a Visual Programming Language?

Programming Languages like Visual Basic, Visual C++, Visual C sharp, Delphi, etc do not satisfy the *multi-dimensional characterization.*

They are primarily Textual languages with:

A graphical GUI builder

A visual user interface



Goal of VPL Research

- To strive for improvements in programming language design.
- To make programming more accessible to some particular audience.
- To improve correctness with which people perform programming tasks.
- To improve the speed with which people perform programming tasks.

Motivation from Psychology

Language determines thought and that linguistic categories limit and determine cognitive categories [1]

In longer sentences meaning of each word may be clear, but the way in which they are strung together makes little sense imposes a tremendous mental workload to understand. [2]

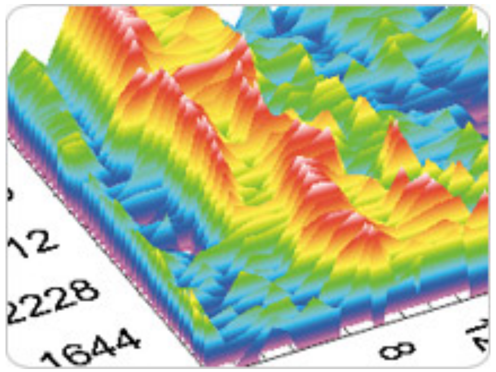
Most design tasks require 3 cognitive skills: search, recognition and inference.
Diverse set of views (and studies) exist today about whether VPLs aid in search or cognition. [3]

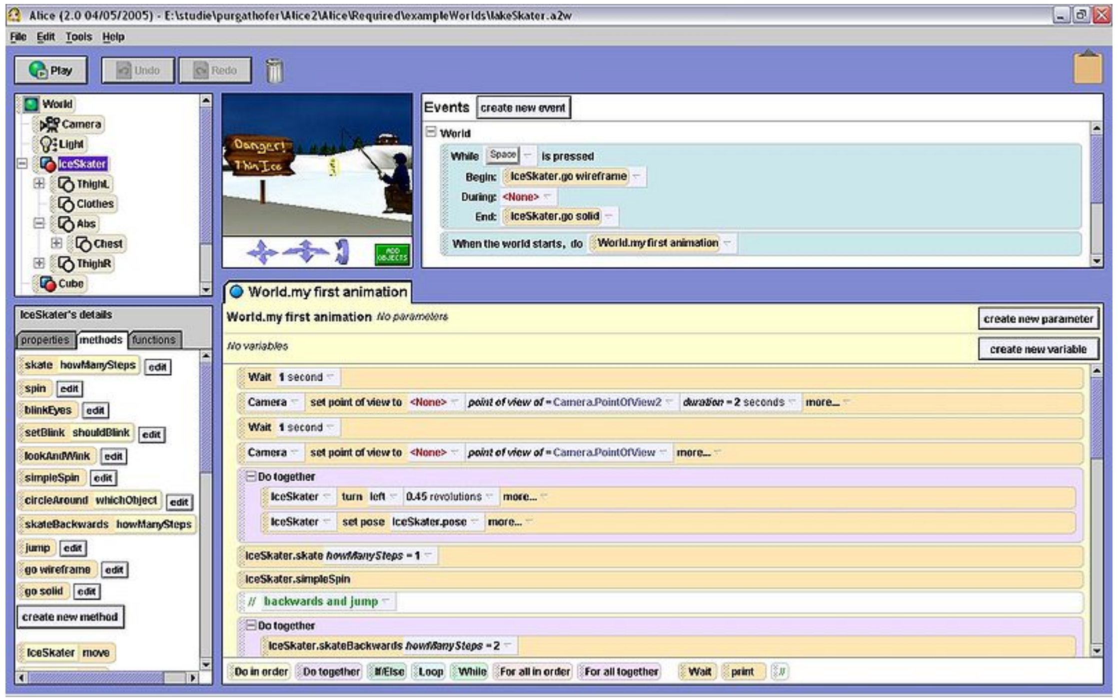
[1] Sapir, E. (1929): 'The Status of Linguistics as a Science'. In E. Sapir (1958): *Culture, Language and Personality* (ed. D. G. Mandelbaum). Berkeley, CA: University of California Press
[2] Christopher D. Wickens, "Engineering Psychology and Human Performance", 3rd Edition
[3] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. Cognitive Science, 11:65-99, 1987.

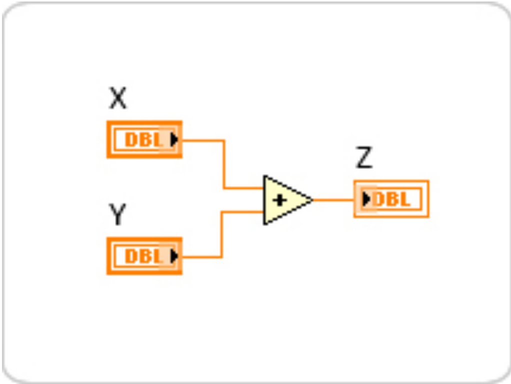
Motivation

Some applications are (believed to be) very well suited to graphical development approaches

- Scientific visualization
- Simulations
- User Interfaces
- Signal Processing
- Data Displays







(Claimed) Advantages of VPLs

- Fewer programming concepts
- Concreteness
- Explicit depiction of relationships
- Immediate visual feedback
- Parallel computation is a natural consequence of many visual programming paradigms

(Claimed) Disadvantages of VPLs

"Deutsch Limit" *

The problem with visual programming is that you can't have more than 50 visual primitives on the screen at the same time.

Some situations in which text has superiority:

Documentation,

Naming to distinguish between elements that are of the same kind, and

Expressing well-known and compact concepts that are inherently textual, e.g. algebraic formulas.

Visual Programming Languages Techniques

- Concreteness: expressing some aspect of a program using instances
  - e.g., display the effects of computation on individual instance
- Directness: small distance between goal and actions required of the user to achieve goal
  - e.g., direct manipulation of object properties
- Explicitness: don't require inference to understand semantics
  - e.g., depict dataflow edges between variables
- Livenesss: offer automatic display of effects of program edits on output
  - e.g., after every edit, IDE reruns code and regenerates output

Levels of liveness

- Level 1: No semantic feedback offered
    - e.g., using ER diagram for documentation
- Level 2: Semantic feedback, but not offered automatically
    - e.g., interpreters
- Level 3: Incremental semantic feedback automatically provided after edit, regenerating onscreen output
    - e.g., spreadsheets
- Level 4: Incremental semantic feedback offered after edits & systems events (e.g., clock ticks, mouse clicks)
    - e.g., some Smalltalk environments (?)

Tanimoto, S., VIVA: a visual language for image processing. Journal of Visual Languages Computing 2(2): 127-139, June 1990.

History of VPLs

1960

- AMBIT/G/L
- Grail
- GAL
- Graphical Program Editor
- Query by Example
- Pygmalion
- I/O Pairs

Techniques

- Graphs
- Flowcharts
- Flowchart derivatives
- FORMS
- Demonstrational

1980

- Action Graphics
- FORMAL
- ThingLab
- Hi-Visual
- LabView
- PROGRAPH
- PIGS
- Pict
- Rehearsal
- SmallStar
- Forms
- Editing by Example
- PICT
- Lotus 1-2-3
- SIL-ICON
- VisiCalc
- HiGraphs
- Miro
- StateMaster

Techniques

- Graphs
- Flowcharts
- Flowchart derivatives
- FORMS
- Demonstrational
- Data Flows
- Spreadsheets
- Matrices
- Jigsaw Puzzles
- Petri nets
- Flowchart derivatives

1990

- Cube
- Cantata
- SchemePaint
- CODE 2.0
- Iconicode
- MViews
- AVS
- Mondrian
- ChemTrains
- Vampire
- VIPR
- SPE

Techniques/Goals

- 3D Rendering
- Visual Hierarchy
- Procedures
- Control Structures
- Programmable Graphics
- Animations
- Video Imagery Exploitation
- General purpose, declarative language
- Audio, video and image processing
- Graphical models from behavioral models
- Learning and Cognitive abilities in vision processes
- Handling Scalability, typing, and imperative design
- Collaborative Software Development

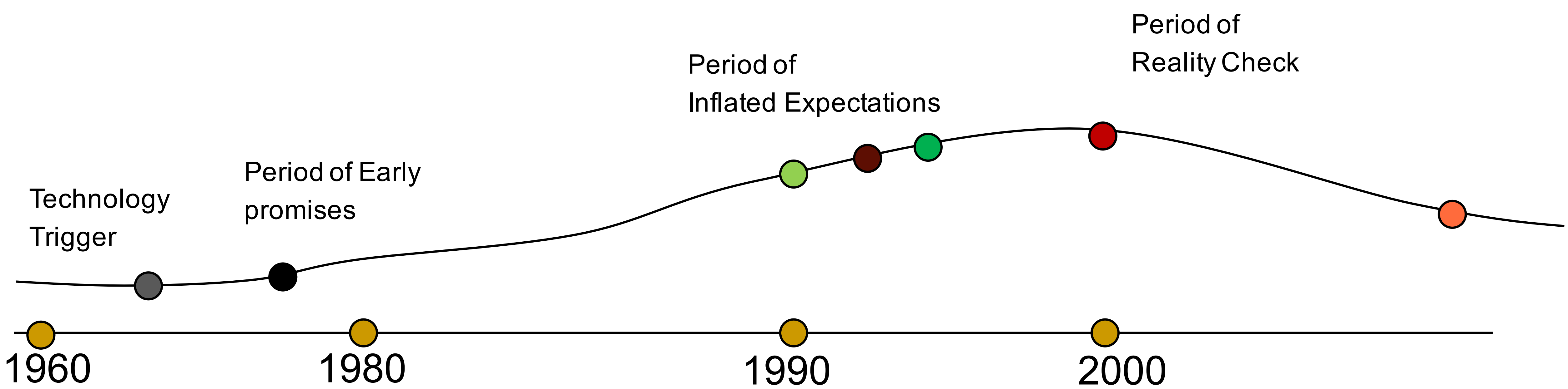
2000

- LOFI/HIPI
- FOXQ
- VMQL
- GXL
- Euler View
- Yahoo Pipes
- Popfly

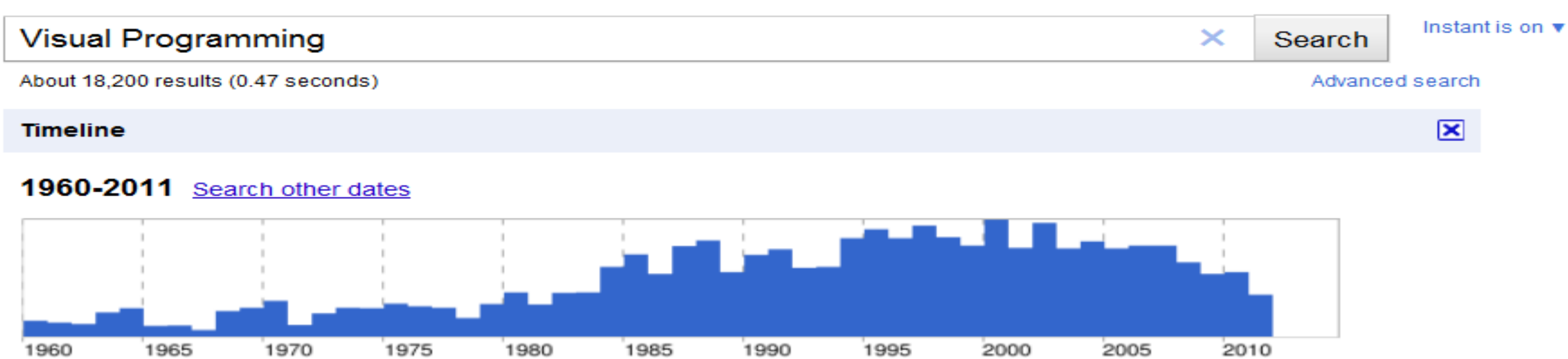
Techniques/Goals

- Child Learning
- Xquery by FORMS
- Spreadsheet Analysis
- Visual Model Query
- Layouts
- Specification and Interchange
- Mashups
- Web-based design
- Programming for end-users (2003) / non-Professionals

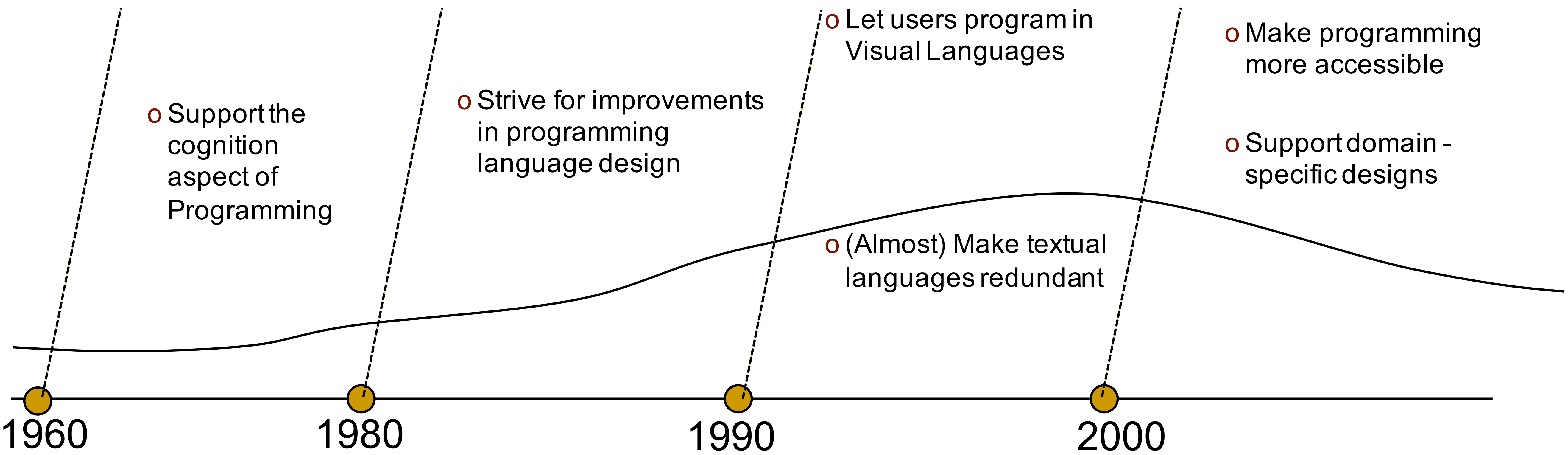
History of VPLs



- [**Ellis**, 1969] : GRAIL
- [**Smith**, 1975] : Pygmalion
- [**Myers, 1990**] : Taxonomies for VPL
- [**Repenning, 1992**] : Agent Sheet
- [**Burnett, 1994**] : Broad Classifications for VPL Research
- [**Kirsten N. Whitley, 1997**]: User Studies (for/against VPLs)
- [**MacLaurin, 2009**] : KODU



History of VPLs



Taxonomy of visual programming languages

Specification Technique:	Systems:
Textual Languages:	Pascal, Ada, Fortran, Lisp, Ada, etc. Tinker, Smallstar
Flowcharts:	Grail, Pict, FPL, IBGE, OPAL
Flowchart derivatives:	GAL, PIGS, SchemaCode, PLAY
Petri nets:	MOPS-2, VERDI
Data flow graphs:	Graphical Program Editor, PROGRAPH, Graphical Thinglab, Music System, HI-VISUAL, LabVIEW, Fabrik, InterCONS
Directed graphs:	AMBIT/G/L, State Transition UIMS, Bauer's Traces
Graph derivatives:	HiGraphs, Miro, StateMaster
Matrices:	ALEX, MPL
Jigsaw puzzle pieces:	Proc-BLOX
Forms:	Query by Example, FORMAL
Iconic Sentences:	SIL-ICON
Spreadsheets*:	VisiCalc, Lotus 1-2-3, Action Graphics, ''Forms''
Demonstrational*:	Pygmalion, Rehearsal World, Peridot
None*:	I/O Pairs, Editing by Example

Brad A. Myers. "Taxonomies of Visual Programming and Program Visualization," Journal of Visual Languages and Computing. vol. 1, no. 1. March, 1990. pp. 97-123.

Dataflow Program Representations

- Represent computation as a network
- Nodes correspond to components
- Edges correspond to data flow between components

Prograph

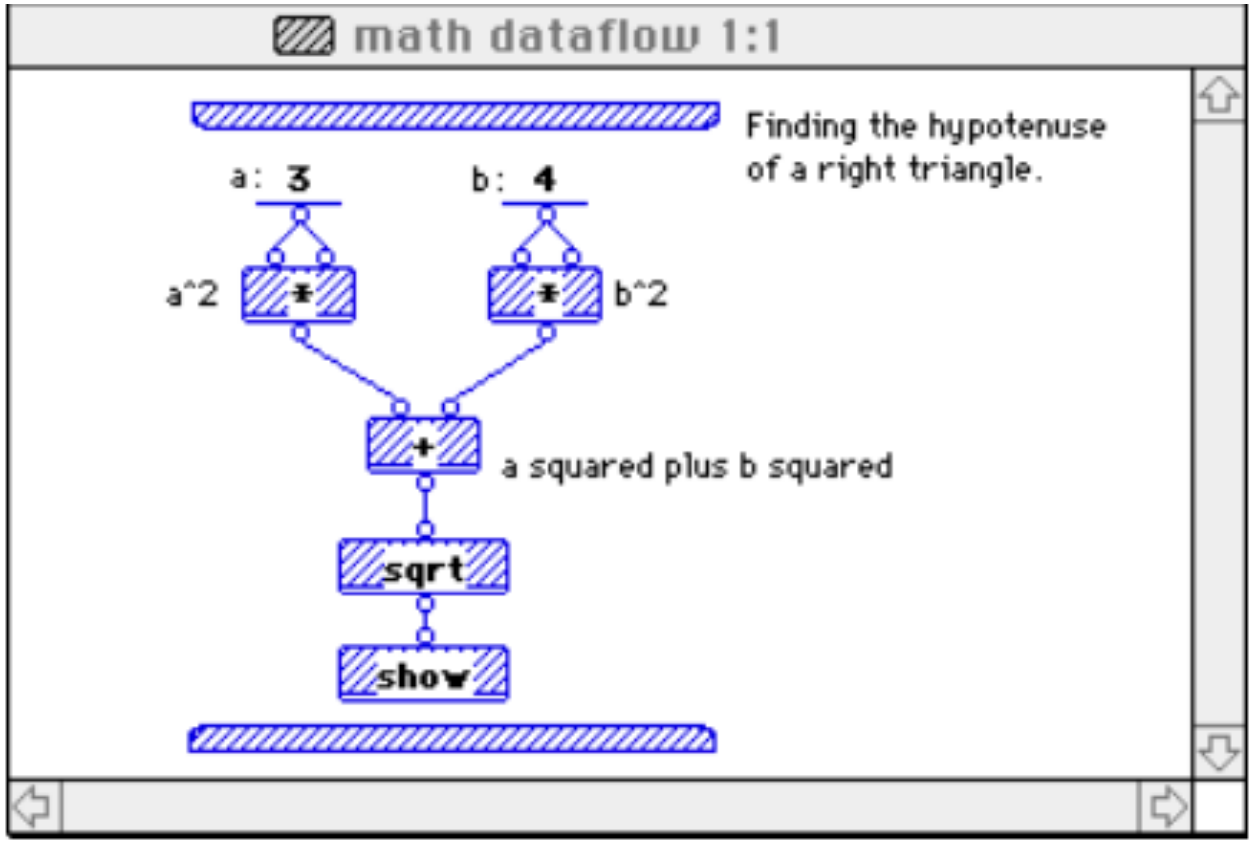
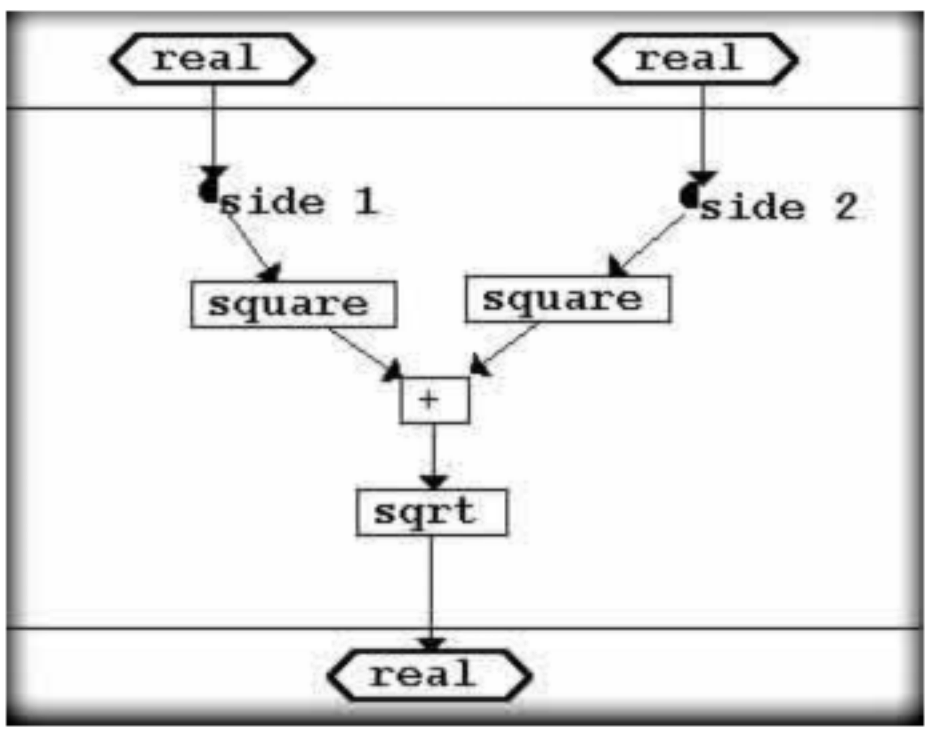


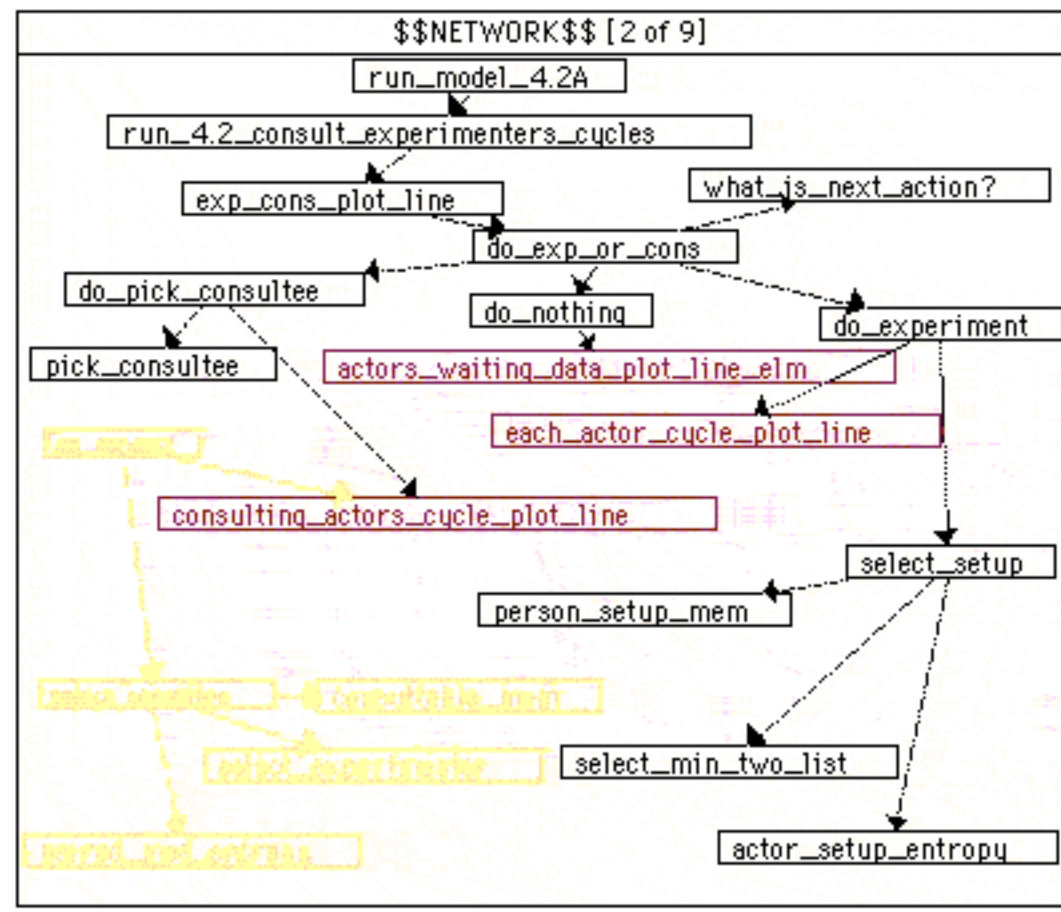
Figure 3: Dataflow programming in Prograph. Here the programmer is using the low-level (primitive) operations to find the hypotenuse of a right triangle. Prograph allows the programmer to name and compose such low-level graphs into higher-level graphs that can then be composed into even higher-level graphs, and so on.

Margaret M. Burnett, "Visual Programming" In the Encyclopedia of Electrical and Electronics Engineering (John G. Webster, ed.), 1999

Industrial Example: Clarity

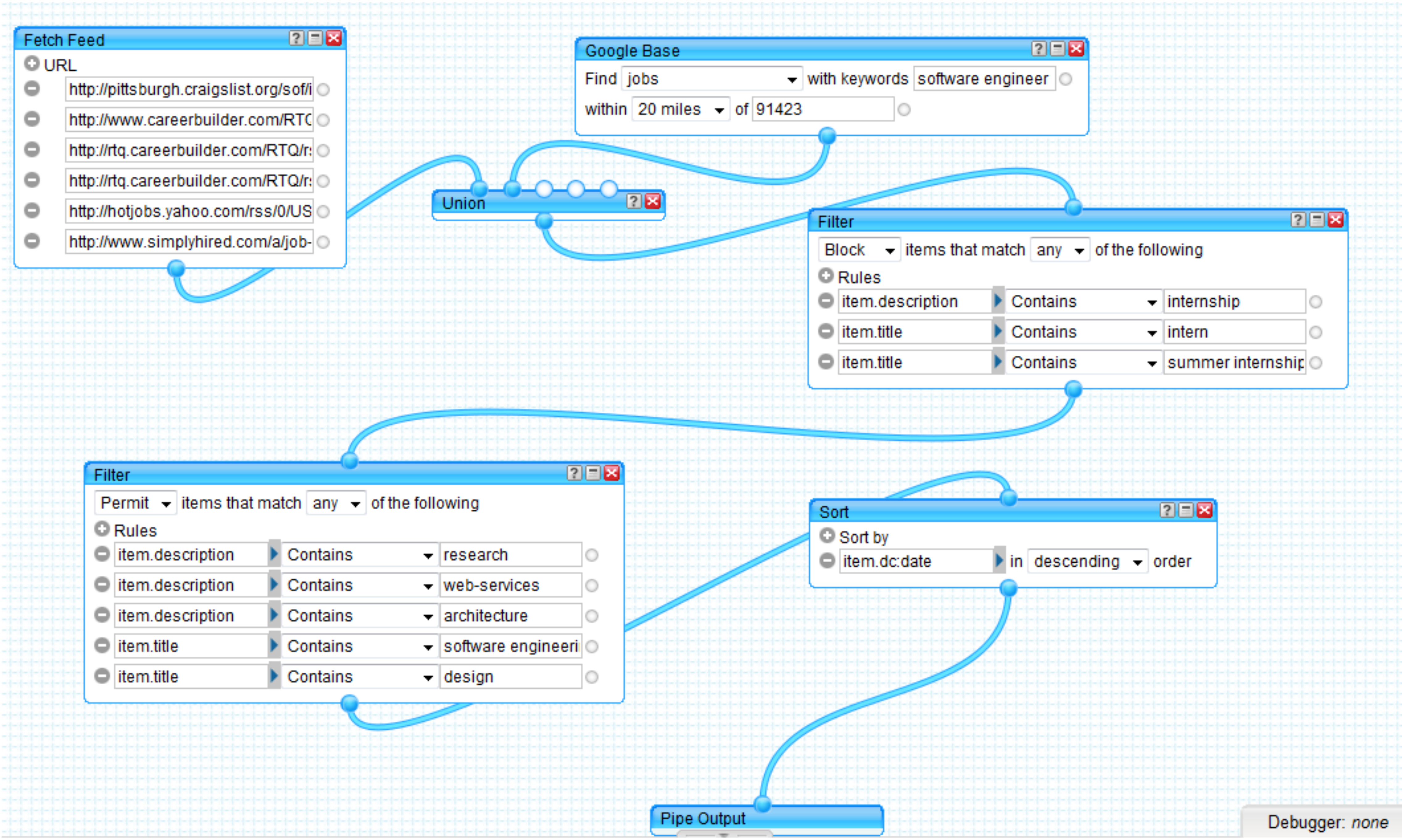
- “Clarity is a schematic functional programming environment that allows you to design and implement programs by drawing them. The picture below shows an example of the hypotenuse function that expresses Pythagoras' theorem.”





http://www.clarity-support.co.uk/products/clarity/

Industrial Example: Yahoo Pipes

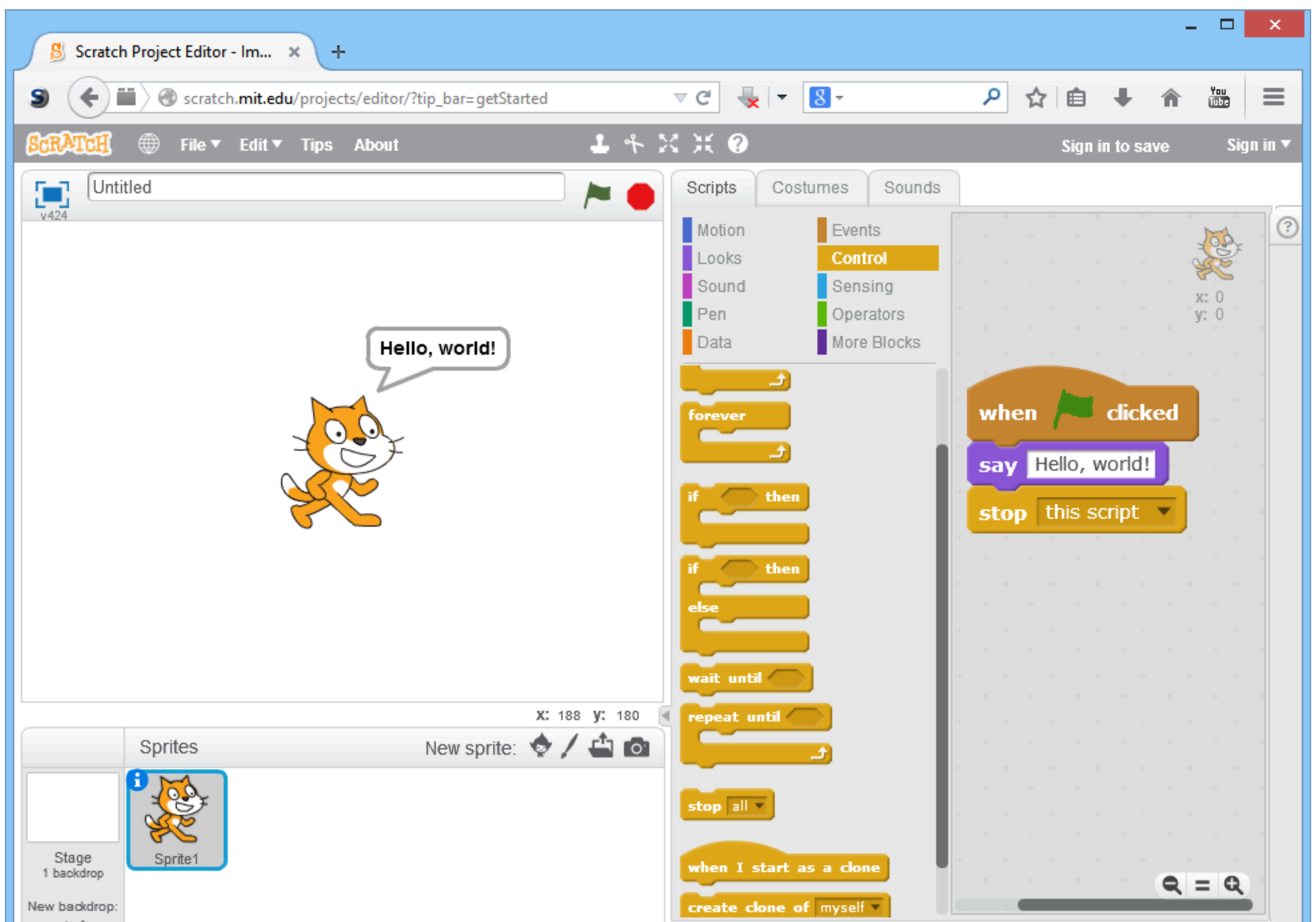


https://en.wikipedia.org/wiki/Yahoo!_Pipes

https://www.youtube.com/watch?v=Xv-4TOit5_g

Structured editors

- Structured editors that utilize extra dimension to capture program semantics can be considered visual programming languages
  - e.g., Alice, Scratch



Form Representations

- Program consists of a form, with a network of interconnected cells
- Developers define cell through combination of pointing, typing, gesturing
- Cells may define constraints describing relationships between cells

Forms/3

- Based on constraints between cells
- Supports graphics, animation, recursion
- Concreteness: resulting box is immediately seen
- Directness: demonstrates elements directly
- Level 4 liveness: immediate visual feedback

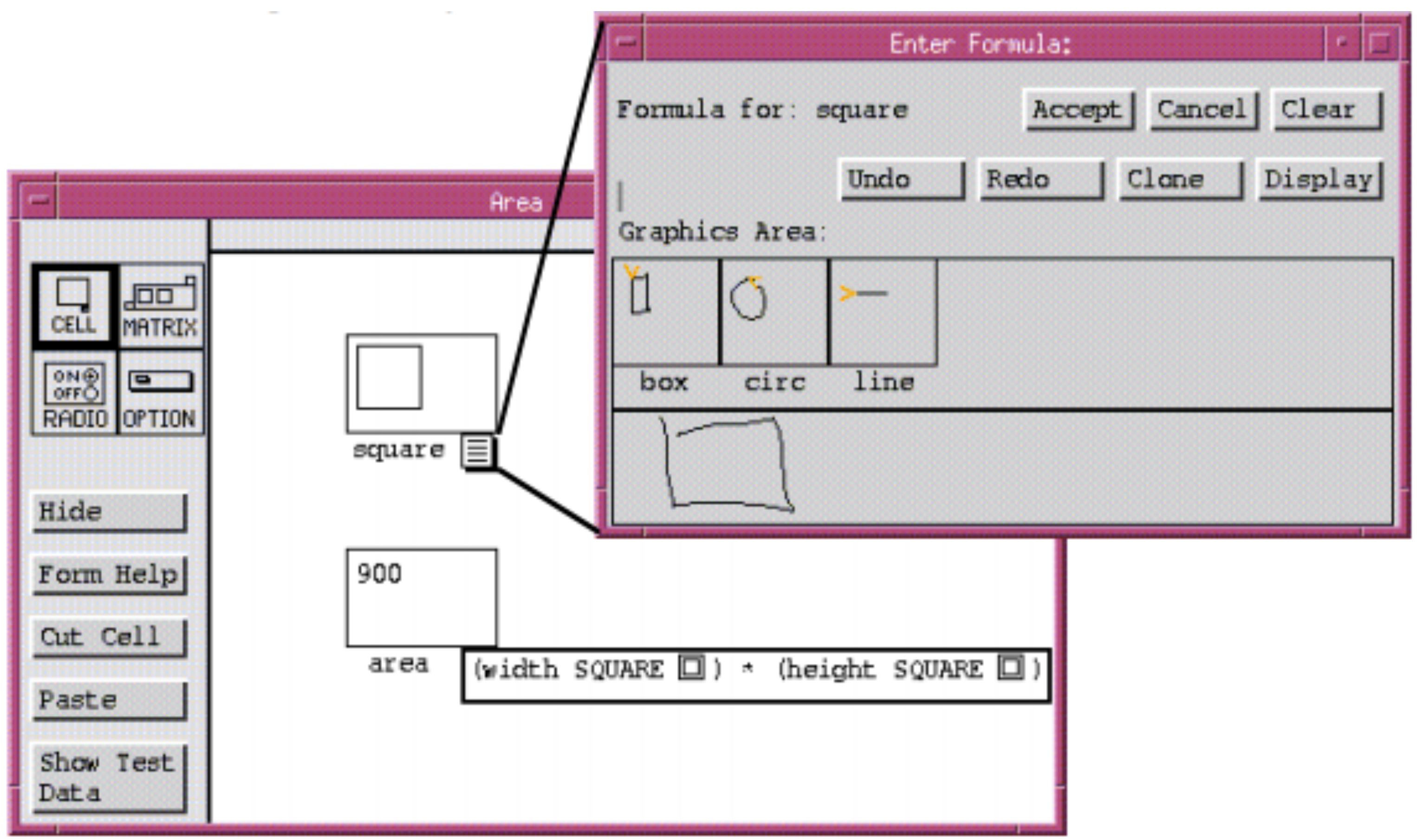
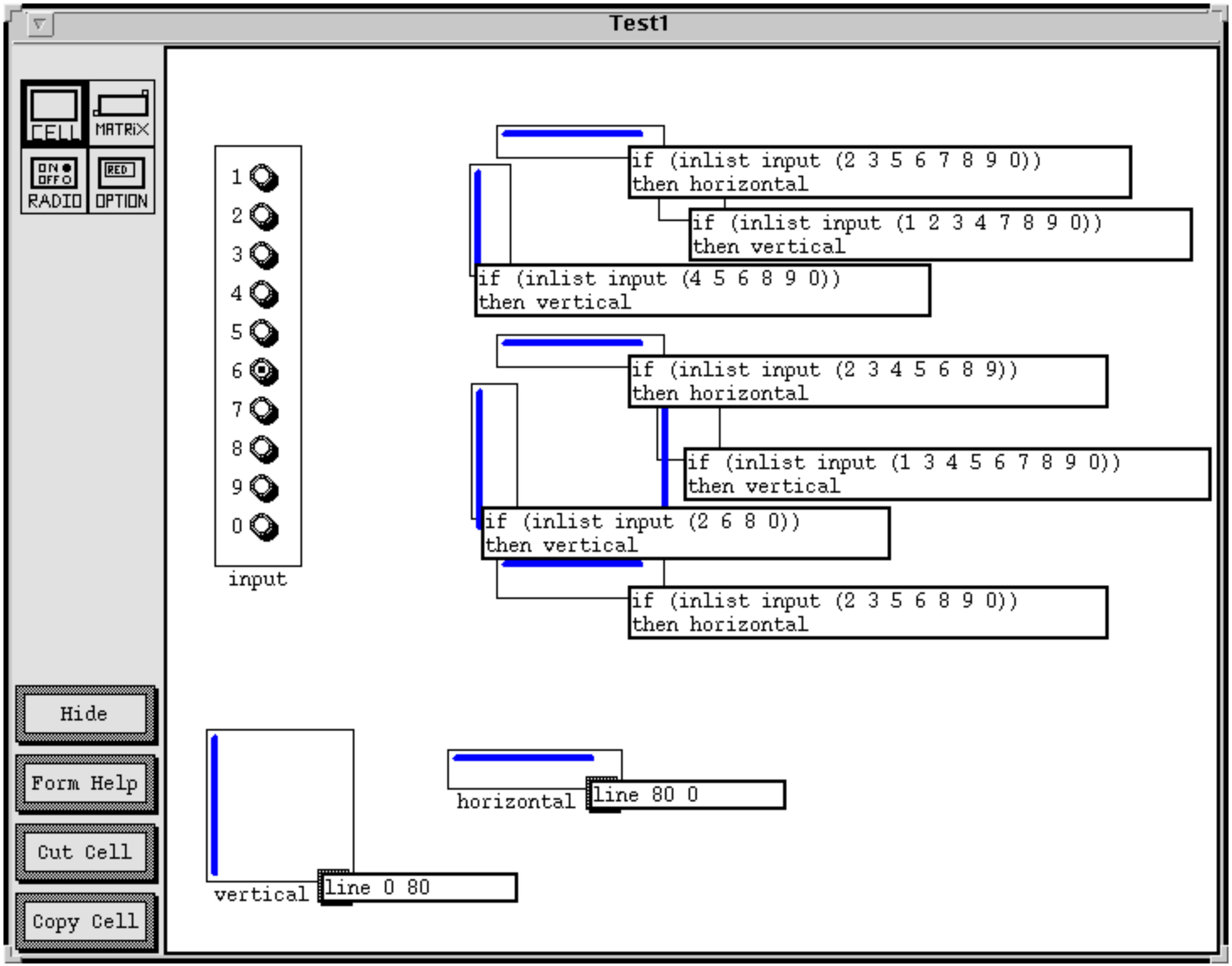


Figure 2: Defining the area of a square using spreadsheet-like cells and formulas in Forms/3. Graphical types are supported as first-class values, and the programmer can enter cell square's formula either by sketching a square box or by typing textual specifications (e.g., "box 30 30").

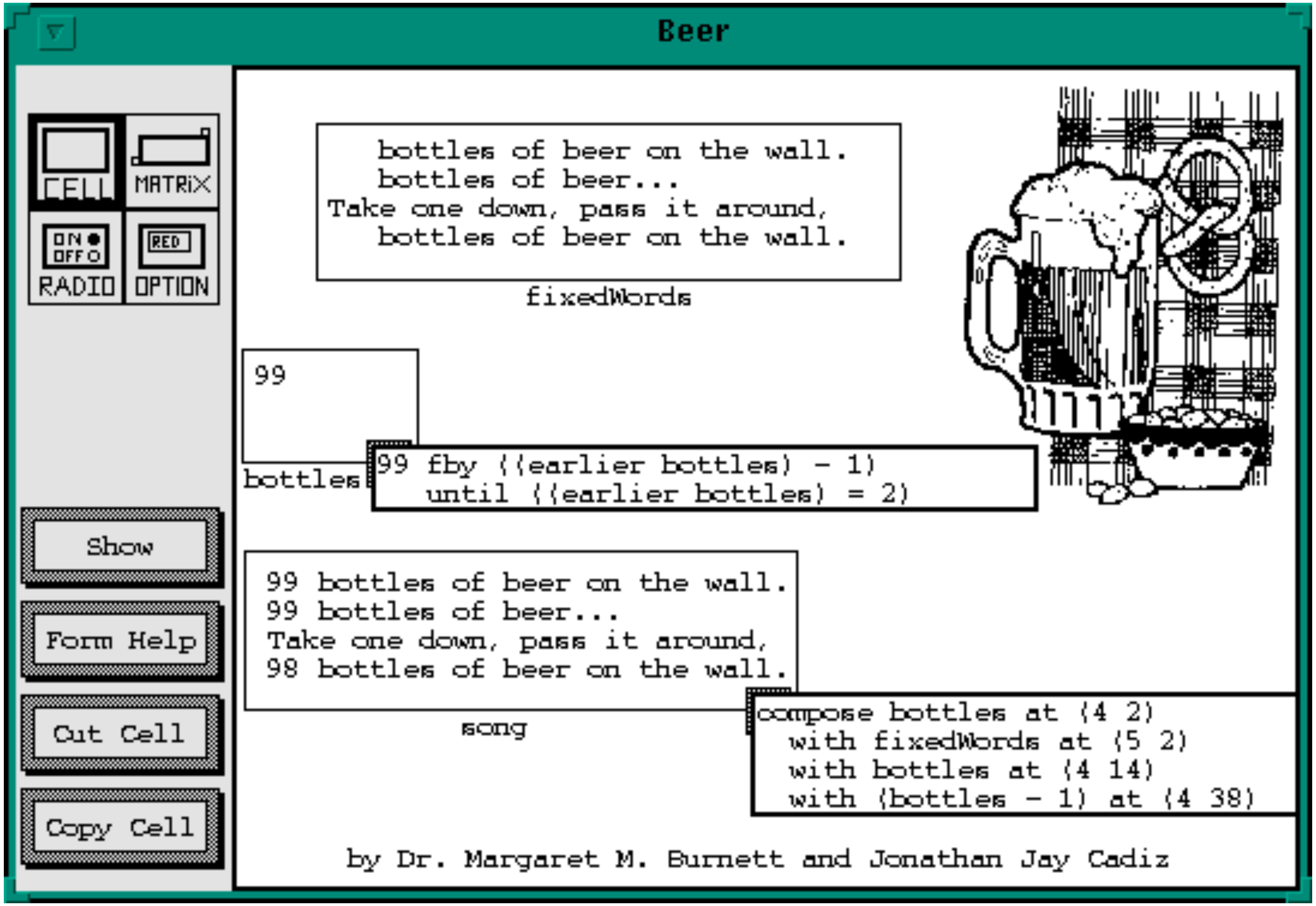
Margaret M. Burnett, "Visual Programming" In the Encyclopedia of Electrical and Electronics Engineering (John G. Webster, ed.), 1999

Forms/3 Example



http://web.engr.oregonstate.edu/~burnett/Forms3/LED.html

Forms/3 Example



Interstate

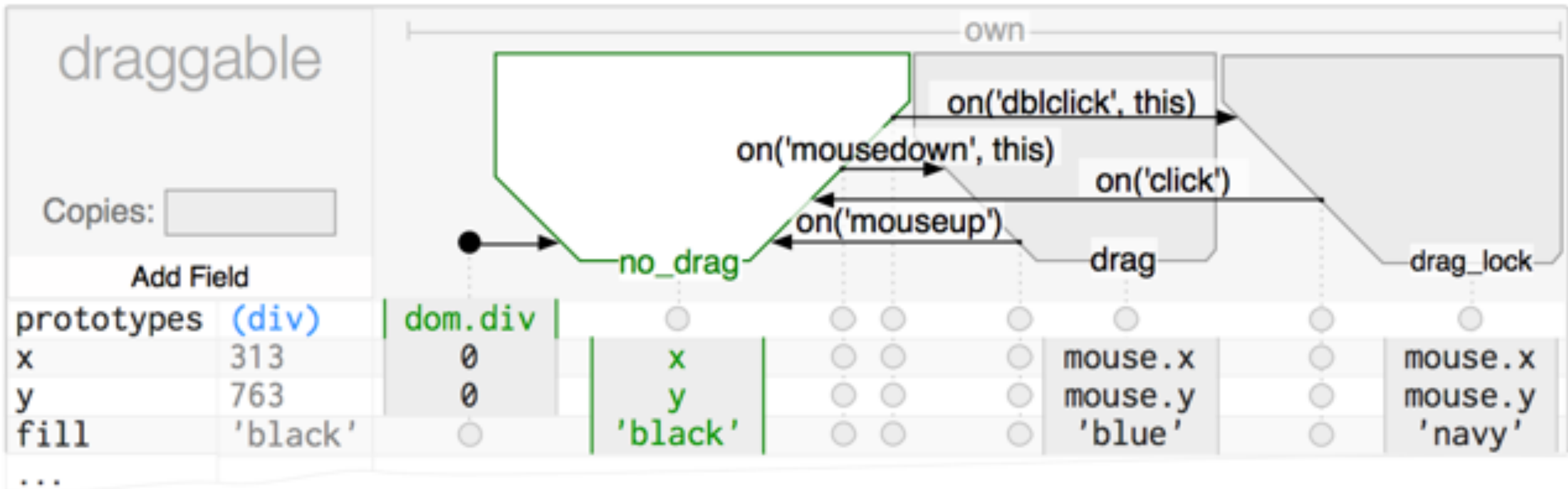


Figure 1: A basic InterState object, named `draggable`, which implements draggable and drag lock behaviors. Properties that control `draggable`'s display are represented as rows (e.g. `x`, `y`, and `fill`). States and transitions are represented as columns (e.g. `no_drag` and `drag`). An entry in a property's row for a particular state specifies a constraint that controls that property's value in that state. Here, while `draggable` is in the `drag` state, `x` and `y` will be constrained to `mouse.x` and `mouse.y` respectively, meaning `draggable` will follow the mouse.

http://interstate.from.so/

https://www.youtube.com/watch?v=M--9jsuDZis

Stephen Oney, Brad Myers, and Joel Brandt. 2014. InterState: a language and environment for expressing interface behavior. *Symposium on User interface software and technology*, 263-272.

Assessing Usability

- Empirical techniques assess usability through studies gathering data
- Analytical techniques use principles & guidelines to estimate the usability of a system

- Will look at a technique for analytical usability evaluation here

Cognitive Dimensions of Notations

- Analytical technique for assessing usability of notation through a set of heuristics
  - Also terminology for describing usability problems

Abstraction gradient	What are the minimum and maximum levels of abstraction? Can fragments be encapsulated?
Closeness of mapping	What 'programming games' need to be learned?
Consistency	When some of the language has been learnt, how much of the rest can be inferred?
Diffuseness	How many symbols or graphic entities are required to express a meaning?
Error-proneness	Does the design of the notation induce 'careless mistakes'?
Hard mental operations	Are there places where the user needs to resort to fingers or penciled annotation to keep track of what's happening?
Hidden dependencies	Is every dependency overtly indicated in both directions? Is the indication perceptual or only symbolic?
Premature commitment	Do programmers have to make decisions before they have the information they need?
Progressive evaluation	Can a partially-complete program be executed to obtain feedback on "How am I doing"?
Role-expressiveness	Can the reader see how each component of a program relates to the whole?
Secondary notation	Can programmers use layout, color, or other cues to convey extra meaning, above and beyond the 'official' semantics of the language?
Viscosity	How much effort is required to perform a single change?
Visibility	Is every part of the code simultaneously visible (assuming a large enough display), or is it at least possible to compare any two parts side-by-side at will? If the code is dispersed, is it at least possible to know in what order to read it?

T. Green and M. Petre, Usability analysis of visual programming environments: a 'cognitive dimensions' framework. Journal of Visual Languages and Computing 7(2): 131-174, June 1996

Diffuseness / Terseness

- How many symbols or graphic elements is required to express a meaning?

- Simple rocket simulation program
- Basic: 22 LOC, 140 words (fits on screen)
- LabView: 45 icons, 59 wires (fits on screen)
- Prograph: 52 icons, 79 connectors, 11 screens

T. Green and M. Petre, Usability analysis of visual programming environments: a ‘cognitive dimensions’ framework. Journal of Visual Languages and Computing 7(2): 131-174, June 1996

Error-proneness

- Does the design of the notation induce slips?

- Compared to textual language, VPLs
  - Do not need delimiters & separators
  - Fewer identifiers are needed, easier to reference
  - Constructs inserted automatically (e.g., loops)

Viscosity

- How much effort is required to make a simple change?

- Edit Rocket program to take account of air resistance
- Basic: 63.3 s
- LabView: 508.3 s
- Prograph: 193.6 s

- VPLs required many wires to be rebuilt, layout to be tweaked

Visibility

- Is every (relevant) part of the code simultaneously visible?

- LabView does not show both branches of conditional at same time (!)
  - Particular problem for nested conditionals
- Prograph has poor support for deep nesting of routines

VPLs Discussion

- Often offers a representation that makes specific tasks easy
    - e.g., tracking data flow
    - Often involves structured editor targeted to specific domain, which may not support full range of programs
- But may make other tasks harder
- Often limited focus on scalability

- May be possible to get benefits of task-specific representations without drawbacks through task specific **editor** rather than language