# Design Issues for a Visual Programming Language and Its Programming Environment

Etsuya Shibayama
Masashi Toyoda
Buntarou Shizuki
Shin Takahashi
Tokyo Institute of Technology

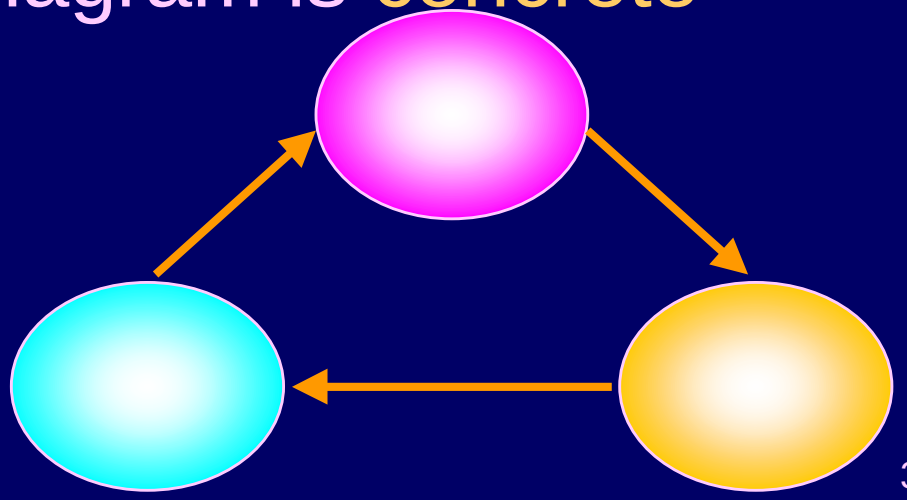
# Overview of the talk

- Why Being Visual?
- A Visual Programming Language (VPL) based on Object Data-flow Diagrams
- Language Design Issues
  - Visual Patterns, Visual Architectures, and Visual Components
  - Pattern-Oriented Programming
- User Interface Design Issues
  - Zooming Interfaces
- Discussions

# Why Being Visual?

- Diagrammatic representations are natural for describing OO designs, programs, and computations
  - e.g., OOA/OOD
  - e.g., Visual Programming Environments
- An object data-flow diagram is concrete and direct

$$\overline{a}.b.P \mid a.\overline{c}.Q \mid c.\overline{b}.R$$



# What else?

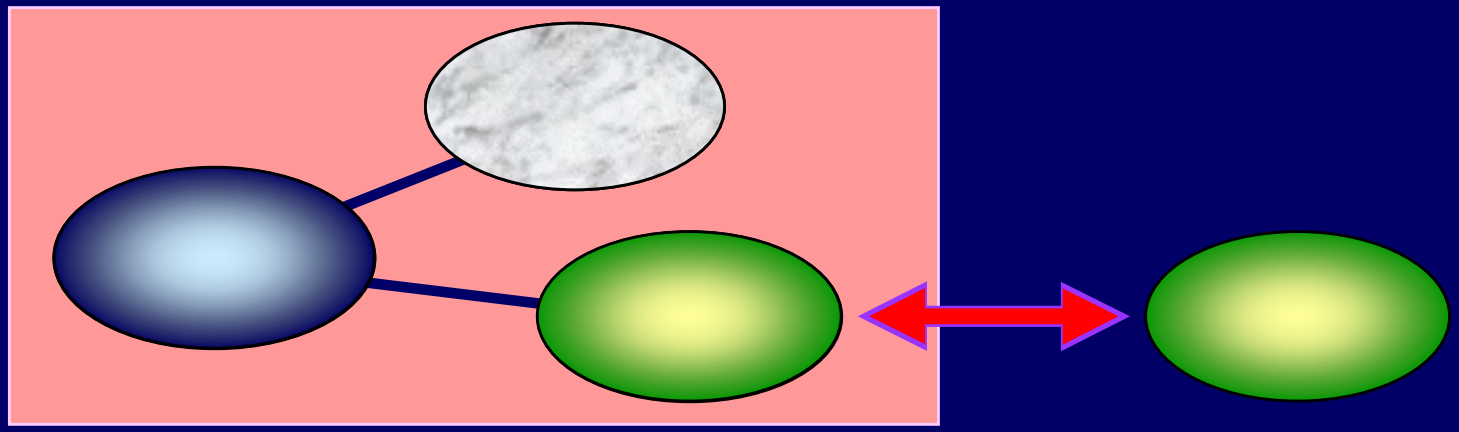
- Diagrams are not always comprehensible
  - e.g., flow-charts
  - Modular structures are necessary
- We need visual language abstractions
  - i.e., concrete abstractions
  - objects, patterns, components, architectures
- We need user interface supports
  - Interactive and scalable interfaces for editing and navigation

# Our Work

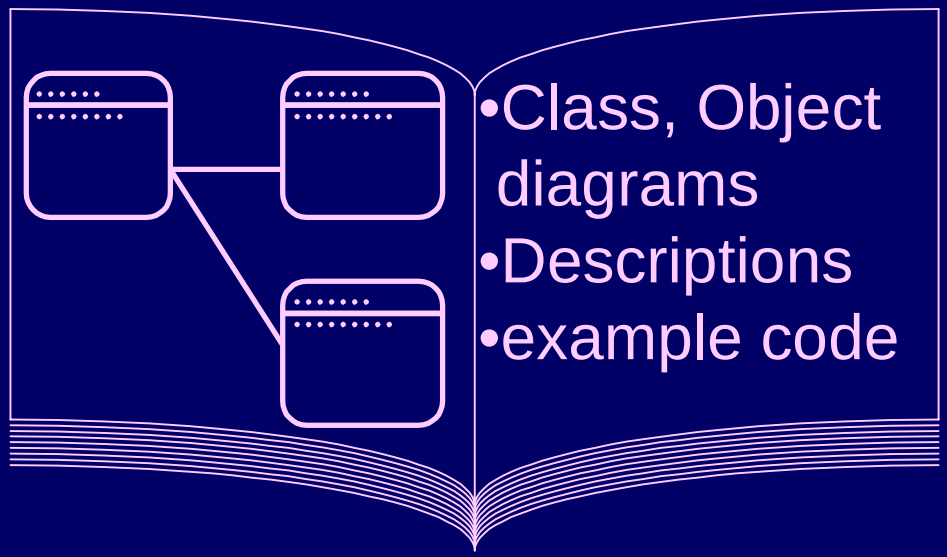
- An Object-Based Parallel Visual Programming Language KLIEG
- A uniform pictorial notation for designs, programs and computations
- Visual abstractions for objects, streams, patterns, components, and software architectures
  - direct manipulation and zooming interface
  - layout information and design information

# Reusable OO systems

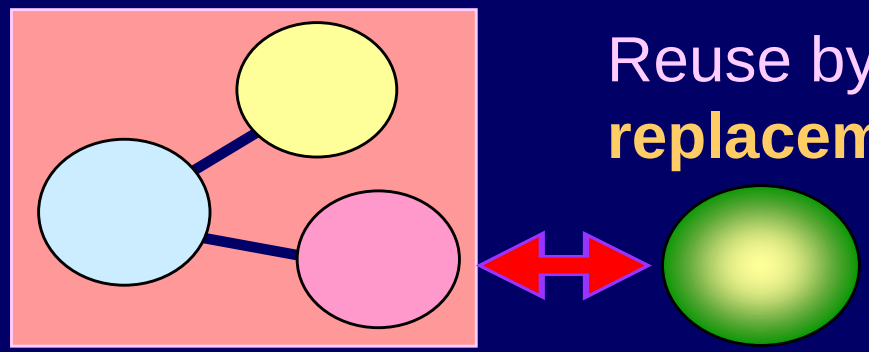
- An OO system is a collection of objects
- It can be flexible if some of the objects are designed to be replaceable and extensible
- It can be reusable if it is flexible and information for reuse is available



# Design Patterns for OO Systems




Implementation with patterns



Reuse by **replacement**

- Design patterns are documents including:
  - coding techniques for replaceable and extensible objects
  - design information for reuse

# Observations

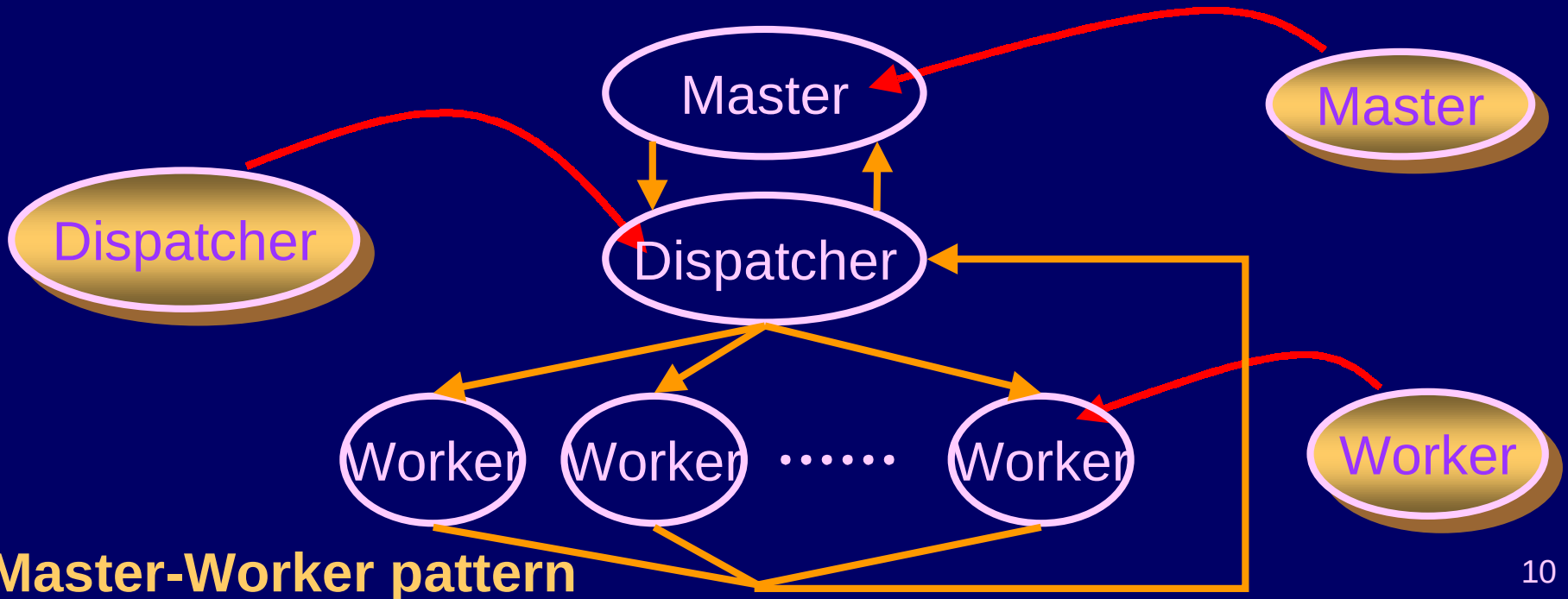
- A replaceable object is an important notion
  - A visual environment could provide interactive supports instead of hacking techniques
- Design information is more important
  - It is a challenge to provide a visual support for design information
  - Pieces of design information are often lost during coding processes
  - Design patterns are merely documents

# Our Approach: Visual Design Patterns (VDP)

- A program is an object data-flow diagram
- An interactive support for replaceable objects
- VDPs as visual programming constructs with design information
  - Simple visual interfaces for definition, (re)use, and customization of VDPs

# Replaceable Objects in VDP

- A VDP is an object data-flow diagram with abstract objects (holes)
- Visual interfaces for:
  - instantiation of a hole with a concrete object
  - replacement of an object with another object




**Master-Worker pattern**

# Significant Design Information for VDP Systems

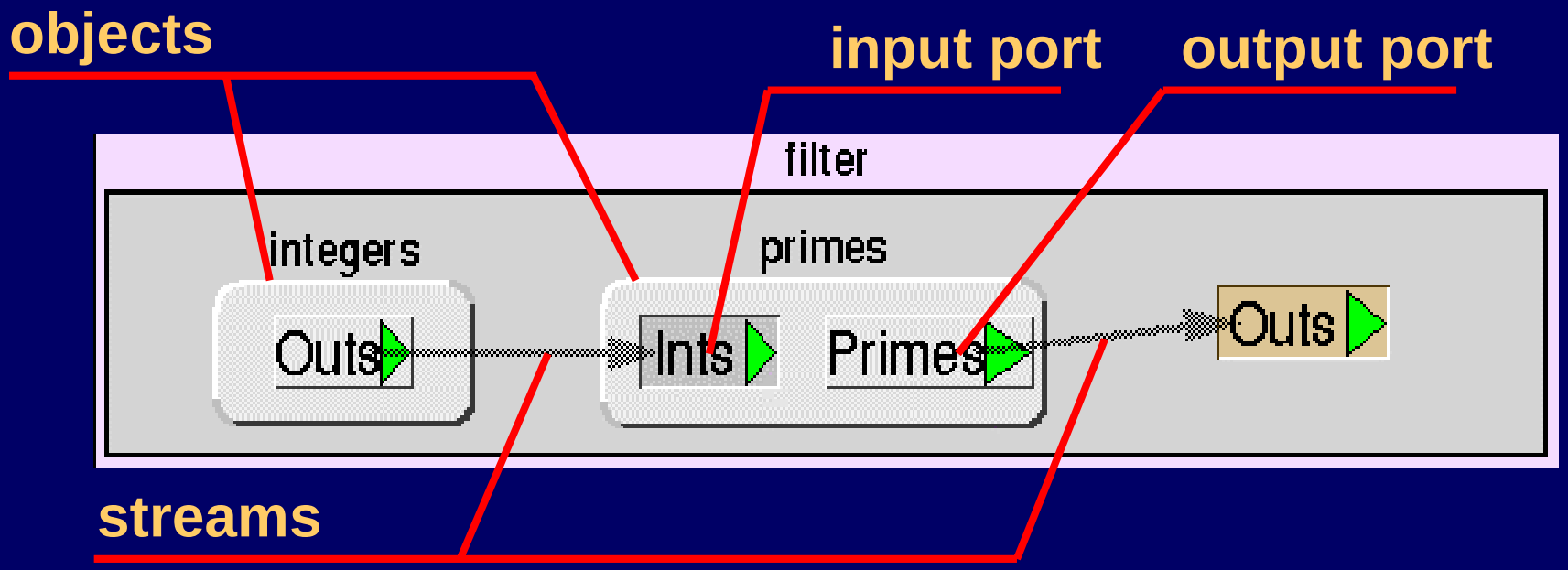
- Aspects of VDP
  - replaceable portions for particular behaviors
  - Which holes or objects shall be replaced?
- Available alternative implementations of each hole
  - Which object it shall be replaced with?
- Dynamic behaviors of VDP
  - How it works?

# KLIEG-VDP System

- The system provides user interfaces for design information
  - Multiple views of VDP by a multi-focus fisheye zooming interface
  - Addition, deletion, and selection interfaces for multiple alternative implementations
  - KLIEG tracer will automatically animates behaviors of sample codes

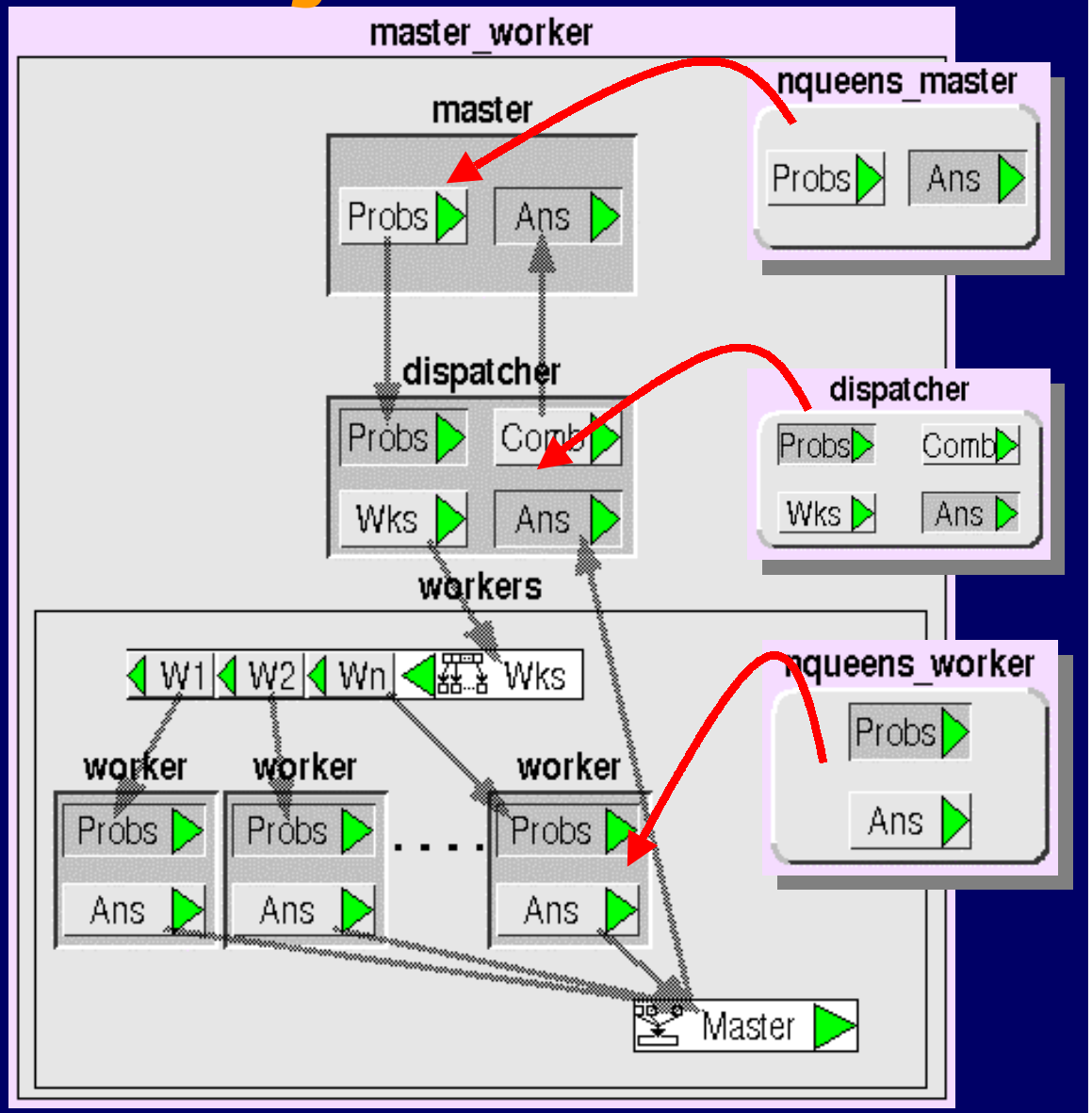
# KLIEG Language

- A program is a collection of diagrams consisting of objects and streams



# Replaceable Objects

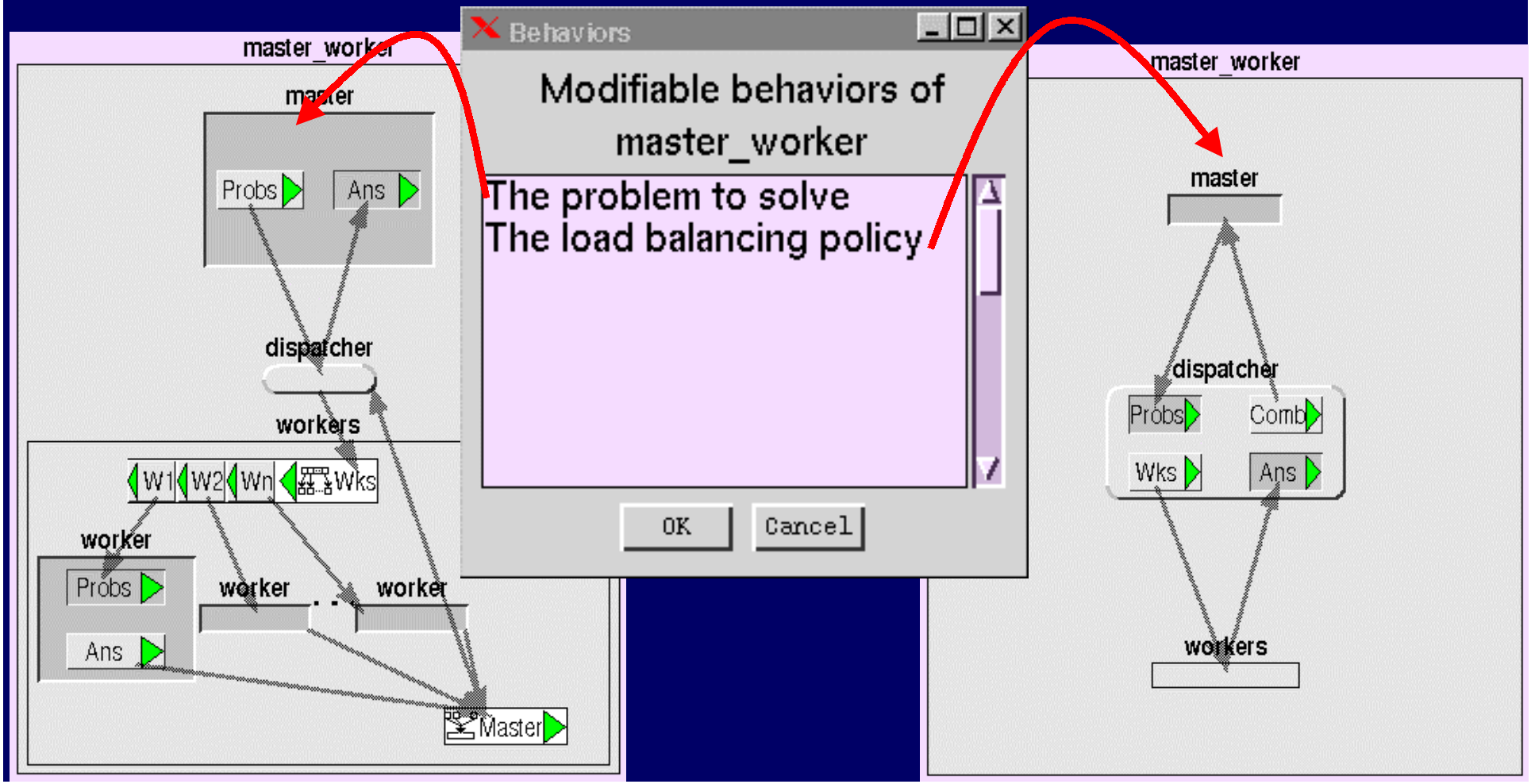
- Supports for editing diagrams with holes
- Instantiation & replacement by drag&dropping
- Almost automatic port connections by type checking & heuristics




**Master-Worker pattern in KLIEG-VDP**

# Multiple Views of KLIEG-VDP

- The creator can register multiple views of a VDP using multi-focus fisheye view
- A user can select a view

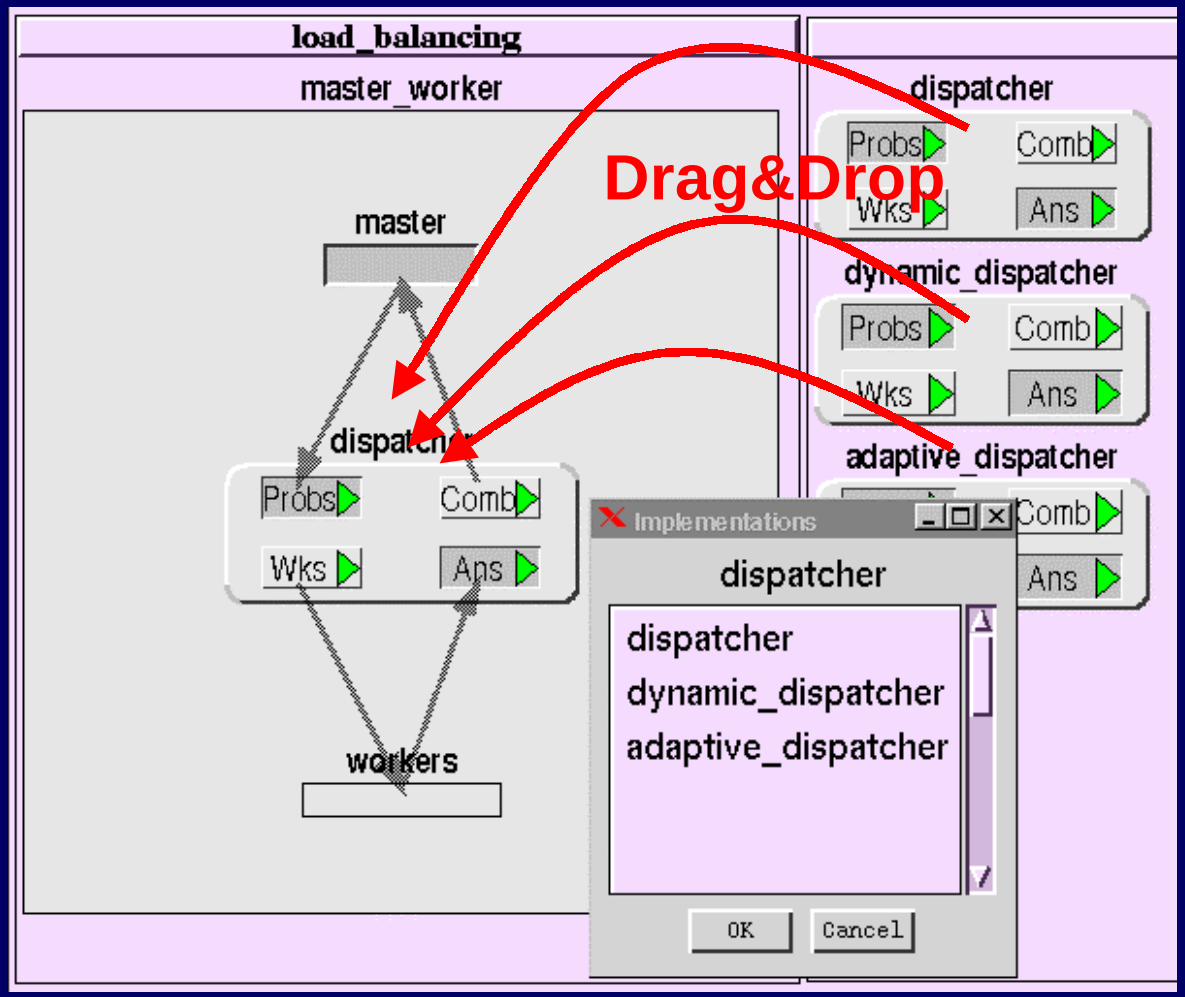


# Multiple Implementations

- A hole may hold multiple objects
- A user can select an object with a dialog box

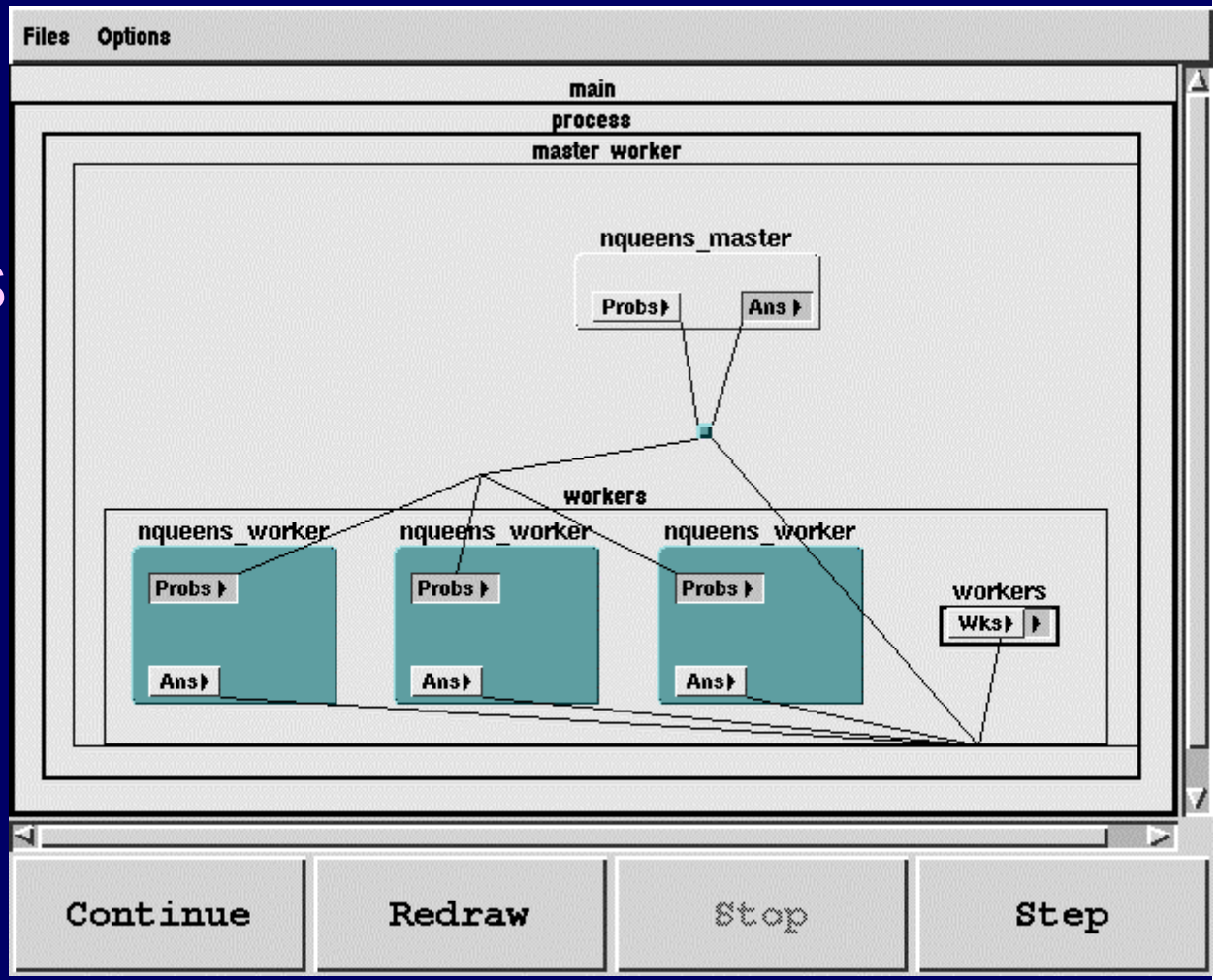
Types of implementations

- Default
- Alternatives
- Sample



# KLIEG Tracer

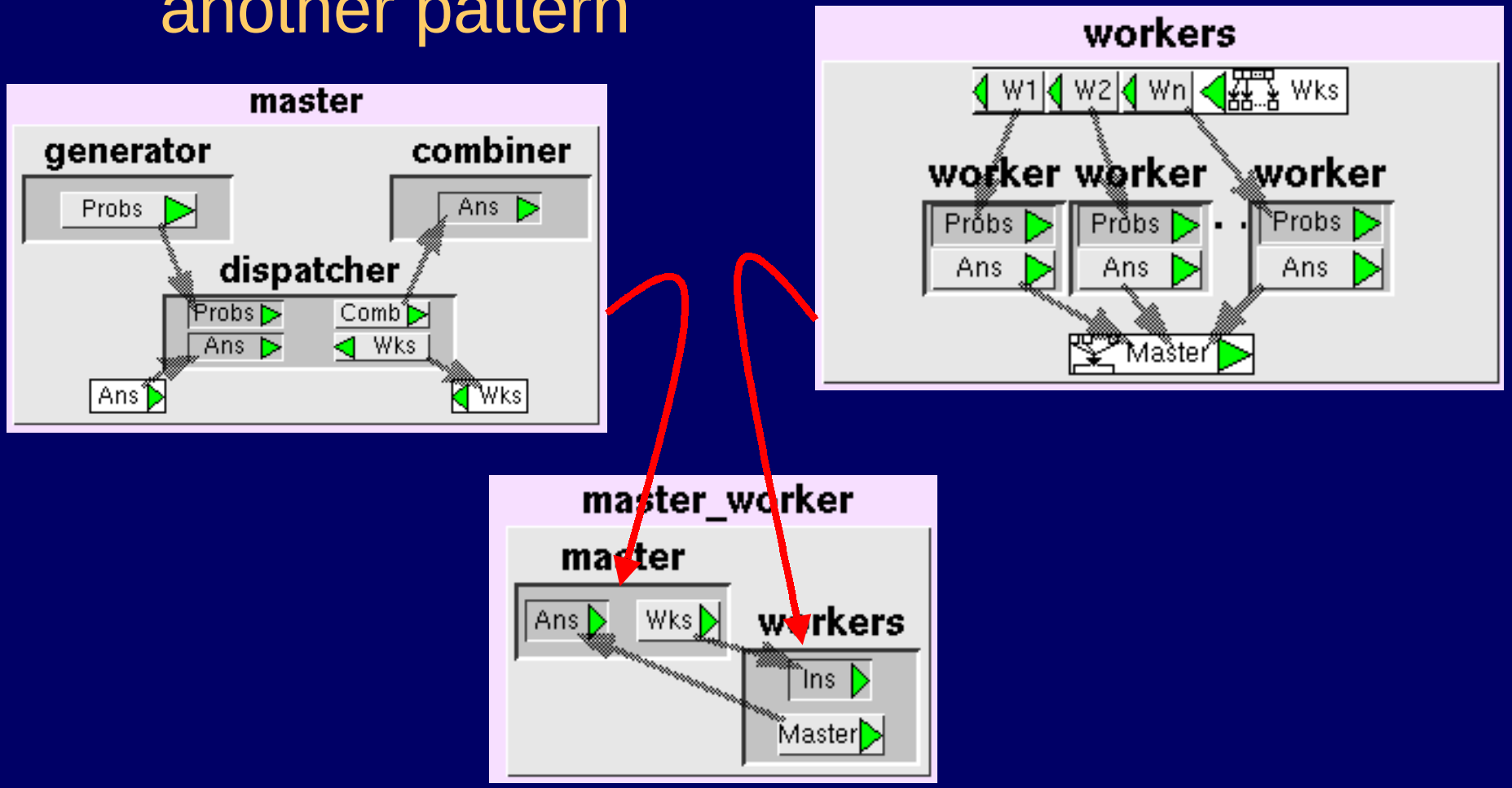
- Visualizes and animates transitions of diagrams
- Maintains VDP layouts
- Shows contents of streams
- Supports multiple views for accessing design info.



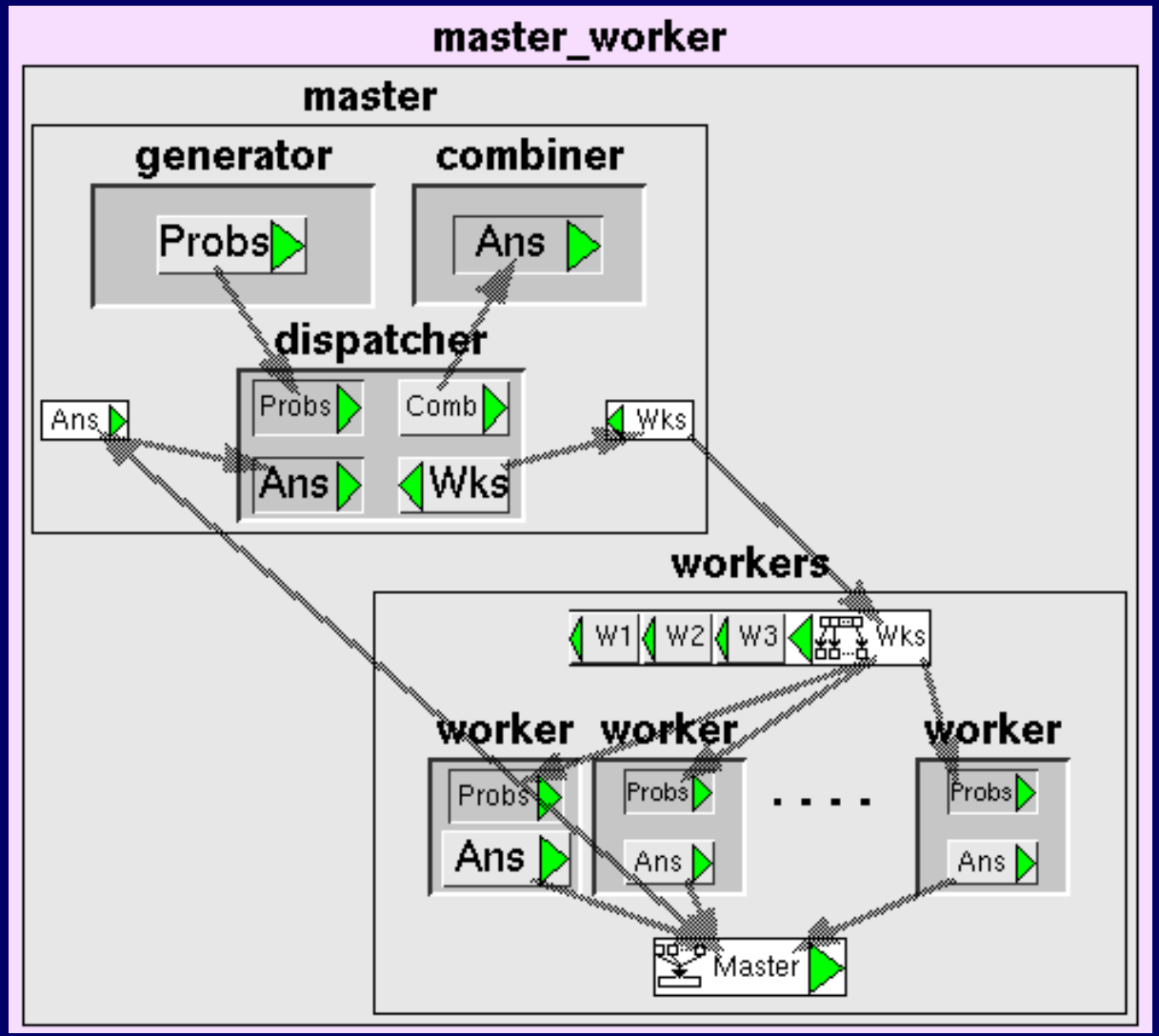

KLIEG execution tracer

# Hierarchical Patterns

- A hole of a pattern may be replaced with another pattern

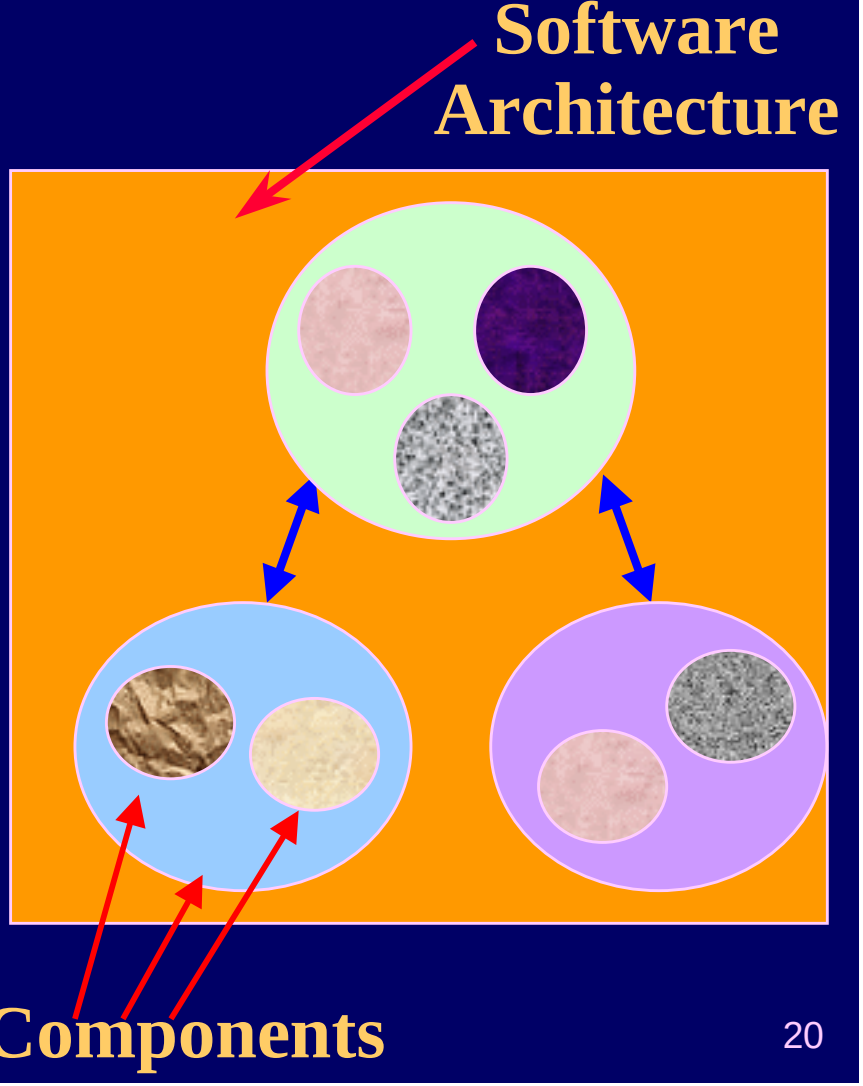


# Hierarchical Patterns



# Pattern-Oriented Visual Programming

- Define the software architecture by hierarchically composing patterns
- Later, replace holes with concrete objects
- Software will evolve by replacing components (objects or patterns)

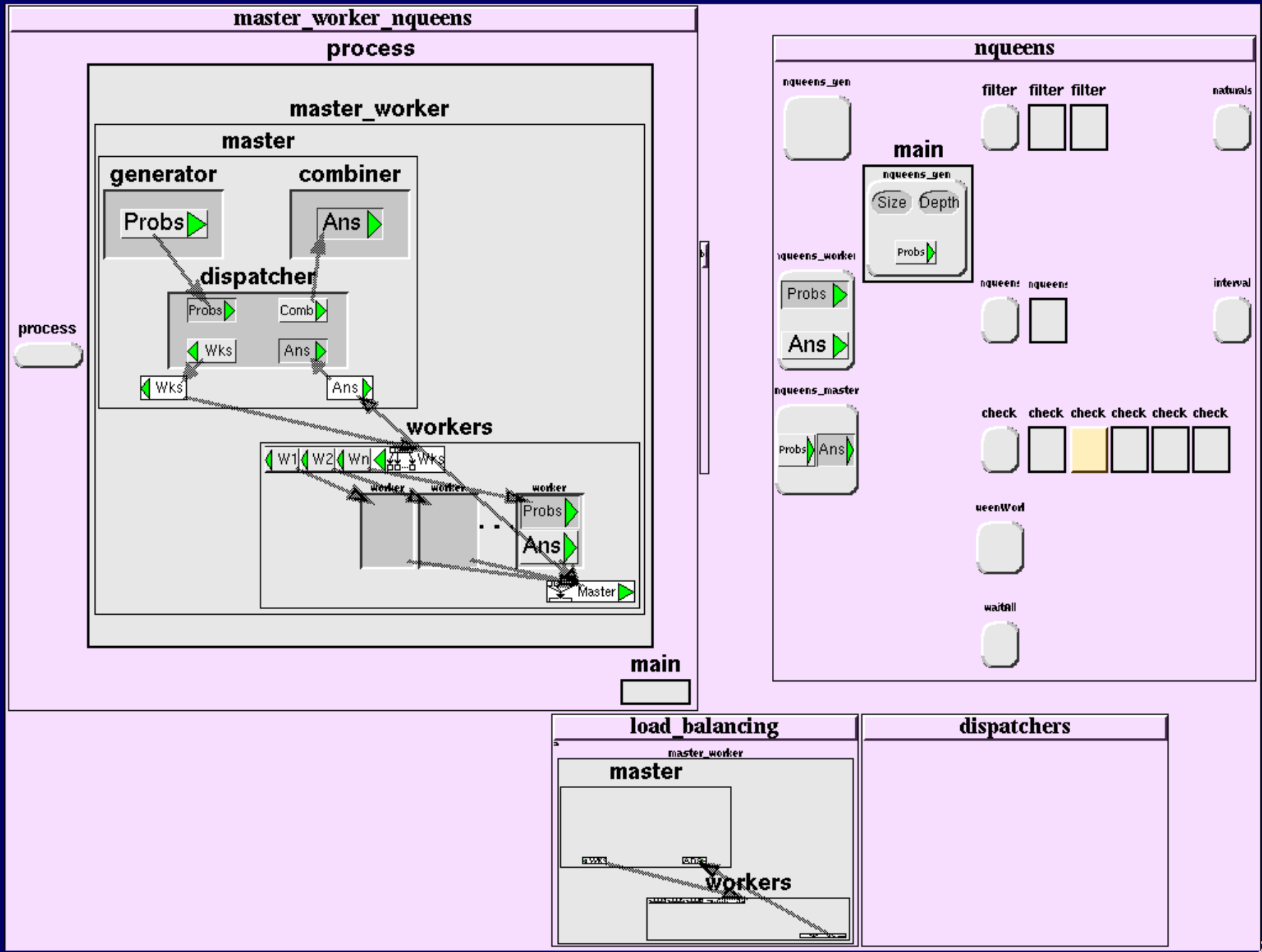


# Scaling-up Visual Programming

- The Problem
  - A program can be large and the screen size is relatively small
  - A computation or even a snapshot of it can be much larger
- Our solution
  - Introduction of multi-focus fisheye zooming interface

# KLIEG Editor

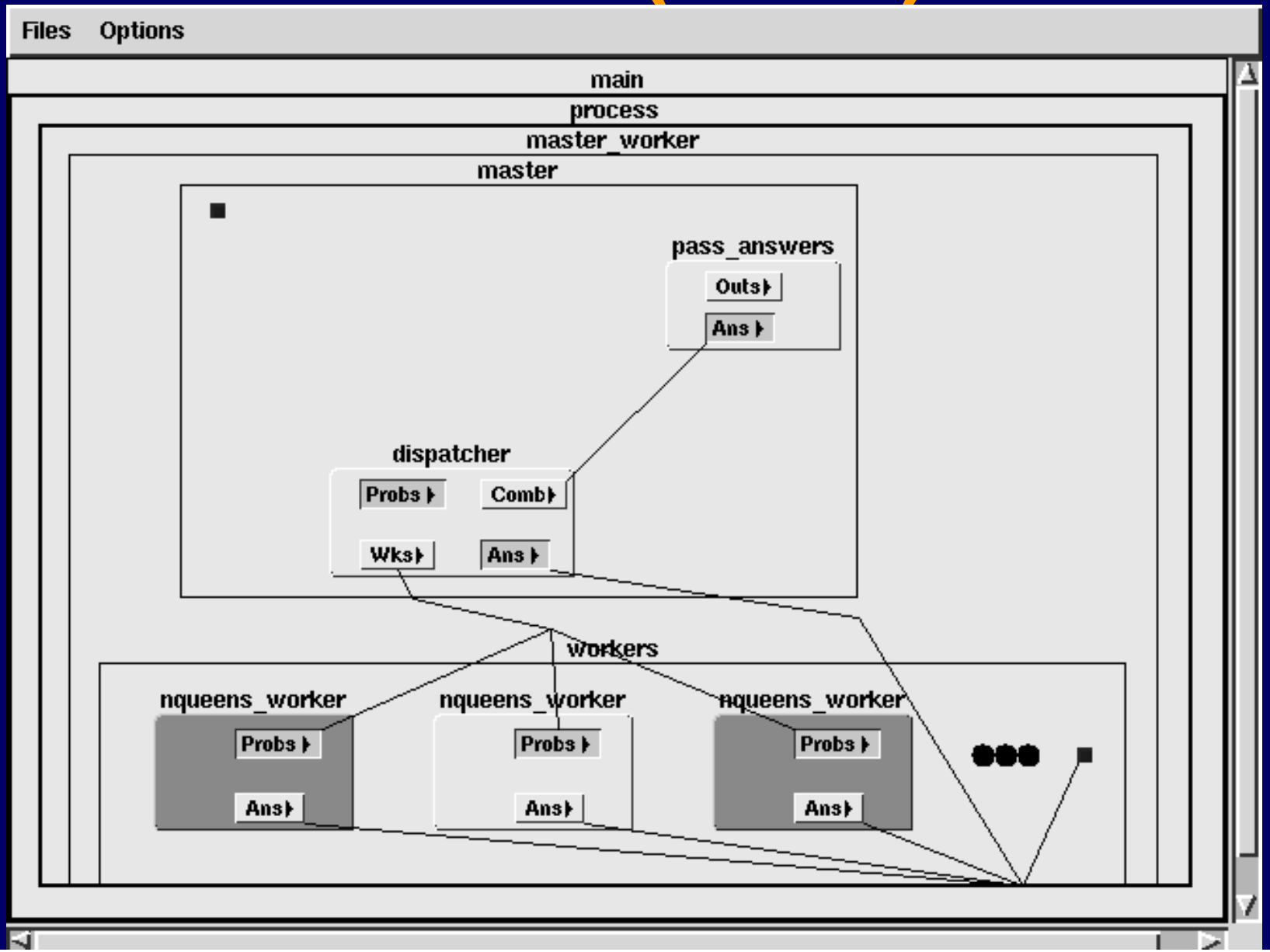
- Every visual is magnified/shrunken
- Multiple portions can simultaneously be magnified for editing
- Multiple views (multiple editing contexts) can be registered
  - Transitions among views are visualized as smooth animations
  - Each aspect of a program may have its own view


master_worker_nqueens
process
master_worker
master
generator
combiner
Probs
Ans
dispatcher
Probs
Comb
Wks
Ans
Wks
Ans
workers
W1
W2
Wn
Wks
worker
Probs
Ans
Master
process
main
nqueens
nqueens_gen
filter
filter
filter
naturals
main
nqueens_gen
Size
Depth
Probs
nqueens_worker
Probs
Ans
nqueens
nqueens
interval
nqueens_master
Probs
Ans
check
check
check
check
check
check
load_balancing
master_worker
master
workers
dispatchers

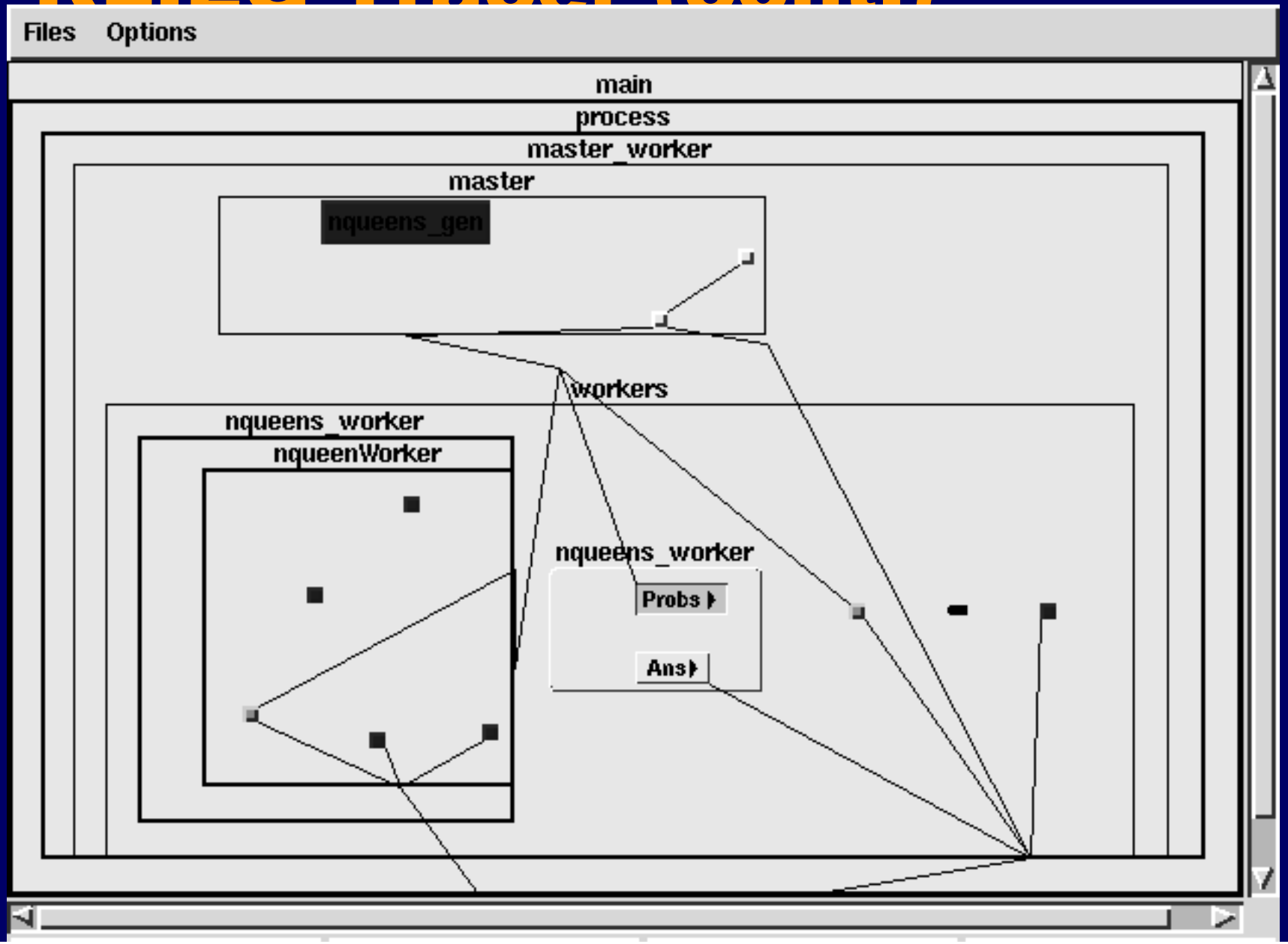
# KLIEG Tracer

- Animating program execution with automatic diagram layout
  - Layouts and views defined by the KLIEG editor can be used
- A multi-focus fisheye zooming interface for browsing computations

# KLIEG Tracer (cont.)



# KLIEG Tracer (cont.)



# Conclusion

- A single visual notation from design/analysis to debugging
- Supports for software evolution
  - A program includes design and layout information that are shared by both editor and tracer
  - Supports for inter-object abstractions
  - Pattern-oriented software constructions
- Scaling-up VPL
  - Zooming interfaces with editing and navigation