**Lecture 9:**

# UI Software Patterns: State Diagrams, MVC, Lexical-Syntax-Semantics


05-431/631 Software Structures for User Interfaces (SSUI)

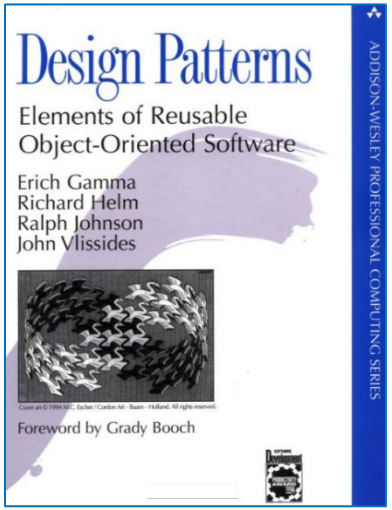
Fall, 2022

# Logistics

- Homework 1 grading done – released
- Homework 2 due today
- Start on Homework 3
  - Instructions now available from the website

# What are Design Patterns?

- Wikipedia: A **design pattern** is the re-usable form of a solution to a design problem. The idea was introduced by the architect Christopher Alexander[1] and has been adapted for various other disciplines, notably software engineering.[2]
- Design Patterns book (1994)
  - By "Gang of 4": Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
  - Patterns for object-oriented systems (originally, Java)
- Now, there are LOTS more



# Design Patterns

- Generally, a way to organize code to achieve certain outcomes
  - Small pieces of the overall design
  - Both “functional” = what it does
  - And “non-functional” = how it does it, e.g., efficiency, reusability
  - Language independent
- We have been using the “Listener Pattern” all along!
  - Also called the “Observer Pattern”
  - Attach an observer (call-back function) to events or other data changing
- We will use the “Command” pattern for undo in Homework 5
- Sometimes called “Models” but that has too many meanings!
- Software Architectures – whole system design; pattern is just one piece

# "Factory" Pattern

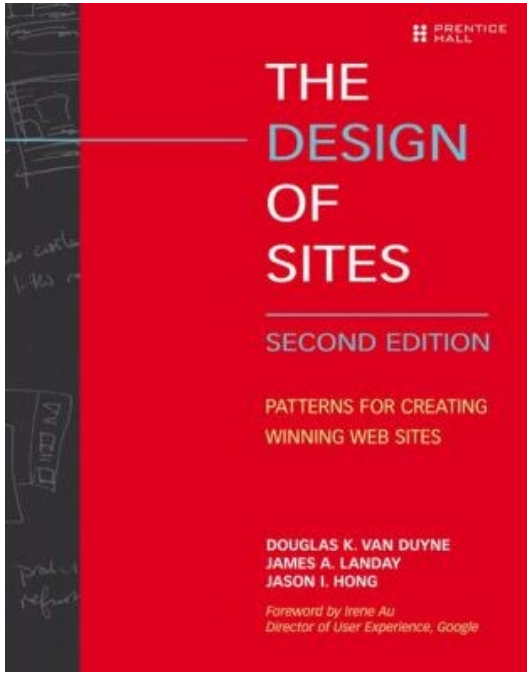
- Instead of "normal" creation: `Widget w = new Widget();`
- Objects must be created by *another* class:

```
AbstractFactory f =  AbstractFactory.getDefault();
Widget w = f.createWidget();
```

- Used frequently in Java (>61) and .Net (>13) and SAP
- Advantages?
  - Don't need to allocate any memory (new not called)
  - Can return a different subtype from what user is aware of
- Our research showed that the "factory pattern" made APIs harder to learn and use (lower "API Usability"):
  - When asked to design on "blank paper", no one designed a factory
  - Time to develop using factories took 2.1 to 5.3 times longer compared to regular constructors (20:05 v 9:31, 7:10 v 1:20)
  - All subjects had difficulties getting using factories in APIs

# Design Patterns for UI *Design*

- E.g.,:
  van Duyne, D.K., James A. Landay, and **Jason I. Hong**, *The Design of Sites: Patterns, Principles, and Processes for Crafting a Customer-Centered Web Experience*. Reading, MA: Addison-Wesley, **2002**.
- Mainly for UI design patterns, for UI *designers*, not for UI software



# Conceptual-Semantic-Syntactic-Lexical-Pragmatic

- Another way to look at the design of software
- Derived from compiler theory and language work.
- Mostly relevant to older, non-interactive interfaces
- Pragmatic
  - How the physical input devices work
  - required "gestures" to make the input.
  - Ergonomics
  - skilled performance: "muscle memory"
  - press down and hold, vs. click-click

# Conceptual-Semantic-Syntactic-Lexical-Pragmatic, cont.

- Lexical
  - spelling and composition of tokens
    - “add” vs. “append” vs. “^a” vs. 
  - Where items are placed on the display
  - “Key-stroke” level analysis
  - For input, is the design of the interaction techniques:
    - how mouse and keyboard combined into menu, button, string, pick, etc.

# Conceptual-Semantic-Syntactic-Lexical-Pragmatic, cont.

- Syntactic
  - sequence of inputs and outputs.
  - For input, the sequence may be represented as a grammar:
    - rules for combining tokens into a legal sentence
  - For output, includes spatial and temporal factors
  - Example: prefix vs. postfix
    - Postfix: <select obj> delete
    - Prefix: del *
      <select rectangle tool> <select where new rectangle goes>
    - Infix: drag-and-drop

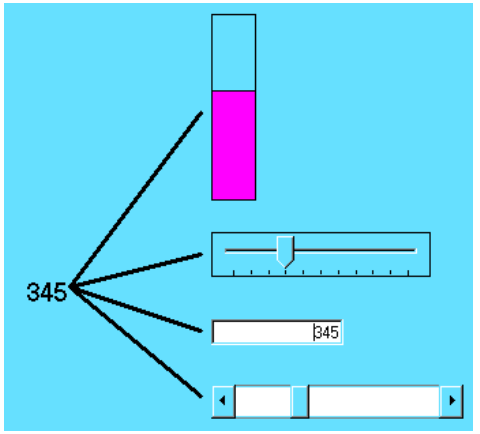
# Conceptual-Semantic-Syntactic-Lexical-Pragmatic, cont.

- Semantic
  - functionality of the system; what can be expressed
  - What information is needed for each operation on object
  - What errors can occur
  - Semantic vs. UI is key issue in UI tools
    - but "semantic" is different than meaning in compilers
  - "Semantic Feedback"
    - Depends on **meaning** of items
    - Example: only appropriate places where item can be dropped highlight during drag

# Conceptual-Semantic-Syntactic-Lexical-Pragmatic, cont.

- Conceptual (definition from Foley & Van Dam text, 1st edition)
  - Key application concepts that must be understood by user
  - User model
    - Objects and classes of objects
    - Relationships among them
    - Operations on them
      - Example: text editor
        - objects = characters, files, paragraphs
        - relationships = files contain paragraphs contain chars
        - operations = insert, delete, etc.
- Overall evaluation of CSSLP model:
  - + "*Separation of concerns*" for input handling
  - + Helps to think about syntax & lexical issues
  - - Not useful for output

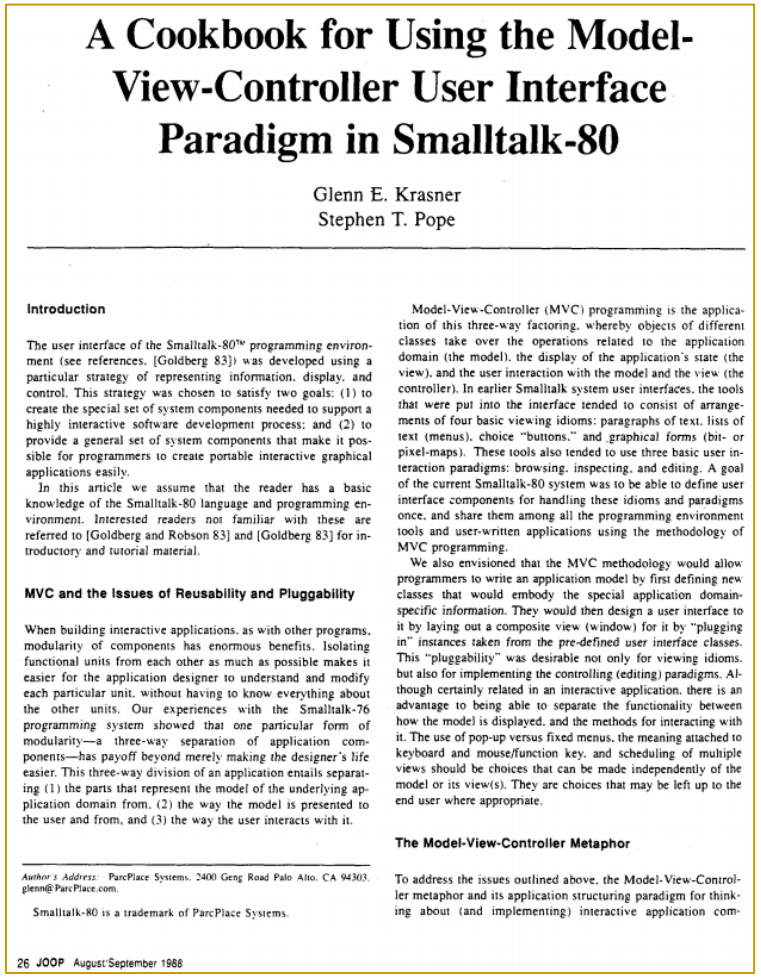
# Model-View-Controller




- Invented in Smalltalk, about 1980
- Idea: separate out presentation (View), user input handling (Controller) and "semantics" (Model) which does the work
- Fairly straightforward in principle, hard to carry through
- Never adequately explained (one article, hard to find)
- Goals
  - Program a new model, and then re-use existing views and controllers
    - better modularization and separation of UI from application
  - Multiple, different kinds of views on same model
- Lots of modern variants

# Classic MVC reference

- Glenn E. Krasner and Stephen T. Pope. "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80," *Journal of Object Oriented Programming. Aug, 1988. vol. 1, no. 3. pp. 26-49.*
- Hard to find original journal
- Luckily, lots of sources for pdf:
- https://www.ics.uci.edu/~redmiles/ics227-SQ04/papers/KrasnerPope88.pdf
- Wikipedia says invented in 1979



A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80

Glenn E. Krasner
Stephen T. Pope

**Introduction**

The user interface of the Smalltalk-80™ programming environment (see references. [Goldberg 83]) was developed using a particular strategy of representing information, display, and control. This strategy was chosen to satisfy two goals: (1) to create the special set of system components needed to support a highly interactive software development process; and (2) to provide a general set of system components that make it possible for programmers to create portable interactive graphical applications easily.

In this article we assume that the reader has a basic knowledge of the Smalltalk-80 language and programming environment. Interested readers not familiar with these are referred to [Goldberg and Robson 83] and [Goldberg 83] for introductory and tutorial material.

**MVC and the Issues of Reusability and Pluggability**

When building interactive applications, as with other programs, modularity of components has enormous benefits. Isolating functional units from each other as much as possible makes it easier for the application designer to understand and modify each particular unit, without having to know everything about the other units. Our experiences with the Smalltalk-76 programming system showed that one particular form of modularity—a three-way separation of application components—has payoff beyond merely making the designer's life easier. This three-way division of an application entails separating (1) the parts that represent the model of the underlying application domain from, (2) the way the model is presented to the user and from, and (3) the way the user interacts with it.

Model-View-Controller (MVC) programming is the application of this three-way factoring, whereby objects of different classes take over the operations related to the application domain (the model), the display of the application's state (the view), and the user interaction with the model and the view (the controller). In earlier Smalltalk system user interfaces, the tools that were put into the interface tended to consist of arrangements of four basic viewing idioms: paragraphs of text, lists of text (menus), choice "buttons," and graphical forms (bit- or pixel-maps). These tools also tended to use three basic user interaction paradigms: browsing, inspecting, and editing. A goal of the current Smalltalk-80 system was to be able to define user interface components for handling these idioms and paradigms once, and share them among all the programming environment tools and user-written applications using the methodology of MVC programming.

We also envisioned that the MVC methodology would allow programmers to write an application model by first defining new classes that would embody the special application domain-specific information. They would then design a user interface to it by laying out a composite view (window) for it by "plugging in" instances taken from the pre-defined user interface classes. This "pluggability" was desirable not only for viewing idioms, but also for implementing the controlling (editing) paradigms. Although certainly related in an interactive application, there is an advantage to being able to separate the functionality between how the model is displayed, and the methods for interacting with it. The use of pop-up versus fixed menus, the meaning attached to keyboard and mouse/function key, and scheduling of multiple views should be choices that can be made independently of the model or its view(s). They are choices that may be left up to the end user where appropriate.

**The Model-View-Controller Metaphor**

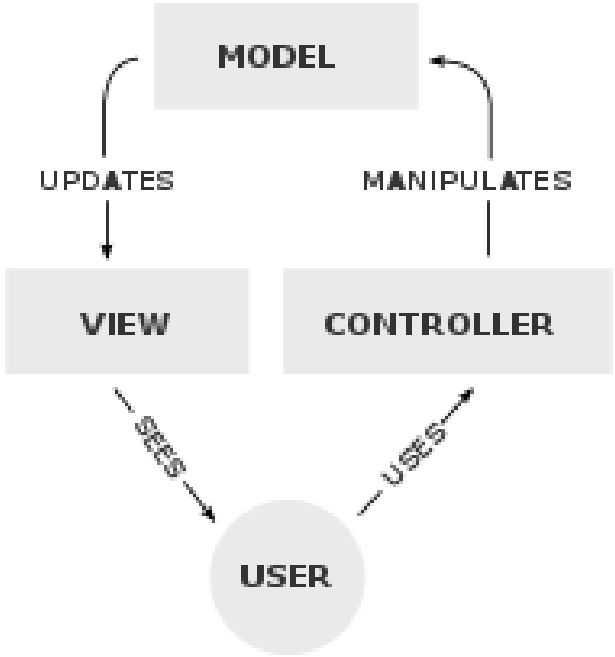
To address the issues outlined above, the Model-View-Controller metaphor and its application structuring paradigm for thinking about (and implementing) interactive application com-

*Author's Address:* ParcPlace Systems, 2400 Geng Road Palo Alto, CA 94303. glenn@ParcPlace.com.

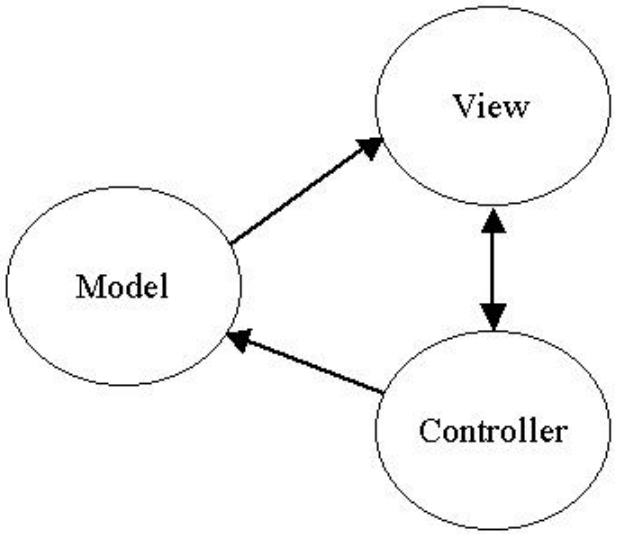
Smalltalk-80 is a trademark of ParcPlace Systems.

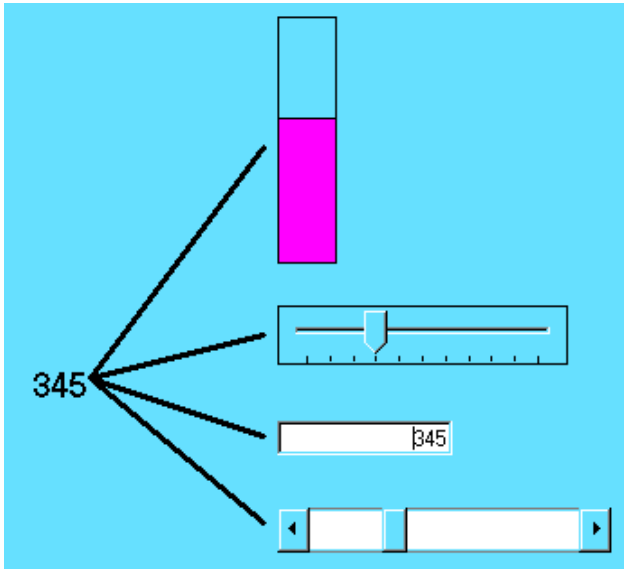
# MVC

- **Model**
  - "the parts that represent the model of the underlying application domain"
  - Simple as an integer for a counter; string for an editor
  - Complex as a molecular simulator
- **View**
  - "the way the model is presented to the user"
  - Everything graphical
  - Layout, subviews, composites
- **Controller**
  - "the way the user interacts with it"
  - Schedule interactions with other VCs
  - A menu is a controller



# MVC



# MVC

- Views closely associated with controllers.
- Each VC has one M; one M can have many VCs.
  - VCs know about their model explicitly, but M doesn't know about views
  - Changes in models broadcast to all "dependents" of a model using a standard protocol.
  - “Observer” pattern again
    - “register” the views with the model, so each is called when model changes
      - Like the event handlers when mouse down

# MVC

- Standard interaction cycle:
  - User operates input device, controller notifies model to change, model broadcasts change notification to its dependent views, views update the screen.
  - Views can query the model
- Problems:
  - Views and controllers tightly coupled
  - What is in each part?
  - Complexities with views with parts, controllers with sub-controllers, models with sub-models...

# Model-View (variant of MVC)

- Since hard to separate view and controller
- Primary goal: support multiple views of same data.
  - Simply switch views and see data differently
- Put into Model "part that needs to be saved to a file"
  - but really need to save parts of the view

# MVC and React / Angular / Vue

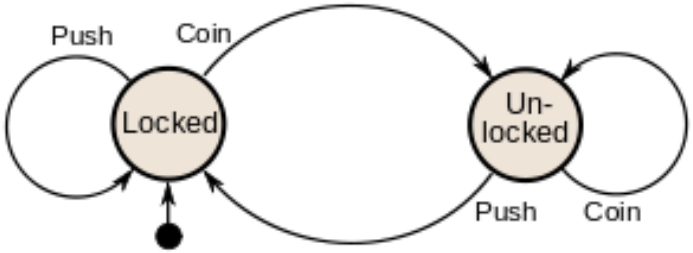
- AngularJS from Google (~2009) relied on the MVC paradigm
- React from Facebook (~2011) explicitly rejected MVC due to requirement of copying values around
  - using constraints or the observer pattern
  - React instead uses shared data, which it calls “states”
- New design of Angular (~2016) less use of MVC
- But Vue tool does use MVC

# Note about the word "Model"

- "Model" used in many ways in HCI!
  - **User model** - *the way a user thinks about a system* - [Don Norman]
  - **Color model, undo model** - *the way that feature is presented to the user*
  - **Software models** - *the way that a programmer should think about the implementation*
    - Examples: **imaging model**, **object model** (**class model** or **prototype-instance model** ), **retained object model**, **document model**, **interactors model**, **event model**, and **model-view-controller (MVC) model** (lectures 6 & 7)
  - **Model** of **M**VC - *the data structure that the user interface is connected to*
  - **Model-based design** – (end of lecture 8) *high-level specification from which the* user *interface is automatically generated*
  - **Human Performance Model** and **Fitt's Law Model** – *mathematical representations from which predictions of human performance can be calculated*
  - **Language models** - *representations of how words are used in a natural language* (lecture 23)
  - **3D models** - *computer representations of three-dimensional objects* (lecture 25)
  - **AI/ML models** – the code that implements the pattern matching (or all of the AI)
  - **Fashion models -** *people who show off clothing*

# Transition Diagrams

- Also called: state diagrams, finite-state machines (FSM), finite automata
  - Is also a kind of "model"
- Set of states and set of arcs
- States: mode that the system can be in
- Transition: how system can change states
- Can be abstracted away from UI or code
- Wikipedia example:




# Transition Diagrams

- Probably the earliest tool to help build UIs:
  - William Newman's "Reaction Handler" in 1968
    http://doi.acm.org/10.1145/1468075.1468083

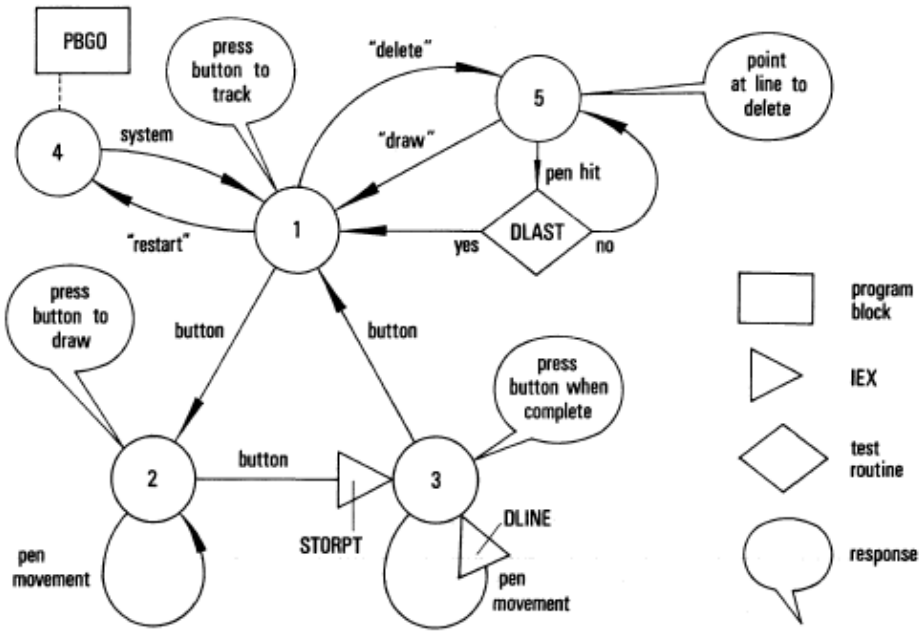



Figure 2—An extended diagram including responses, and with provision for drawing and deleting lines and for initialization

```
STAT  1                    Comment:  State definition, state 1
RESP  PRESS BUTTON TO TRACK          State 1 response, "Press button to track"
 ACT  0                              Branch definition, action of category 0 (command)
 MES  RESTART                          Message "restart"
 SE   4                                State entry, i.e. branch leads to state 4
 ACT  0                              Branch definition; command "delete" leads to state 5
 MES  DELETE
 SE   5
 ACT  10                             Branch definition, category 10 (button)
 SE   2                                Pressing button leads to state 2

STAT  2                              State 2 definition
RESP  PRESS BUTTON TO DRAW           State 2 response
 ACT  7                              Branch definition, category 7 (pen movement)
 ACT  10                             Branch definition; pressing button leads to state 3
 IEX  STORPT                           STORPT stores pen position as starting point when button is pressed
 SE   3

STAT  3                              State 3 definition
RESP  PRESS BUTTON WHEN COMPLETE
 ACT  10                             Branch definition; pressing button leads to state 1
 SE   1
 ACT  7                              Branch definition, pen movement
 IEX  DLINE                            DLINE computes and displays fresh line at every pen movement

STAT  4                              State 4 definition
INIT                                 Initial state, program starts here
PB    PBGO                           Program block PBGO, executed on entering state 4
 ACT  5                              Branch definition, category 5 (system)
 SE   1                                Completion of PBGO leads to state 1

STAT  5                              State 5 definition
RESP  POINT AT LINE TO DELETE
 ACT  0                              Branch definition; command "draw" leads to state 1
 MES  DRAW
 SE   1
 ACT  6                              Branch definition: category 6 (pen hit)
 TEST DLAST                            Test routine DLAST deletes indicated line
 SE   1                                If last line, branch to state 1
END
```

TABLE I: The example of Figure 2 coded into Network Definition Language

# Transition Diagrams, cont.

- Simplest form, arcs are user input events.
  - arcs can be extended by listing feedback (output) and semantic actions on the arcs
  - actions usually written in a conventional language (e.g., C)
  - picture, Olsen, p. 37 (Fig 3:1)

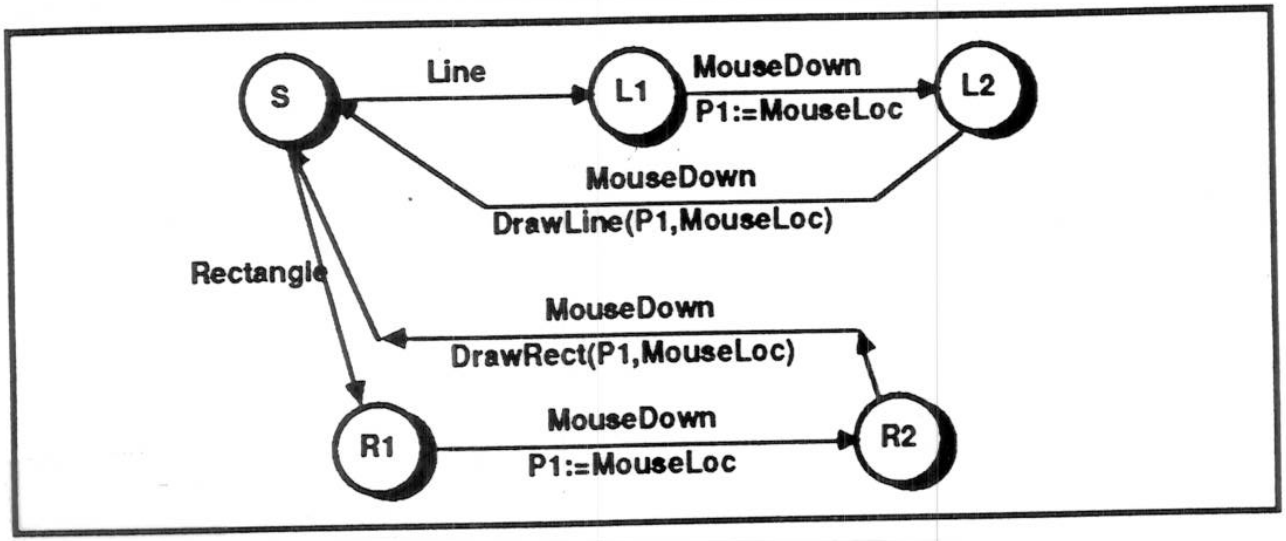



*Fig. 3:1*
*Simple State Machine*

# Transition Diagrams, cont.

- Often, represented textually:
  - picture, Olsen p. 38

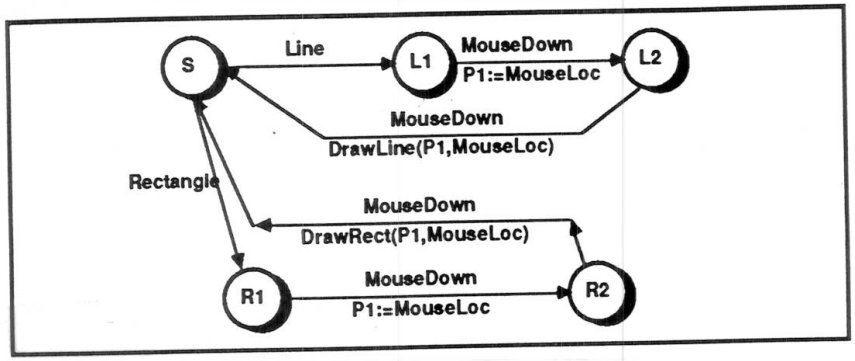



*Fig. 3:1*
*Simple State Machine*

```
State Start:
  On Line Then L1;
  On Rectangle Then R1;
State L1:
  On MouseDown
    Do P1:=MouseLoc;
    Then L2;
State L2:
  On MouseDown
    Do DrawLine(P1,MouseLoc);
    Then Start;
State R1:
  On MouseDown
    Do P1:=MouseLoc;
    Then R2;
State R2:
  On MouseDown
    Do DrawRect(P1,MouseLoc);
    Then Start;
```

# Transition Diagrams, cont.

- "Pervasive states" to handle help, abort, undo, etc.
  - "Escape" transitions to abort (permanently leave) a dialog
    - picture, Olsen p. 53 (Fig 3:11)
  - "Re-enter" sub-dialogs for temporary excursions that return to same place. E.g., help, use calculator, etc.
    - picture, Olsen p. 55 (Fig 3:12)
  - Transitions are taken if no specific arcs from node

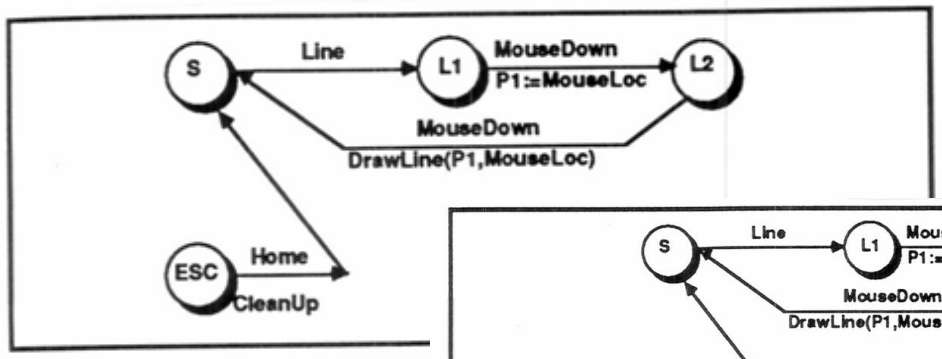



*Fig. 3:11*
*HOME Feature*

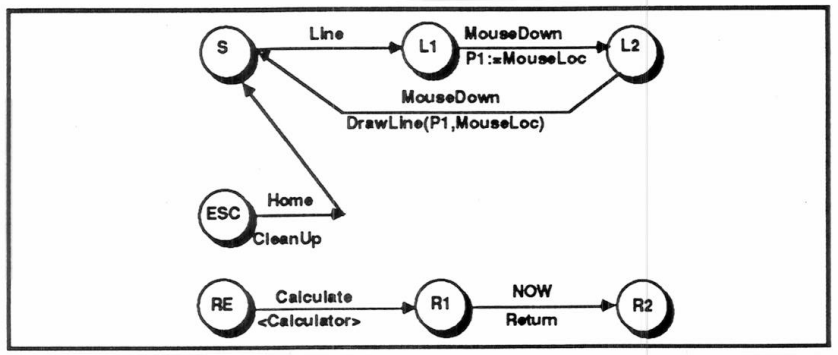



*Fig. 3:12*
*Reenter*

# Transition Diagrams, cont.

- "Augmented transition networks"
  - local variables
  - function on arcs can determine transitions
  - "guards" determine whether transition is legal
  - "conditional transitions" calculate where to go
    - picture, Olsen p. 57 (Fig 3:14)
  - upgrades the power to context-free-grammar

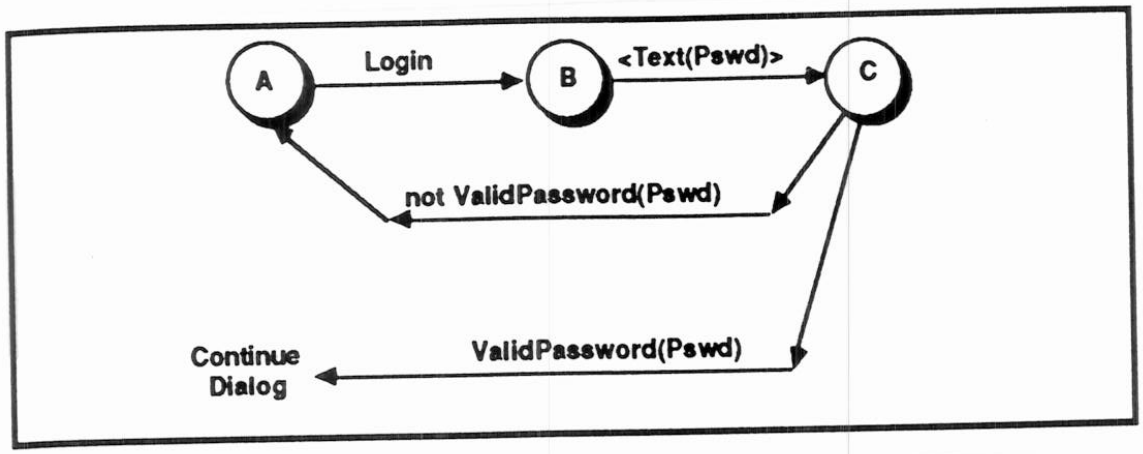


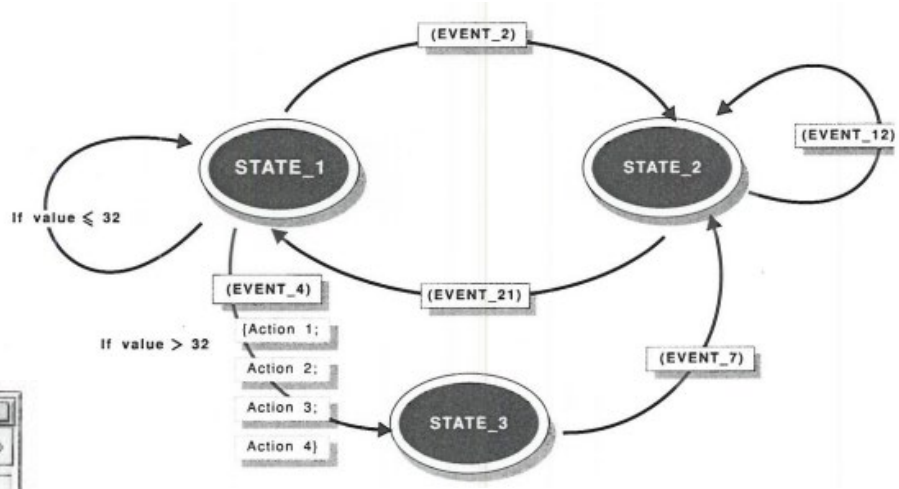

*Fig. 3:14*
*Login Dialog*

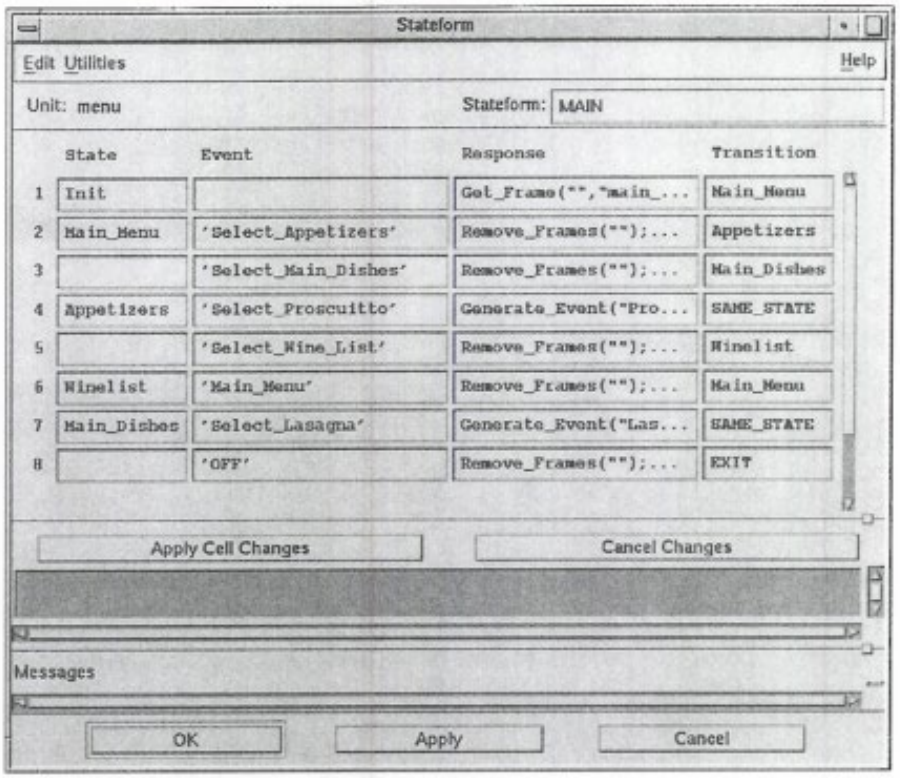
# Transition Diagrams, cont.

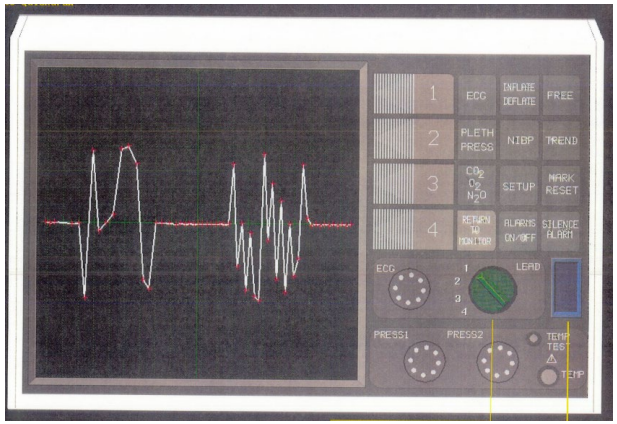
- Transition Networks, in general, used in various tools:
  - Research: Newman's "reaction handler", Jacob's RTN, Olsen's "Interactive Pushdown Automata", Stephen Oney's Euclase, etc.
  - Commercial: IDE's RAPID/USE,
    Virtual Prototypes's VAPS
  - VAPS used spreadsheet interface to the ATN

# VAPS pictures

- Pictures

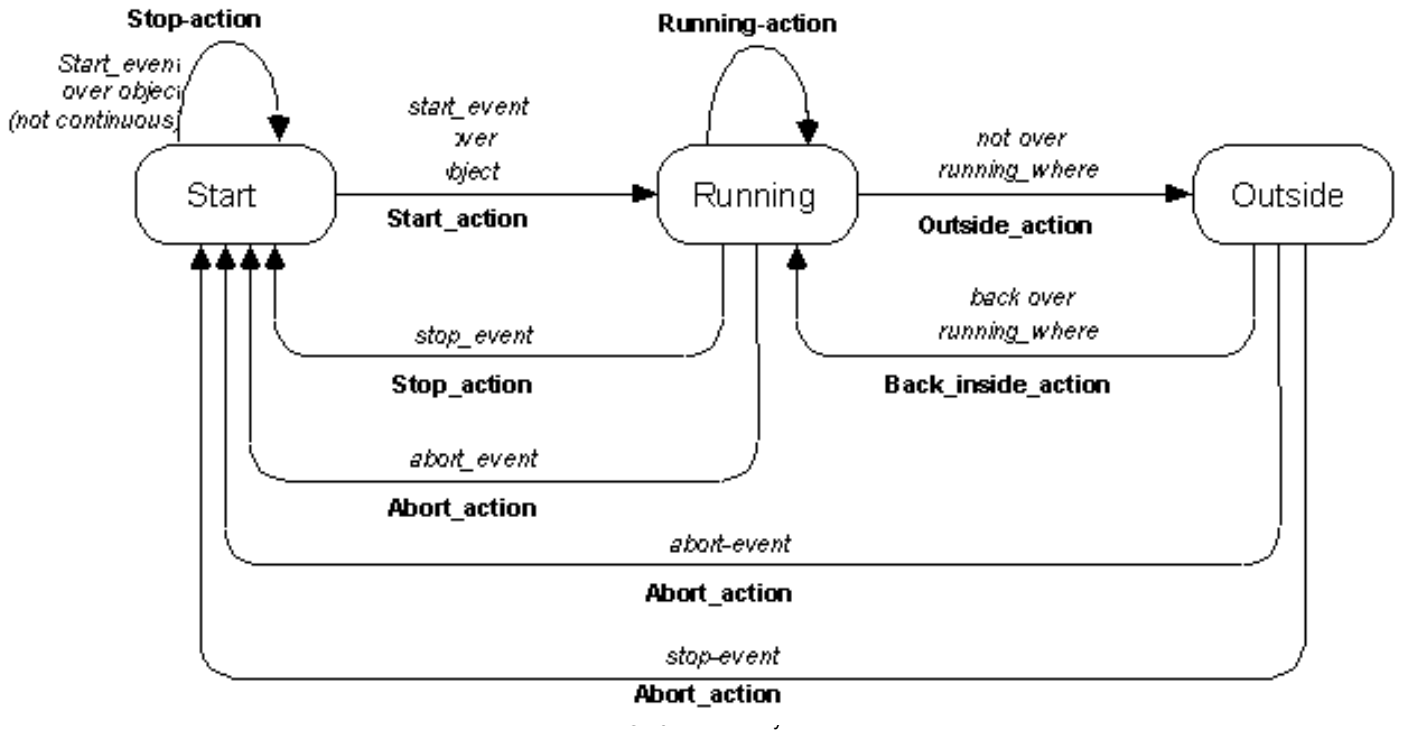


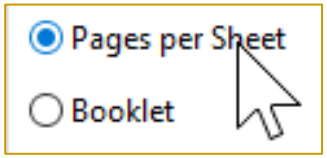



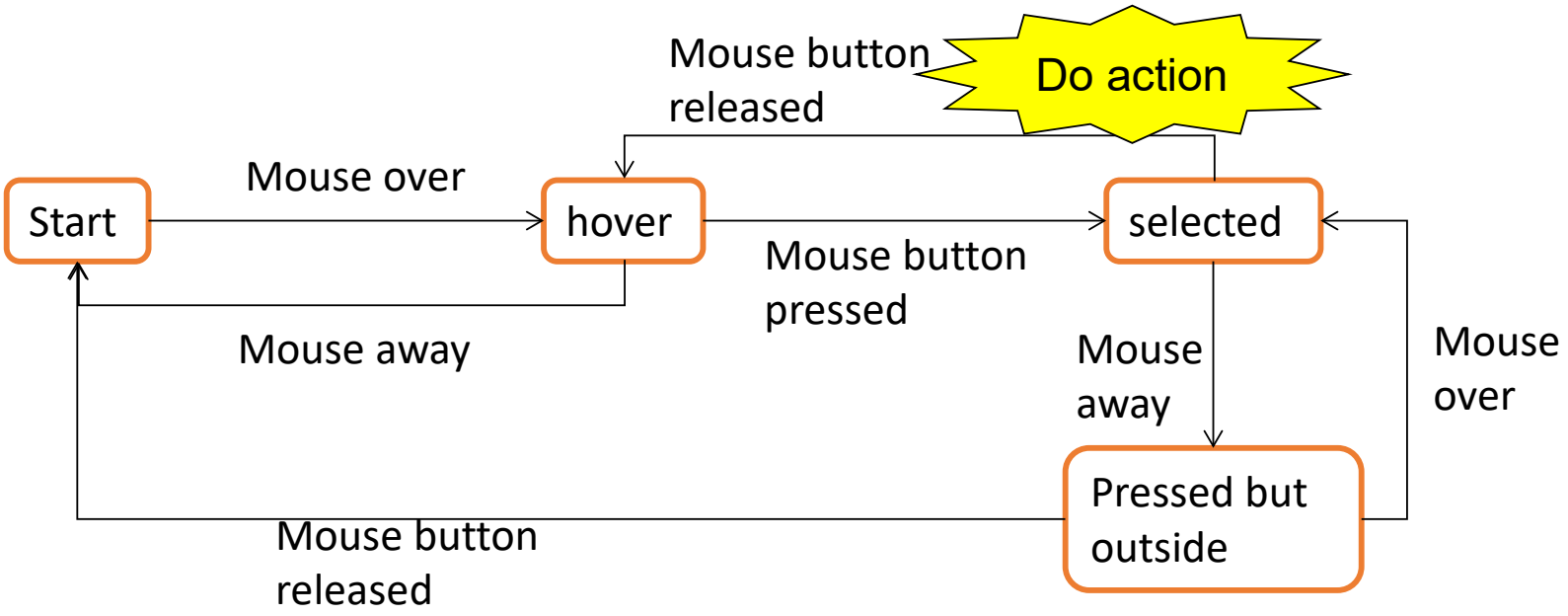
# Transition Diagrams for Widget Behavior

- Used internally for Garnet / Amulet "Interactors" (lecture 5)
  - Note: no "hover" state



# Another example, with hover



# StateCharts

- Invented by David Harel
  David Harel, "Statecharts: a visual formalism for complex systems", Science of Computer Programming, Volume 8, Issue 3, (Elsevier) June 1987, Pages 231-274, online
- Generalization of state diagrams
  - Reduce state explosion and other problems
- Cluster (nested) states for abstractions and identical behaviors
- "AND" states – system is in both A and D
  - E.g., watch light is always on or off, may be independent of other modes
- Concurrent charts – can operate independently
- Many other features
- Used by Stephen Oney's InterState

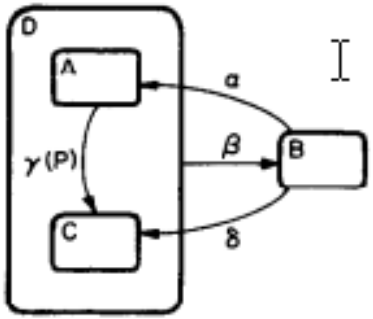



Fig. 2.

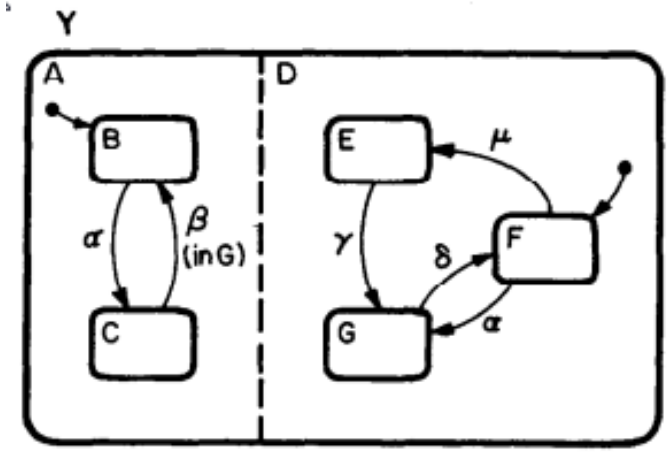
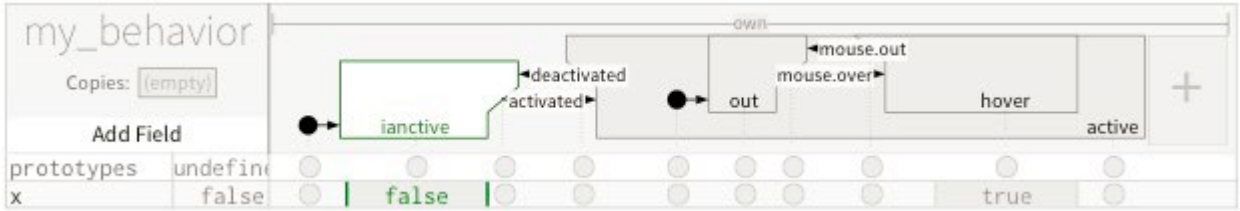



Fig. 19.



# Transition Diagrams, evaluation

- **Good**
  - Make explicit the interpretation of all events in each state
  - Emphasize the temporal sequence of user and system actions
  - Natural and easily understood if small
    - easy to teach, learn, and read
  - Appropriate for some parts of GUIs: widget behaviors, dialog box transitions, etc.
  - May be appropriate to model transitions around web sites

# Transition Diagrams, evaluation

- **Bad**
  - Does not scale:
    150 commands with 3 or 4 states each
    - explosion of lines and states for normal interfaces: "maze of wires"
  - unordered inputs
    - picture, Olsen p. 91 (Fig 6:1)
  - Textual form is like GOTO-based assembly language
  - Communication through global variables
  - Doesn't handle GUI mode-free style well
  - What to do with un-specified input? crash, ignore input
  - Doesn't address output

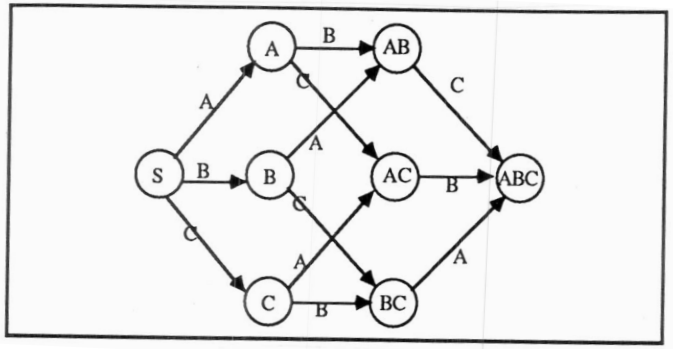
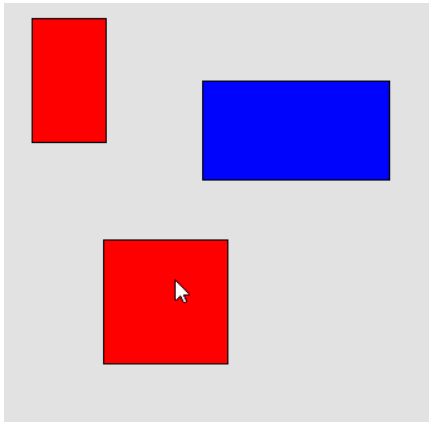



*Fig. 6:1 Handling Unordered Inputs*

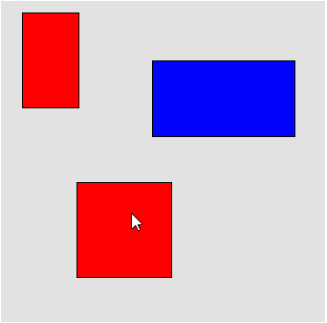
# Let's Design one for HW2, mouse behavior



*Done in class – see next slide*



# Let's Design one for HW2



*Prepared in advance*