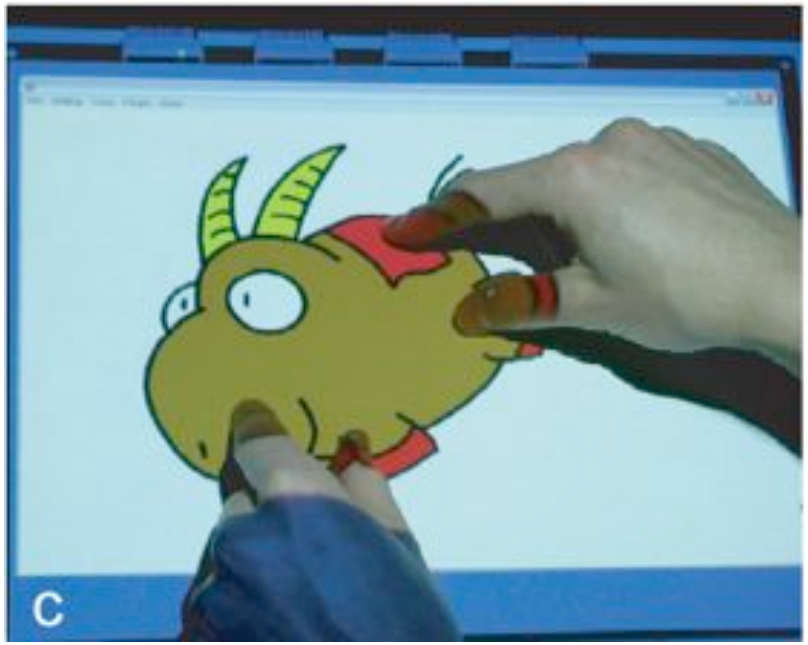
GUI Object Level Architectures

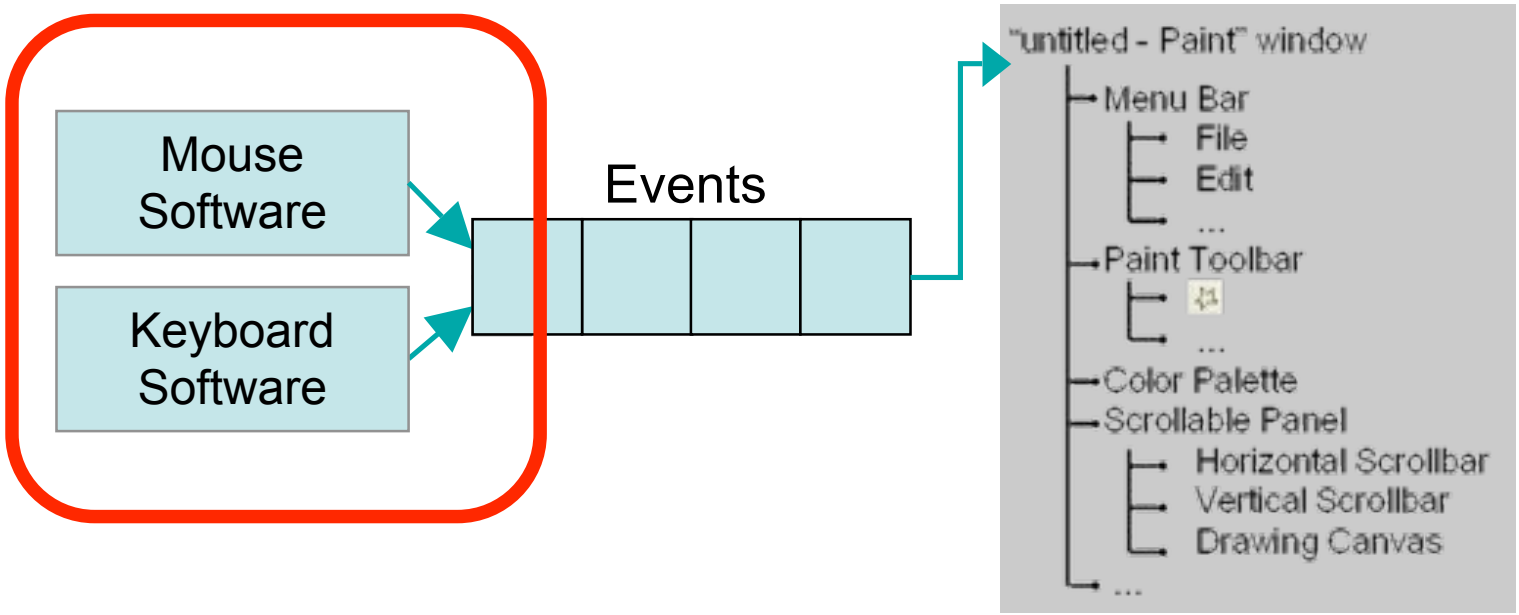
Recap

- Lots of Input Devices
  - Basic input devices (keyboard, mouse, buttons, valuators)
  - Exotic input devices (3D Input, Gloves, Crosspads)
  - Research input devices (Peephole display, speech, touch)



Recap

- Handling input
  - Predefine all kinds of devices (too rigid, didn't work too well)
  - Instead, organize everything as event or sampled devices
  - Handle everything in software as events

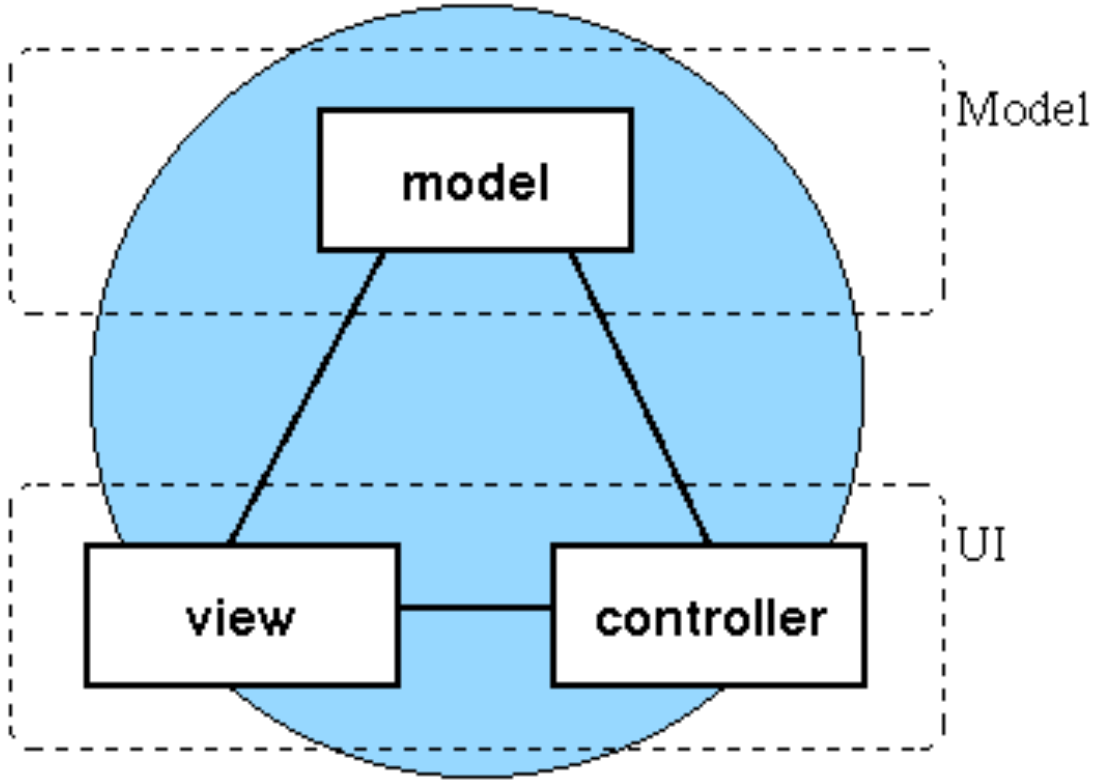


Today

- Object-level architectures
  - Design patterns for GUIs
  - Model-View-Controller
  - Pluggable Look and Feel
  - Undo / Redo

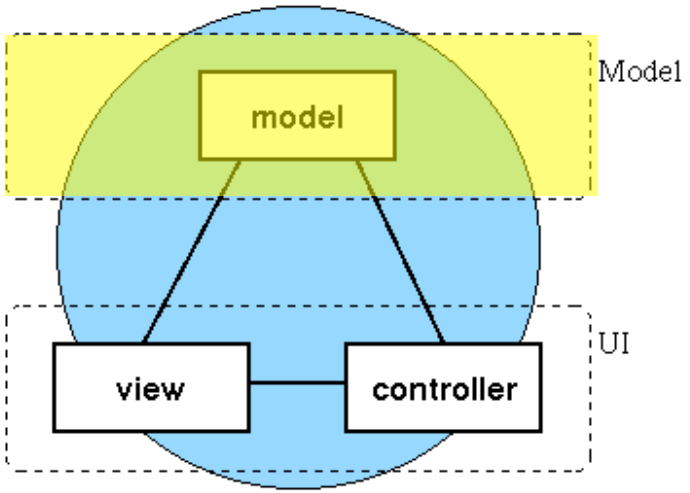
Internal Organization of Widgets

- GUI widgets organized Model-View-Controller (MVC)
  - Basic idea: split widget into three separate objects
  - Each handles different aspect of widget



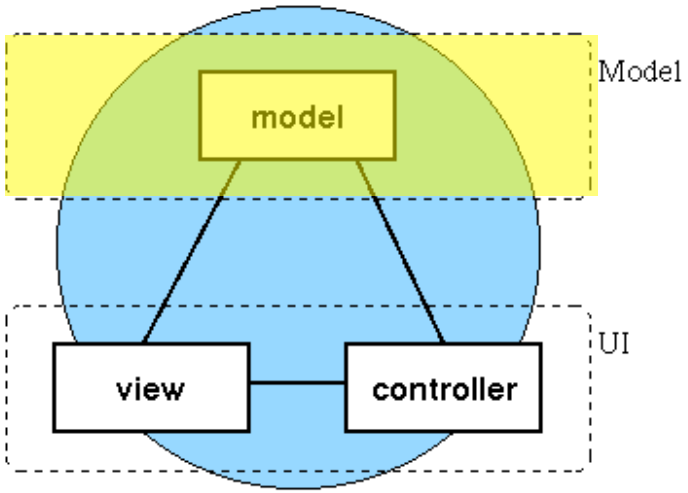
Model-View-Controller

- Model handles core functionality and data
- Micro-level (internal to widget)
  - Scrollbar state
  - Checkbox state
  - What cell in table is currently highlighted
- Macro-level (application)
  - Table data
  - Content in a document
  - Image in paint program



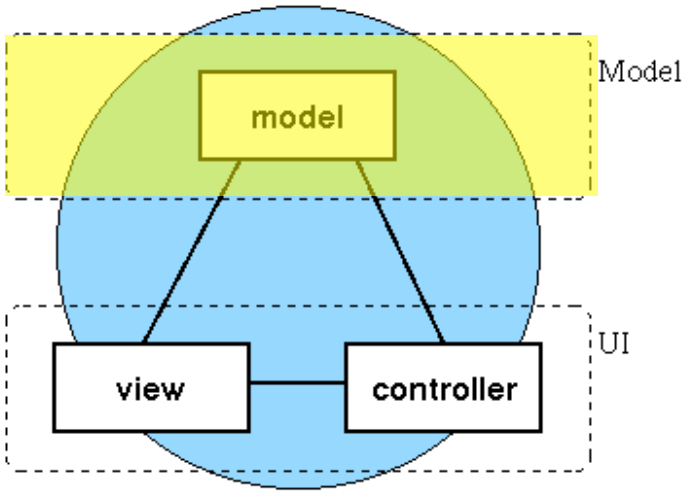
Model-View-Controller

- Model provides:
  - methods to edit data, which Controller can call
  - methods to access state, which View and Controller can request
- Model has registry of dependent Views to notify on data changes
- In Swing, listeners stored here



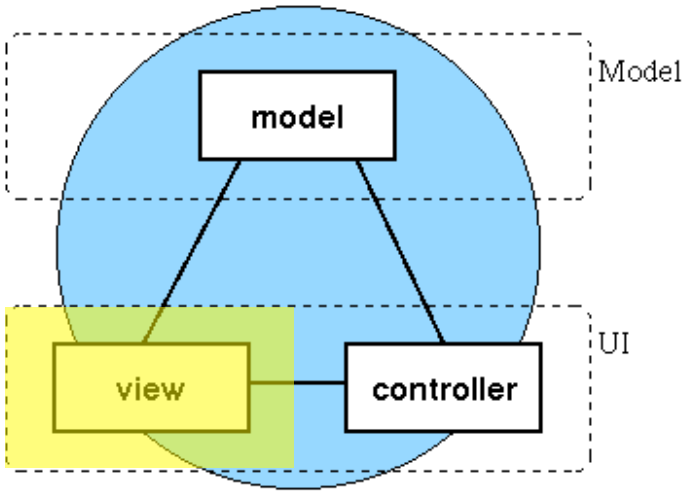
Model-View-Controller

- Model examples:
  - text editor: model is text string
  - slider: model is an integer
  - spreadsheet: collection of values related by functional constraints



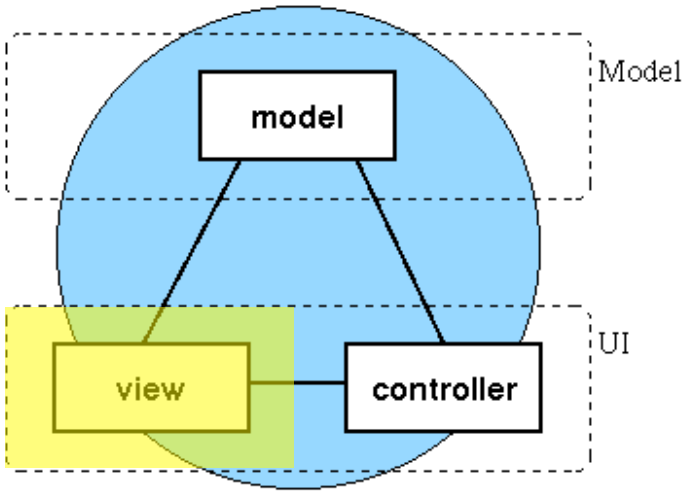
Model-View-Controller

- View handles how the widget appears
  - Handles display of information in Model
  - Handles rendering
- View is informed when Model changed
  - View requests relevant model information
  - View arranges to update screen
    - Declare damaged areas
    - Redraw when requested



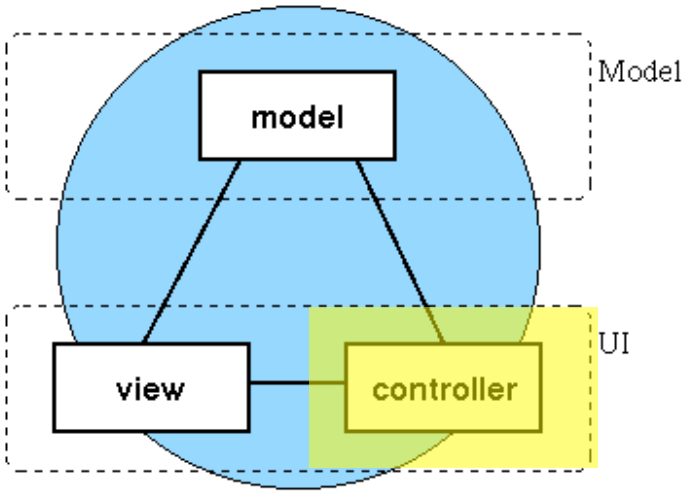
Model-View-Controller

- View Examples:
  - Slider: text-field, temperature gauge
  - Spreadsheet can have multiple views of same model
    - Tabular representation
    - Bar chart
    - Histogram



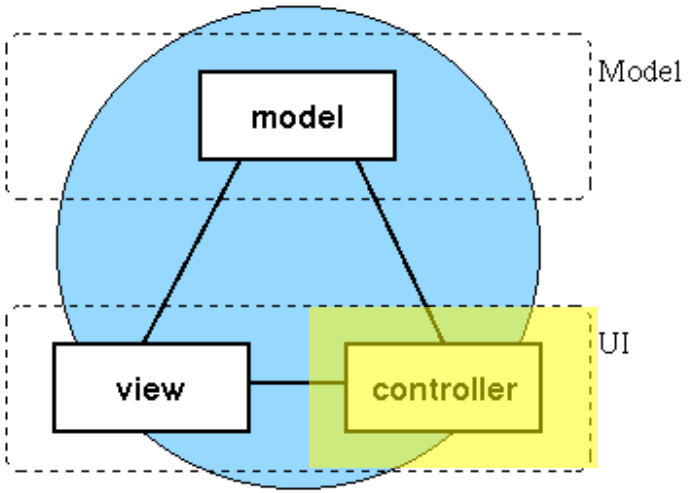
Model-View-Controller

- Controller handles widget input
  - Handles all events as needed
  - Calls appropriate methods in Model to change state



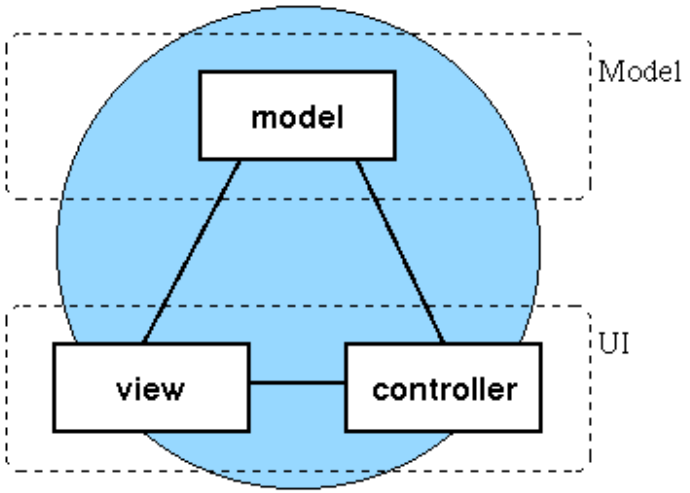
Model-View-Controller

- Controller handles widget input
  - Handles all events as needed
- Controller Examples:
  - Transforms keyboard shortcuts to commands
  - Transforms mouse input to commands



Model-View-Controller Recap

- Split up a widget into three basic parts
- Change-propagation mechanism ensures consistency between Model and UI
  - Model tells View when it's been updated
  - Controller tells View and Model when mouse click happens

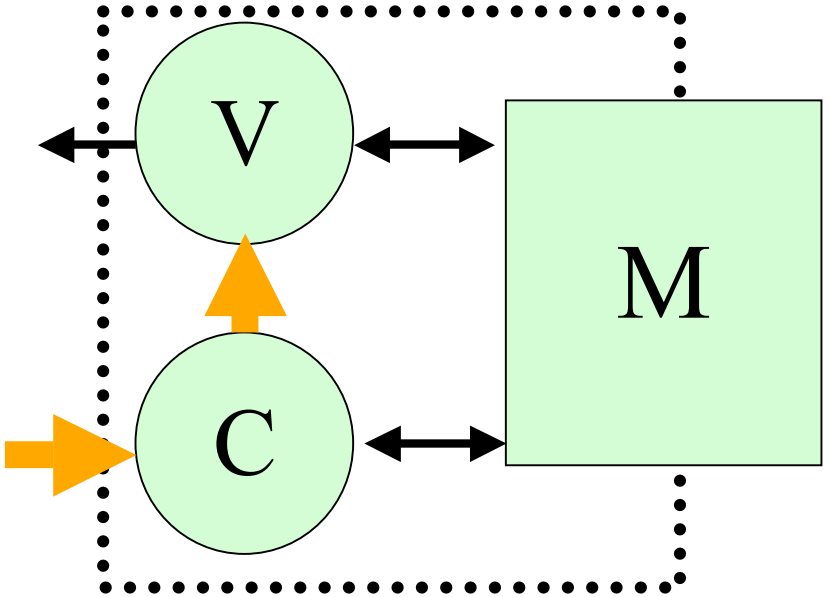


MVC motivation

- Q: Why MVC?
- A: Flexibility and robustness:
  - Same info can be shown in different windows
    - Changes to underlying data should be reflected quickly everywhere (spreadsheet)
  - Changes to UI should be easy, even at runtime
    - Different “look and feel” should not affect functional core (separate front-end from back-end)
  - Easier to program, forced you to be honest, and had cleaner semantics

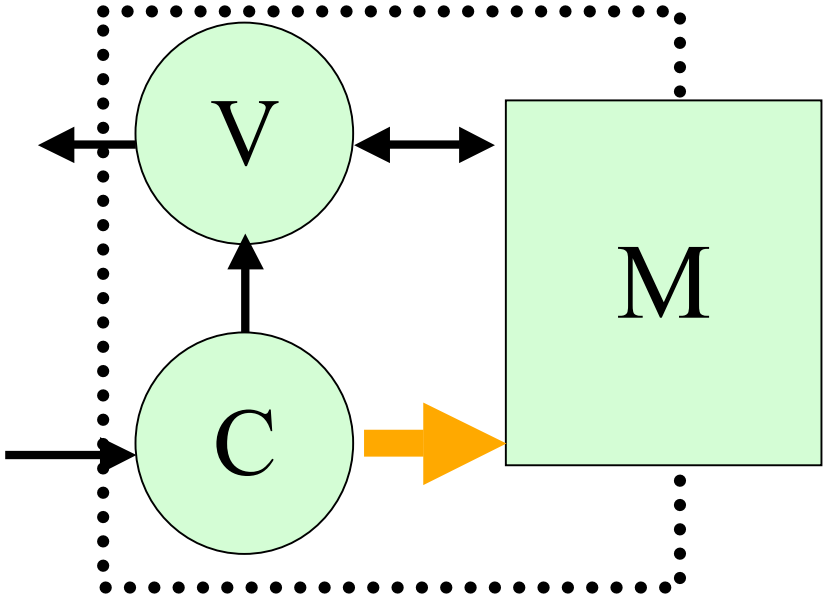
MVC Dynamics

1. User clicks on widget
2. Event gets dispatched to right widget
3. Controller portion handles the event
4. Controller portion may tell View to look selected or pressed



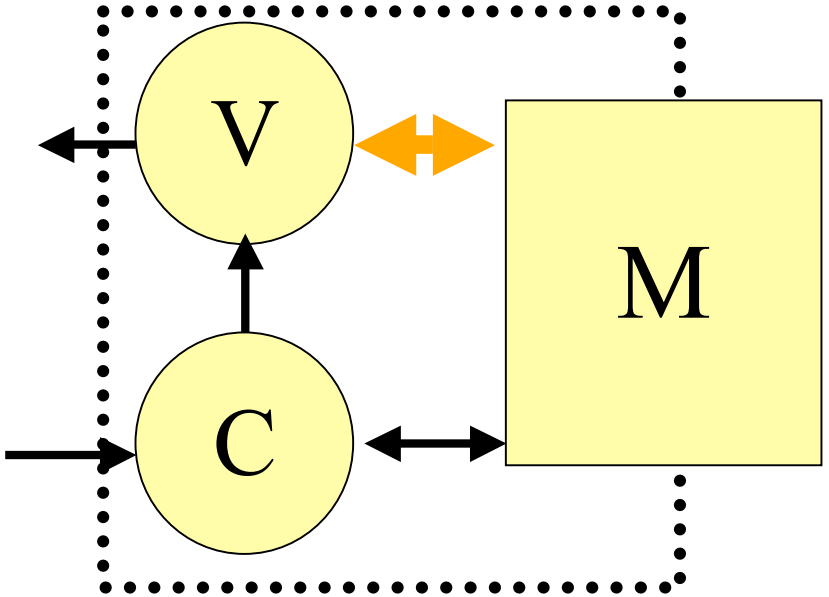
MVC Dynamics

5. Controller figures out what method in Model to call to change its state
6. Model changes its internal state
7. Model generates higher-level events (ex. action performed) and sends to any listeners



MVC Dynamics

8. Model notifies all dependent Views that data has changed
9. View requests from Model current data values (might be part of notification)
10. View requests redraw if needed



View + Controller linking

- In practice, View and Controller implemented together
  - Controller almost always has to “talk to” view
  - Need geometry to interpret input (e.g., picking)
  - Need to do feedback
- As a result, MVC is usually implemented as M-VC

MVC in Java

Component	Model Interface	Model Type
JButton	ButtonModel	GUI
JCheckBox	ButtonModel	GUI/data
JRadioButton	ButtonModel	GUI/data
JComboBox	ComboBoxModel	data
JSlider	BoundedRangeModel	GUI/data
JTextArea/ JTextField	Document	data

Digression

- Model View Controller perhaps most useful design pattern ever
  - Micro-scale, widget level
  - Macro-scale, entire system architecture level
- Other variants of MVC:
  - Separate content from presentation
  - Separate front-end from back-end
  - Separate business logic from everything else

- May seem like more work upfront
  - But keeps you honest, cleaner semantics, cleaner APIs

Two Variations of MVC in Java

- Java peers (AWT)
- Java pluggable looks and feels (Swing)

- Both address same problem:
  - Java is supposed to be cross-platform
  - However, different GUIs have different looks and feels
    - Mac, MS Windows, Motif, GTK, etc

- Design constraints
  - Don’t want to force developers to rewrite apps for platforms
  - Don’t want different APIs for different platforms

Java Peers

- Solution #1:
  - Link Java widget set to native platform
    - Java button peered with native button
    - Java scrollbar peered with native scrollbar
  - Java just provides a thin layer of abstraction
  - Each Java runtime needs to support each peer for each platform

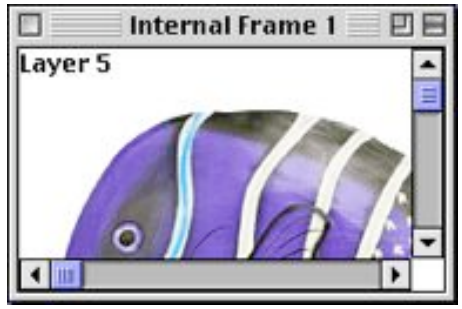
- Implications
  - All rendering, event handling, etc happens at OS level

Java Peers

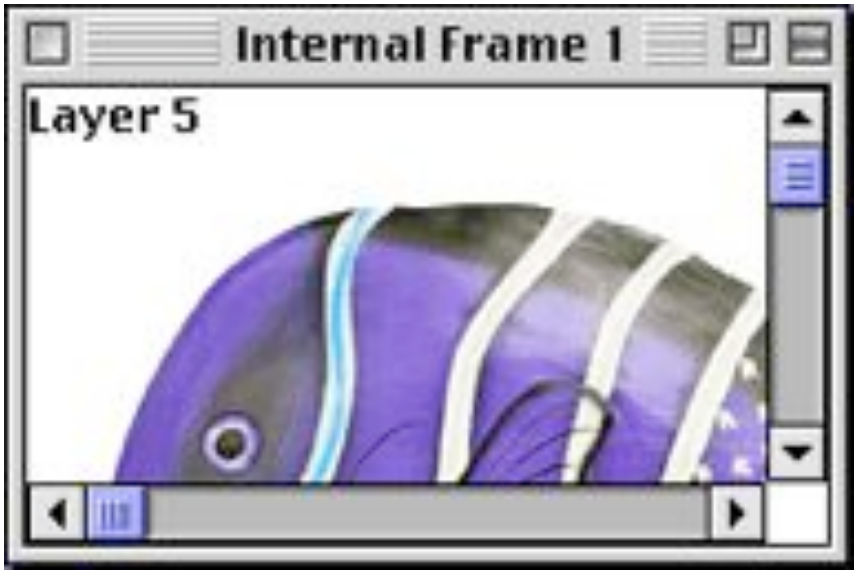
- Advantages:
  - Looks like native platform (because it is!)
  - Fast, took only a few weeks to do
- Weaknesses:
  - Inconsistent looks and feels
    - Developers unsure how app will look and perform
    - Write Once Test Everywhere
  - Source code not as useful (because of reliance on OS)

Java Pluggable Look and Feel

- Solution #2:
  - Developers can use basic widget set provided by Swing
  - Can also change look and feel of widget set as needed
    - Pluggable look and feel
    - Rendered entirely in Java
  - So app happens to look like a “native” app
- Implications
  - All rendering, event handling, etc happens in Java



Java Pluggable Look and Feel



Java Pluggable Look and Feel

- Some PL&Fs with Java
  - Basic, Ocean, Metal, Synth
- Some commercial and open-source ones
  - Alloy and Looks

Java Pluggable Look and Feel

- Advantages
  - Can test on a single machine (just change PL&F)
  - Can get reliable look and feel (all in Java vs OS)
  - Consistent set of widgets (ex. JTable)
  - Easier to do “skinning”
- Weaknesses:
  - Have to create new PL&F on every update
    - Java will always lag somewhat
  - Only useful if new widgets have a PL&F provided

Large Class Exercise

- Suppose you want to “sketchify” your GUI
  - Fun look and feel, or for prototyping
  - **Pros** and **cons** for sketchifying at each layer?
  - Best layer to do this?
- Discuss in groups of 3 for ~8 minutes

Objects
(Widgets, Retained Object Model)

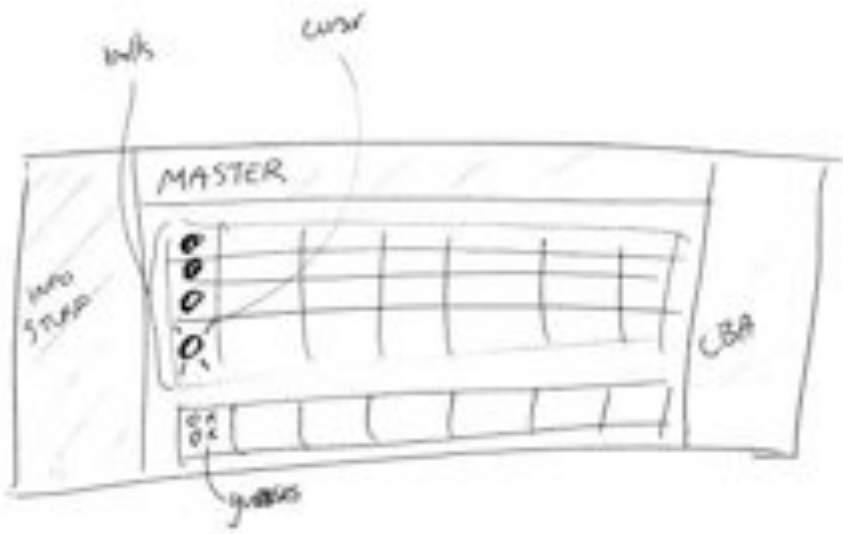
Strokes
(Lines, curves, path models, fonts)

Pixels
(Frame buffer, images)

Large Class Exercise

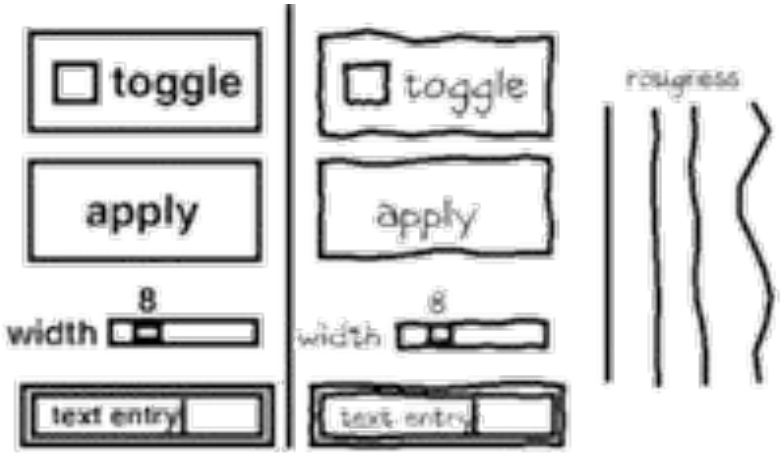
- Object Layer Pros
  - Simple Pluggable Look and Feel, trivial to change

- Object Layer Cons
  - Doesn't work for custom widgets
  - Hard to do text



Large Class Exercise

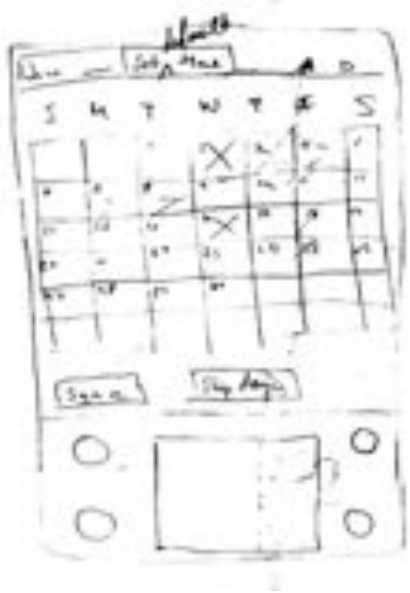
- Stroke Layer Pros
  - Modify Graphics2D to SketchyGraphics2D
  - drawLine() implemented by drawSketchyLine()
  - Works for all things rendered
- Stroke Layer Cons
  - Won't work for images well
  - Text might be hard to read
    - Custom font?



Large Class Exercise

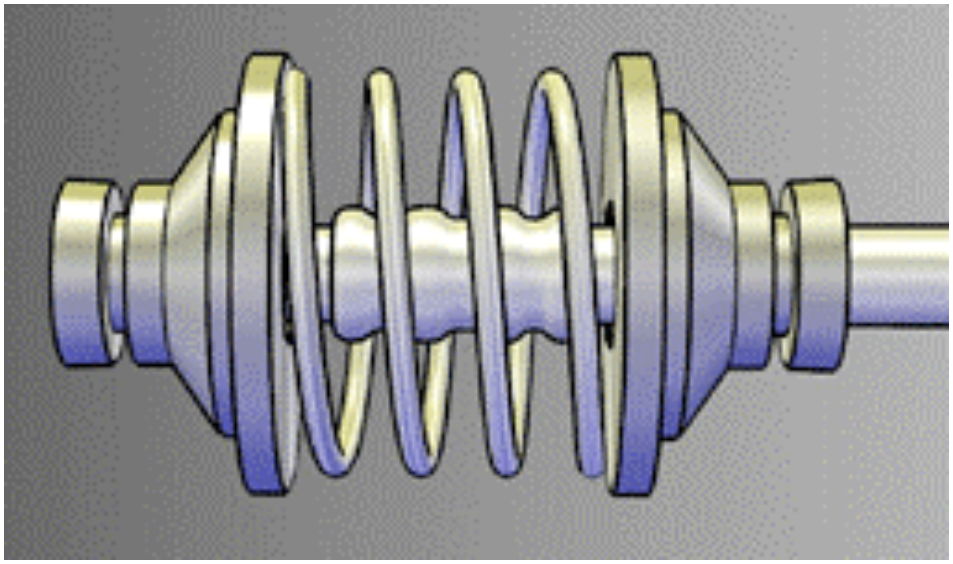
- Pixel Layer Pros
  - No semantics needed
  - Works for all things rendered (including images)

- Pixel Layer Cons
  - Hard to implement (need a good noise function)
  - Hard to make it look good



Non-Photorealistic Rendering (NPR)

- See http://www.red3d.com/cwr/npr/



Non-Photorealistic Rendering (NPR)

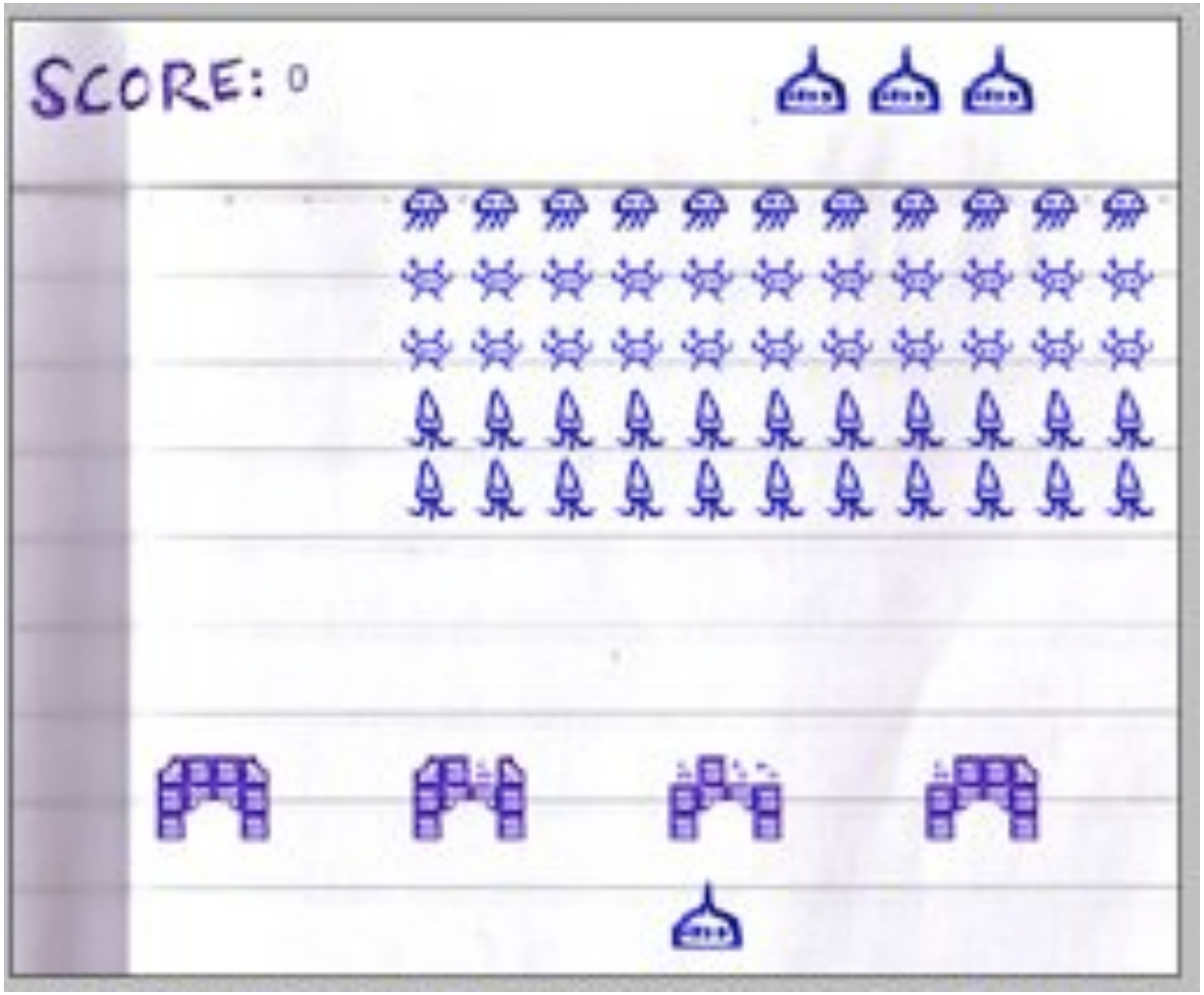
- See http://www.red3d.com/cwr/npr/

Non-Photorealistic Rendering (NPR)

- See http://www.red3d.com/cwr/npr/

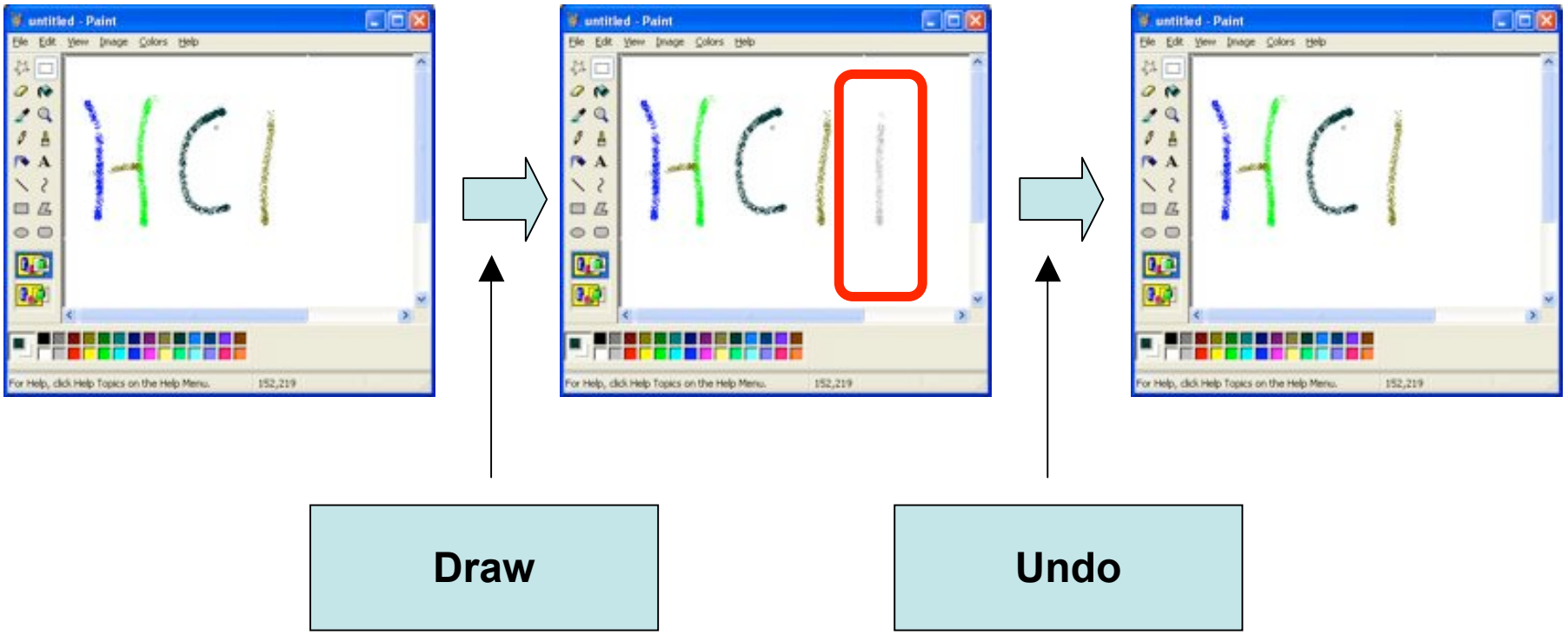
Non-Photorealistic Rendering (NPR)

- Notepad Invaders



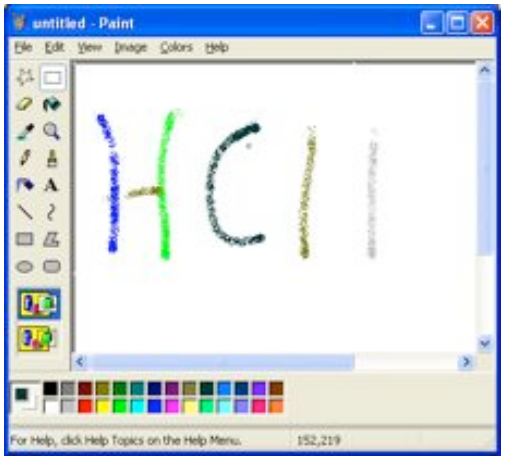
Undo / Redo

- How to support undo and redo in apps?



Approach #1: Save All State

- Save all state as you go
  - Ex. save entire state of canvas on every action
  - To undo, just throw away current canvas and go backwards



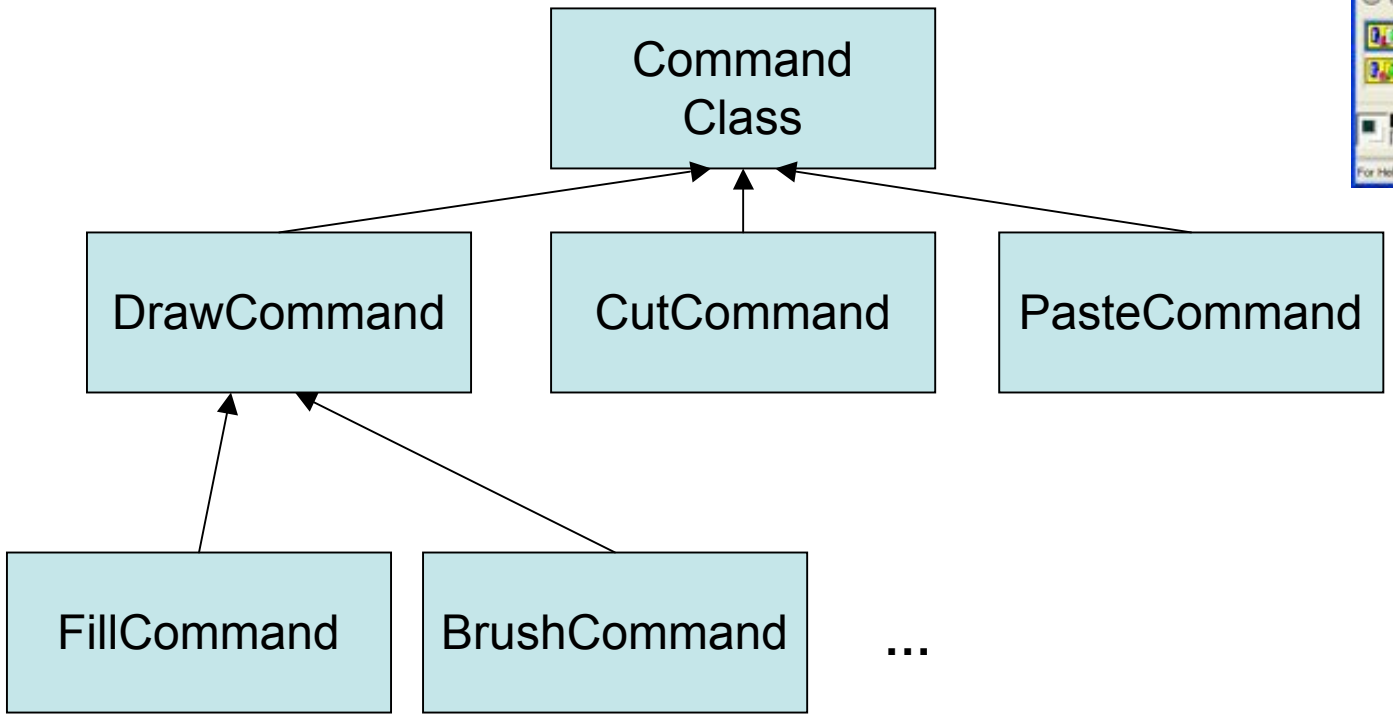
- Pros
  - Relatively easy to implement
- Cons
  - Lots of copying for every action done
  - Lots of memory required

Approach #2: Save Diffs

- Same basic idea as previous slide, but finer grained
  - Rather than saving complete state each time, save diffs

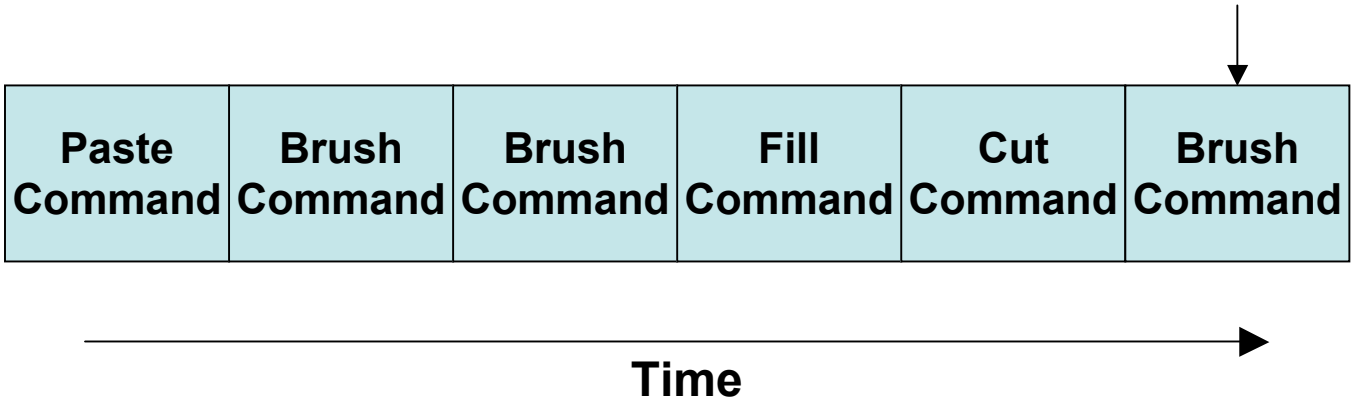
- Command objects
  - Encapsulate all commands as Command objects
  - Two methods:
    - `execute()`
    - `undo()`

Command Objects – Example



Command Queue

- Single queue that contains all Commands executed
  - Methods `undo()` and `redo()`



- To undo, just go backwards
- To redo, just go forwards

Advantages of Command Objects

- Good reuse of code
  - Menu, keyboard shortcuts, GUI widgets can all point to same Command
  - Reduces copy-and-paste of code
  - (Level of indirection)
- Easy to Enable / Disable commands
  - Enable / Disable in one place only (rather than multiple)
  - (Need a way of messaging views to update selves)
- Remote execution
  - Remote client just sends Commands

Advantages of Command Objects

- Macros easy to implement
  - Just have a `MacroCommand` that contains Commands
- Logging
  - Easy to log user actions (and analyze)

Java Swing

- Java does not have explicit Command objects

- **`javax.swing.Action`**
  - Think of it as **`execute()`** without **`undo()`**
  - Is an **`ActionListener`**
  - Can contain listeners (for updates to state)
  - Single point for menus, GUI widgets, etc
  - Easily linked to keyboard short cuts

Java Swing

- Also see **`javax.swing.undo.*`**

- Objects with state implement **`StateEditable`**
  - **`storeState()`** and **`restoreState()`**
- Edits implement **`UndoableEdit`**
  - **`undo()`**, **`redo()`**, **`isSignificant()`**

- **`UndoManager`** stores **`UndoableEdits`**
  - Like the command queue

Java Swing

- A little more structured than standard Command
  - Different programming model, no explicit `execute()`
    - Edits happen “elsewhere” in program
  - `UndoManager` and friends just manage state values and changes rather than executing code

Design Issues for Command Objects

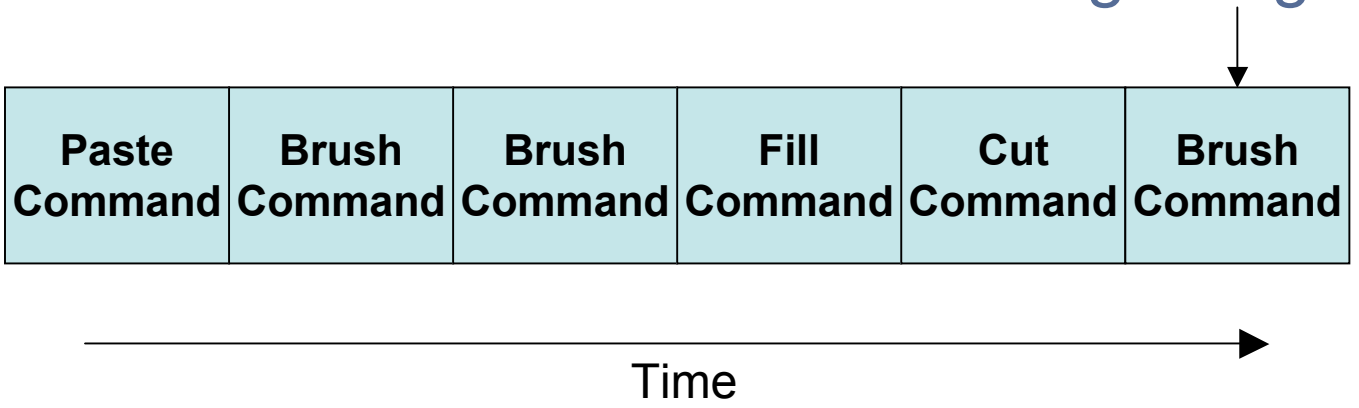
- Requires entire system buy-in
  - All commands have to be implemented this way for it to work
  - Difficult to modify existing system to use command objects
- Hard to get right
  - Have to remember to capture ALL state in `execute()`
  - Have to remember to undo ALL state for `undo()`
  - Forgetting anything can be disastrous
- Undo / Redo size
  - Can't be unlimited
  - Snapshot (ex. on save)

Design Issues for Command Objects

- Granularity of a Command
  - Word processor useless with single char Commands
  - Need time-based approach to coalesce things together
- Can’t easily undo some things
  - Open file, save file, launch rocket

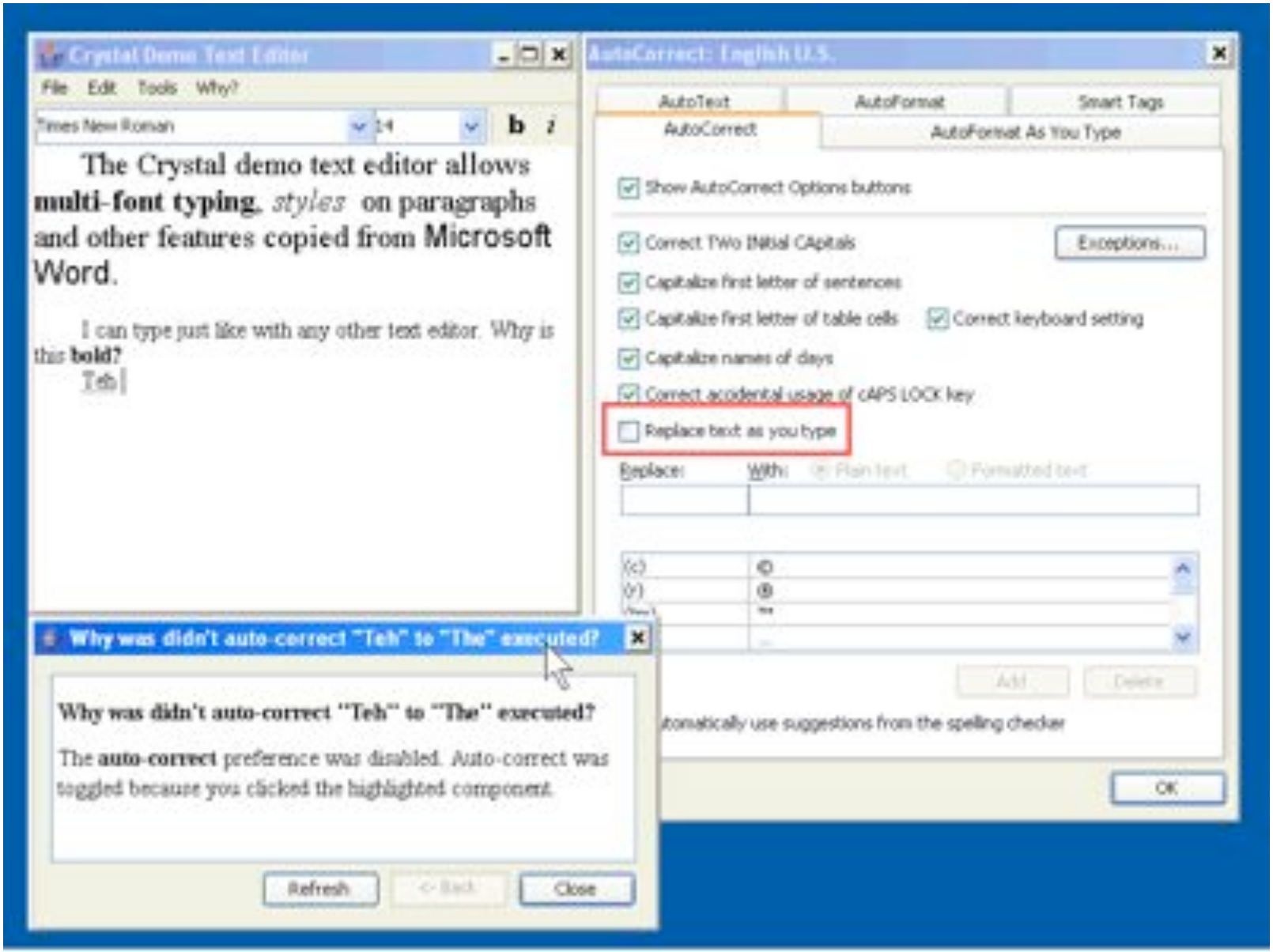
Approach #3: Just Redo it All

- Just redo all commands from the beginning



- Easy to implement
  - Only really need `execute()`, not `undo()`
  - Simpler code base (again, undo can be hard to get right)
  - Periodically snapshot state and save to disk (like databases)

Novel Use of Command Objects



Novel Use of Command Objects

- Answering Why and Why Not Questions in User Interfaces

- Uses Command objects to tell what happened recently where
  - Some “why” and “why not” questions generated from these
- Extensions to Commands
  - Dependencies (ex. Properties like Auto-correct)

Summary

- Input Models
  - Higher level events
  - Dispatch
- Object Level Architectures
  - Model View Controller
  - Pluggable Look and Feel
  - Undo / Redo