MiniDraw

Introducing a Framework

... and a few patterns
What is it?

[Demo]
MiniDraw helps you building apps that have

- 2D image based graphics
  - GIF files
  - Optimized repainting
- Direct manipulation
  - Manipulate objects directly using the mouse
- Semantic constraints
  - Keep objects semantically linked
MiniDraw is downsized from JHotDraw

JHotDraw
- Thomas Eggenschwiler and Erich Gamma
- Java version of HotDraw

HotDraw
- Kent Beck and Ward Cunningham.
- Part of a smalltalk research project that lead to the ideas we now call design patterns and frameworks
MiniDraw

- supporting board games mainly
- cut down detail for teaching purposes
- one day convert to C# (hmm ?)
- MiniDraw: compositional design (most of the time)
- JHotDraw: polymorphic design (quite a lot of the time)

Newest addition

- BoardGame extension:
  * High support for board games
Our first MiniDraw application
public class LogoPuzzle {

    public static void main(String[] args) {
        DrawingEditor editor =
            new MiniDrawApplication("Put the pieces into place",
                new PuzzleFactory()
            );
        editor.open();
        editor.setTool(new SelectionTool(editor));

        Drawing drawing = editor.drawing();
        drawing.add(new ImageFigure("11", new Point(5, 5)));
        drawing.add(new ImageFigure("12", new Point(10, 10)));
        drawing.add(new ImageFigure("13", new Point(15, 15)));
        drawing.add(new ImageFigure("21", new Point(20, 20)));
        drawing.add(new ImageFigure("22", new Point(25, 25)));
        drawing.add(new ImageFigure("23", new Point(30, 30)));
        drawing.add(new ImageFigure("31", new Point(35, 35)));
        drawing.add(new ImageFigure("32", new Point(40, 40)));
        drawing.add(new ImageFigure("33", new Point(45, 45)));
    }

    class PuzzleFactory implements Factory {

        public DrawingView createDrawingView(DrawingEditor editor) {
            DrawingView view =
                new StdViewWithBackground(editor, "au-seal-large");
            return view;
        }

        public Drawing createDrawing(DrawingEditor editor) {
            return new StandardDrawing();
        }

        public JTextField createStatusField(DrawingEditor editor) {
            return null;
        }
    }
}
The Patterns in MiniDraw

Not *what* but *why*?

The 3-1-2 principles in action again...
MiniDraw software architecture

Main JHotDraw architecture remains

- **Model-View-Controller** architectural pattern
  - Drawing-DrawingView-Tool

- **Observer** pattern event mechanism
MVC’ problem statement

Challenge:

- writing programs with a graphical user interface
- multiple open windows showing the same data – keeping them consistent
- manipulating data in many different ways by direct manipulation (eg. move, resize, delete, create, ...)
  - i.e. switching tool will switch the object manipulation
Keeping multiple windows consistent?

Analysis:

- **Data** is shared but **visualization** is variable!

  ① Data **visualisation** is variable behaviour
  ② Responsibility to visualize data is expressed in interface: View
  ③ Instead of data object (model) itself is responsible for drawing graphics it *lets someone else do the job*: the views
Few mouse events (down, up, drag) translate to open-ended number of actions (move, resize, create, ?) on data.

- Events are the same but manipulation is variable

③ Data manipulation is variable behaviour

① Responsibility to manipulate data is expressed in interface: Controller

② Instead of graphical view itself is responsible for manipulating data it lets someone else do the job: the controller
Challenge 1:

– Also known as **observer pattern**

**Intent**

– Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Observer: Structure

Structure:

Subject

- addObserver(Observer)
- removeObserver(Observer)
- setState(newState)
- getState()
- notifyObservers()

for all o : observers {
    o.update();
}

interface Observer

- update()

ConcreteObserver
Observer Protocol

Protocol:

*A convention detailing the expected sequence of interactions or actions expected by a set of roles.*
Observer

Benefits

• open ended number of viewer types (run-time binding)
• need not be known at develop time
  – change by addition, not by modification...
• any number of views open at the same time when executing
• all guarantied to be synchronized

Liabilities

• update sequence can become cyclic or costly to maintain
Challenge 2:
- Also known as *state pattern*

**Intent**

- *Allow an object to alter its behaviour when its internal state changes. The object will appear to change its class.*

- i.e. when the editor is in “draw rectangle” state, the mouse events (click, drag, release) will create a rectangle; when in “select object” state, the same (click, drag, release) will move an object…
Consequences

- the manipulation that is active determine the application state ("am I moving or resizing figures?")
- open ended number of manipulations (run-time binding)
- need not know all states at compile time
  - change by addition...

```java
state.request();
```
Selected Tool defines the State

In MiniDraw (HotDraw) the editor is in a state that determines how mouse events are interpreted – do they move a checker, do they select a set of figures, or do they create a rectangle?

Mouse events are forwarded to the editor’s tool. *By changing the tool I change how mouse events are interpreted.*
The MVC is an architectural pattern because it defines a solution to the problem of structuring the ’large-scale’ / architectural challenge of building graphical user interface applications. But the ’engine behind the scene’ is a careful combination of state and observer...

That again are example of using the 3-1-2 variability handling process.
Figure 29.2: MVC role structure.
Responsibilities

Model
- Store application state.
- Maintain the set of Views associated.
- Notify all views in case of state changes.

View
- Visualize model state graphically.
- Accept user input events, delegate them to the associated Controller.
- Potentially manage a set of controllers and allow the user to set which controller is active.

Controller
- Interpret user input events and translate them into state changes in the Model.
Dynamics

c: ConcreteController

v: ConcreteView

m: ConcreteModel

handleEvent() → event() → changeState() → notify

→ redraw()

→ update() → getState()
MiniDraw
MiniDraw: Static+Pattern View

Observer 1: Observer
MVC: View
«interface» DrawingView

Observer 1: Subject
Observer 2: Observer
MVC: Model

State: Context
«interface» DrawingEditor

State: State
MVC: Controller
«interface» Tool

current

1

«interface» Drawing
MVC: Model

1

* manipulate

«interface» Figure
MVC: Model
Observer 2: Subject
Tool: The Controller role
MiniDraw: Tool Interaction

Basic paradigm: *Direct Manipulation*

*[Demo: puzzle]*
View -> Controller interaction

Mouse events *do* hit the JPanel, but MiniDraw simply delegate to its active tool...
Example Tool

Scenario: *User drags image figure around.* Then a *DragTracker* is the active tool:
The real story

… is somewhat more complex as it involves a bit more delegation 😊

StandardDrawingView is-a JPanel.
– This view requests access to the editor’s current tool

```java
/**
 * Handles mouse down events. The event is delegated to the
 * currently active tool.
 */
public void mousePressed(MouseEvent e) {
    requestFocus();
    Point p = constrainPoint(new Point(e.getX(), e.getY()));
    fLastClick = new Point(e.getX(), e.getY());
    editor.tool().mouseDown(e, p.x, p.y);
    checkDamage();
}
```
MiniDraw has some simple tools defined
It is very simple to set a new tool:

```java
tool.
```

where \( t \) is the tool you want to become active.

NullTool is a *Null Object*: a tool that does nothing.
Drawing: The Model role
MiniDraw: Drawing

Static view

- `DrawingChangeEventListener`
- `FigureChangeListener`
- `SelectionHandler`
- `DrawingChangeEvent`
- `ImageFigure`
- `AbstractFigure`
- `Figure`

Diagram showing the relationships between these classes and events.
But how does the view get repainted?

- Double observer chain
  - Figure *notifies* drawing *notifies* drawing view.

![Diagram showing the double observer chain](image-url)
Exercise:

Observer pattern has two roles
- Subject: Container of data
- Observer: Object to notify upon data changes

Who are who here???
DrawingView: The View role
The View is rather simple

- JPanel to couple MiniDraw to concrete Swing GUI implementation
- Listen to mouse events to forward them to tool/controller.
The Compositional Advantage

Note that this design combines two frameworks

- MiniDraw and Swing
- If DrawingView was *not* an interface then 🙁
DrawingEditor: The Coordinator
**Static View**

**DrawingEditor**
- Main class of a minidraw application, that is the editor must instantiate all parts of the application.
- Opens a window to make a visible application.
- Acts as central access point for the various parts of MiniDraw.
- Allows changing the active tool.
- Allows displaying a message in the status bar.
Implementation
Default Implementations

Most MiniDraw roles have default implementations:
- Interface X has default implementation StandardX
- Drawing -> StandardDrawing

There are also some partial implementations:
- Interface X has partial implementation AbstractX
- Tool -> AbstractTool
- Figure -> AbstractFigure
Complex behaviour as a result of combining simple behaviour...

Example:
- StandardDrawing
- Responsibilities

<table>
<thead>
<tr>
<th>Drawing</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Be a collection of figures.</td>
</tr>
<tr>
<td>● Allow figures to be added and removed.</td>
</tr>
<tr>
<td>● Maintain a temporary, possibly empty, subset of all figures, called a selection.</td>
</tr>
<tr>
<td>● Broadcast DrawingChangeEvent to all registered DrawingChangeListener when any modification of the drawing happens.</td>
</tr>
</tbody>
</table>
How do we do that?

Proposal 1:
– *implement ahead...*

Proposal 2:
– *encapsulate major responsibilities in separate objects and compose behaviour*

Refactoring pending!
Code view: delegations!

Examples:

```java
/**
 * Adds a listener for this drawing.
 */
public void addDrawingChangeListener(DrawingChangeListener listener) {
    listenerHandler.addDrawingChangeListener(listener);
}

[...]
/**
 * Get a list of all selected figures
 */
public List<Figure> selection() {
    return selectionHandler.selection();
}
```
What do I achieve?

Implementing the HotCiv UnitDrawing

– The figure collection must behave radically different from what the StandardDrawing does

– but I can reuse the selection and drawing-change handler behaviour directly!

```java
public class UnitDrawing implements Drawing, GameObserver {
    /** list of all figures currently selected */
    protected SelectionHandler selectionHandler;

    /** use a StandardDrawingChangeListener to handle all observer pattern subject role behavior */
    protected StandardDrawingChangeListener listenerHandler;

    @Override
    public void addToSelection(Figure arg0) {
        selectionHandler.addToSelection(arg0);
    }

    @Override
    public void clearSelection() {
        selectionHandler.clearSelection();
    }

    @Override
    public void removeFromSelection(Figure arg0) {
        selectionHandler.removeFromSelection(arg0);
    }
}
```
MiniDraw Variability Points
Variability Points

Images
- By putting GIF images in the right folder and use them through ImageFigures

Tools
- Implement Tool and invoke editor.setTool(t)

Figures
- You may make any new type you wish

Drawing
- Own collection of figures (e.g. observe a game instance)

Observer Figure changes
- Make semantic constraints

Views
- Special purpose rendering
MiniDraw is

- A *framework*: A skeleton application that can be tailored for a specific purpose

- A *demonstration*:
  - of MVC, Observer, State, Abstract Factory, Null Object, Strategy, ...
  - of compositional design: *Make complex behaviour by combining simpler behaviours*

- A *basis*: for the mandatory project GUI.