AARHUS UNIVERSITET

Software Engineering and Architecture

MiniDraw

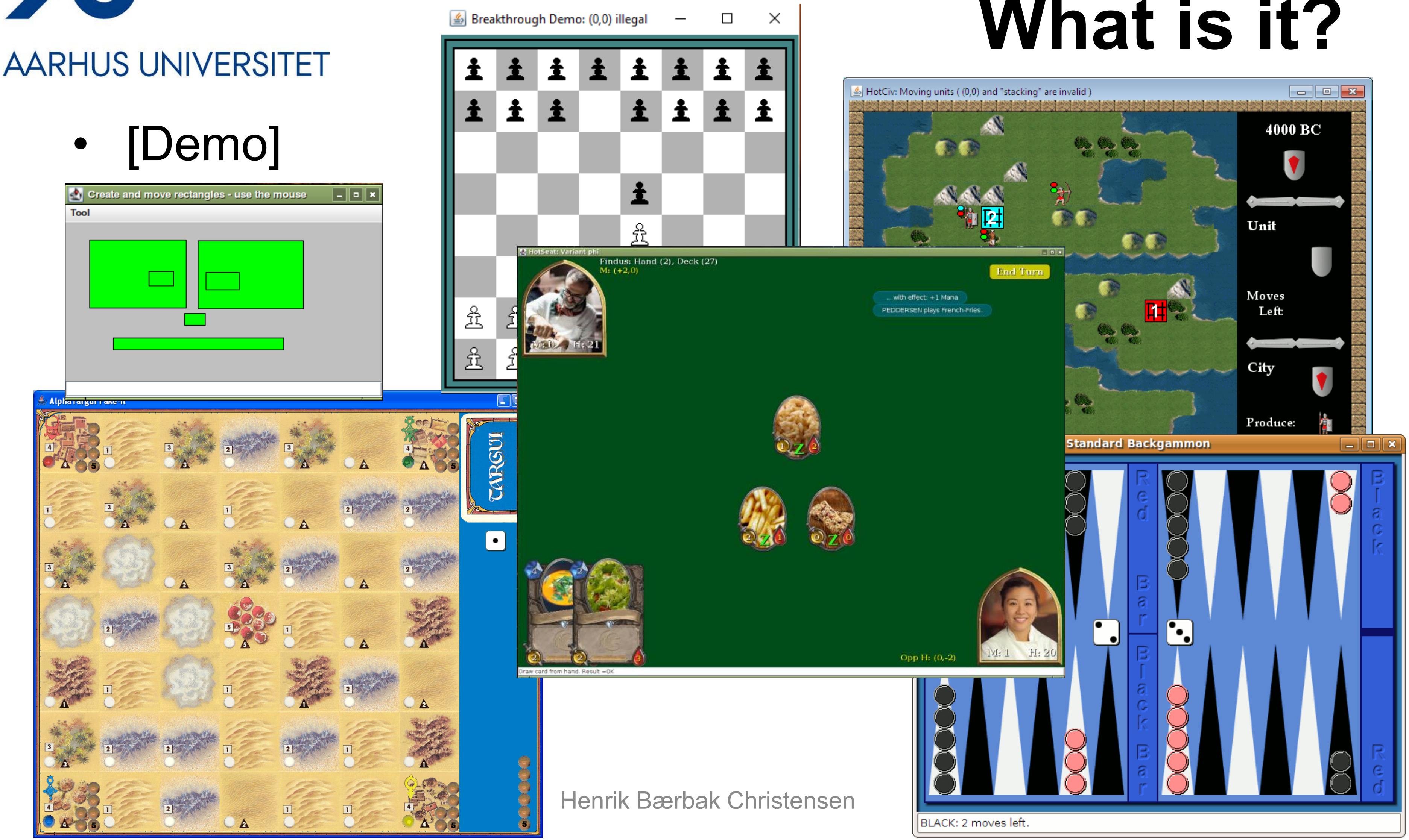
A Framework Example

and lots of patterns

AARHUS UNIVERSITET

What is it?

- [Demo]



Breakthrough Demo: (0,0) illegal

Create and move rectangles - use the mouse

HotCiv: Moving units ((0,0) and "stacking" are invalid)

HotSeat: Variant phi

Findus: Hand (2), Deck (27)
M: (+2,0)

... with effect: +1 Mana

PEDDERSEN plays French-Fries.

Opp H: (0,-2)

Draw card from hand. Result =OK

Standard Backgammon

BLACK: 2 moves left.

Henrik Bærbak Christensen

What do I get?

- MiniDraw is a **framework** that helps you building apps that have
  - 2D image based graphics
    - GIF/JPG files
    - Optimized repainting ☺
  - Direct manipulation
    - Manipulate objects directly using the mouse
  - Semantic constraints
    - Keep objects semantically linked

Focus: 2D graphical systems like board game GUI's.

History

- MiniDraw is downsized from JHotDraw

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

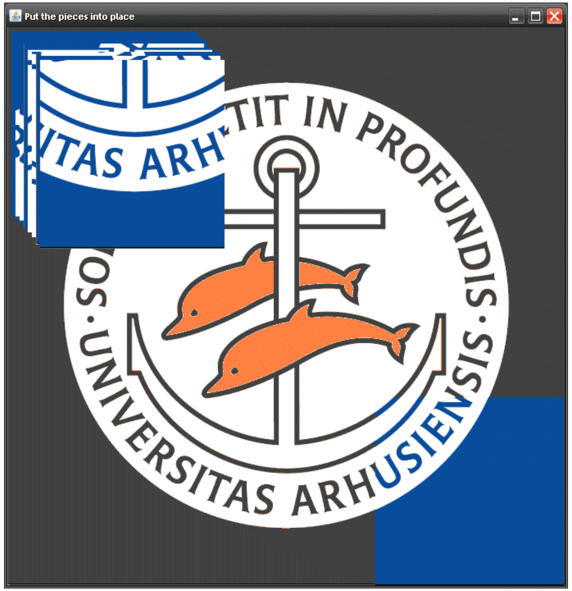
- HotDraw
  - Kent Beck and Ward Cunningham.
  - Part of a smalltalk research project that lead to the ideas we now call *design patterns* and *frameworks*

MiniDraw 3.0

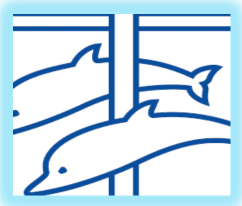
- I did extensive rewriting of MiniDraw Spring 2022.
  - HotStone required more elaborate control of
    - Z-ordering
    - Concurrency (supporting ‘poor-man-animation’ using threads)
  - And the architecture was rewritten
    - Much more compositional approach than in JHotDraw
- Interesting observation
  - ***There were numerous tricky bugs that had never been exposed during the last 15 years !***
    - Anti-Composition Axiom: System testing != Unit testing

AARHUS UNIVERSITET

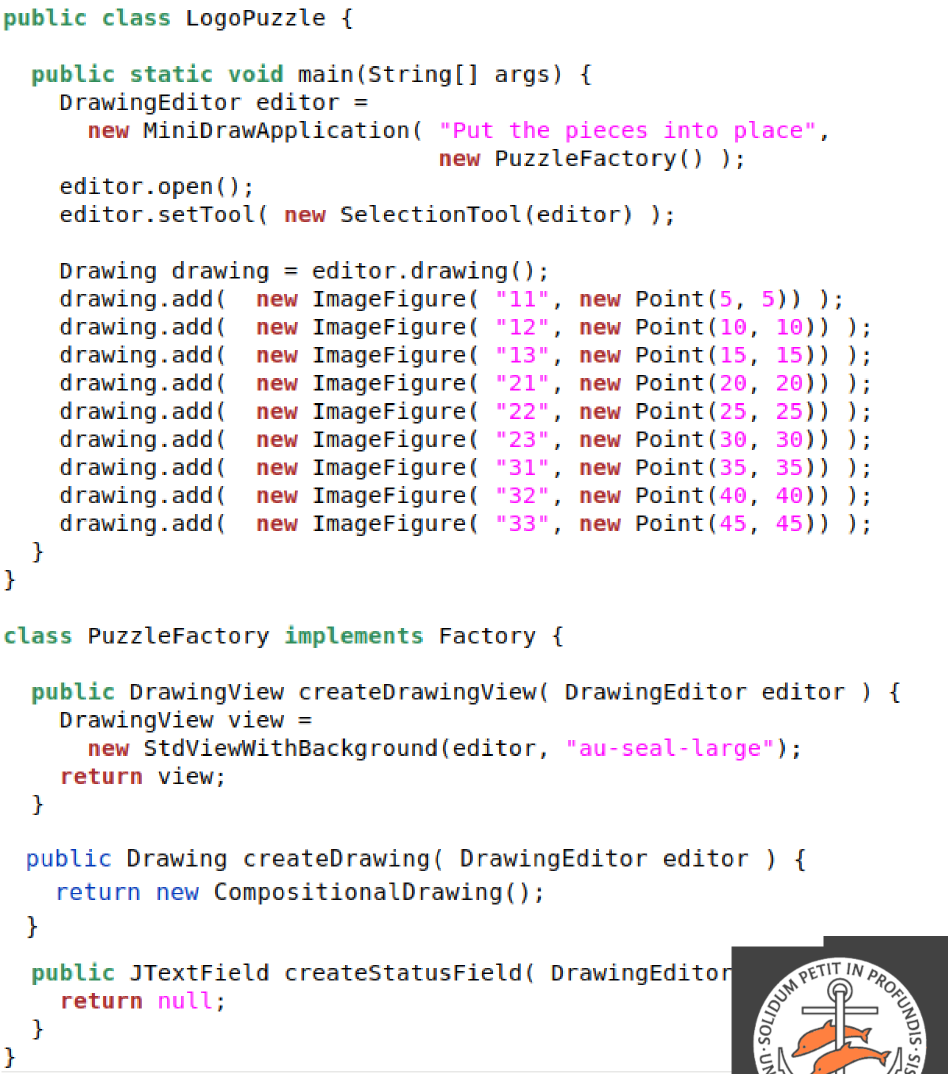
Our first MiniDraw application



- DrawingEditor
  - “Project manager”/Redaktør
  - Default implementation
- Figure
  - Visible element
  - ImageFigure



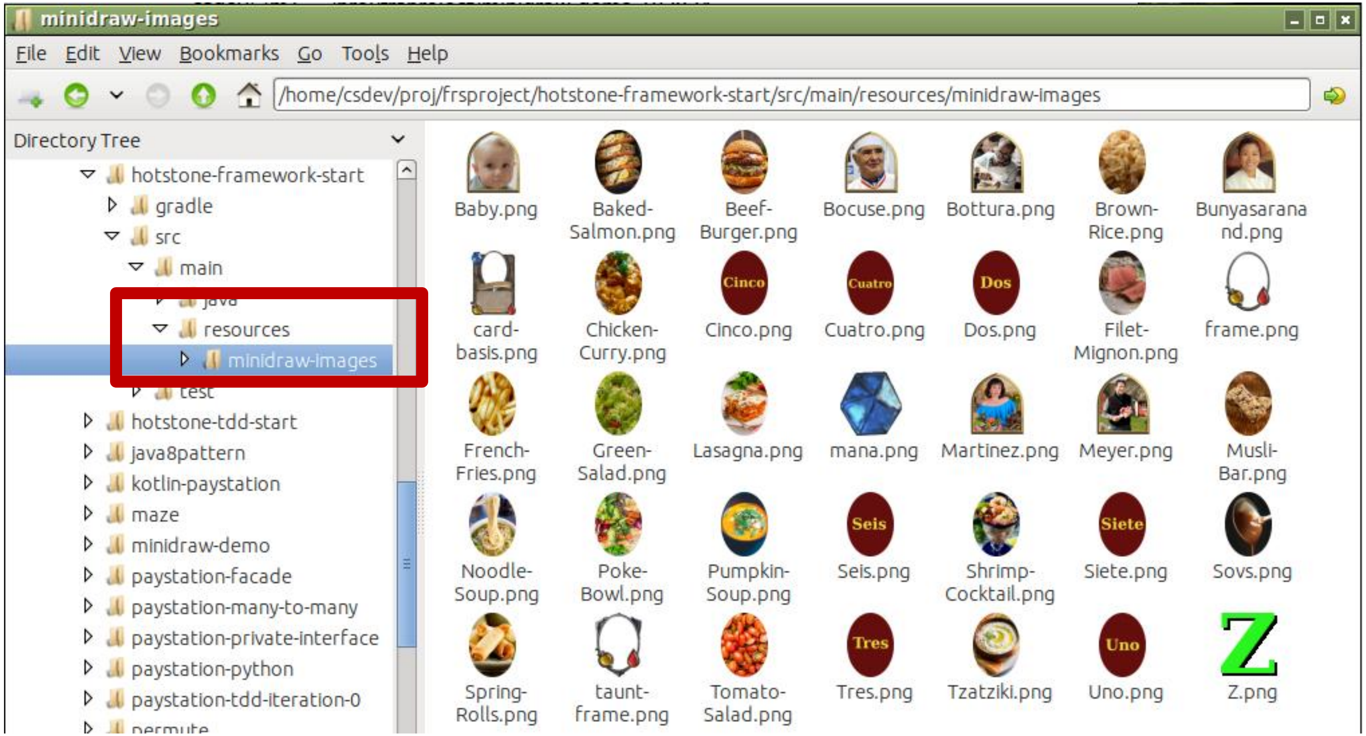
- Drawing
  - container of figures
- Tool
  - = controller
- Factory
  - create impl. of MiniDraw roles
- DrawingView
  - view type to use...



```java
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 CompositionalDrawing();
  }

  public JTextField createStatusField( DrawingEditor
    return null;
  }
}
```

Convention Over Configuration

- How does MiniDraw know about images?
  - Gradle ‘resources’ folder must have ‘minidraw-images’ folder



AARHUS UNIVERSITET

The Patterns in MiniDraw

Not *what* but *why*?

The 3-1-2 principles in action again...

MiniDraw’s Software Architecture

- Main JHotDraw architecture remains
  - **Model-View-Controller (MVC)** architectural pattern
    - In Minidraw:

– Model =	Drawing
– View =	DrawingView
– Controller =	Tool

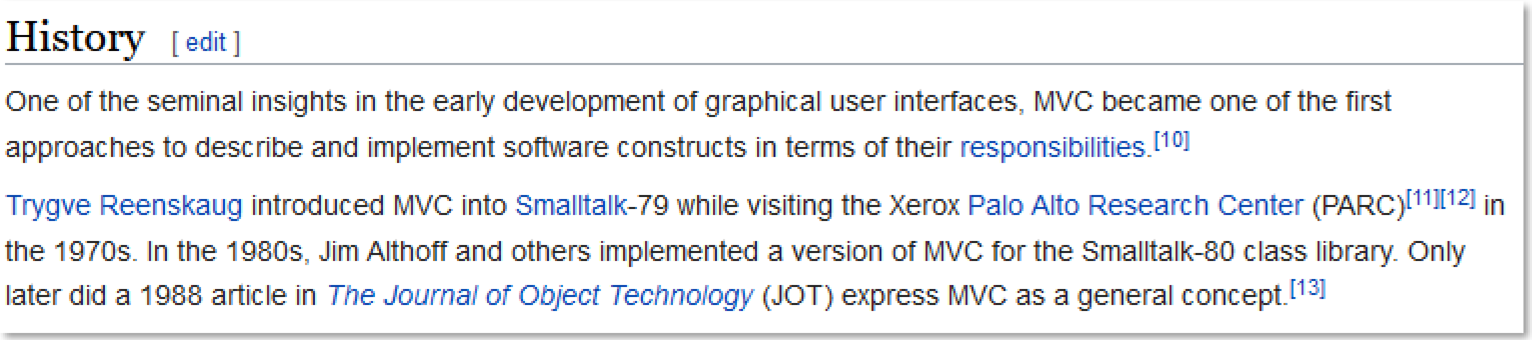
  - And a central ‘sub-pattern’ in MVC is
  - **Observer** pattern event mechanism

MiniDraw software architecture

- All 2D GUI systems I know of, use these!
  - **Model-View-Controller** architectural pattern
    - Java Swing
    - Android UI
  - **Observer** pattern event mechanism
    - Java Swing
    - Android UI (and ‘managers’)
    - Every window-based operating system (Windows/Mac/Ubuntu)
      - Programs *react* to mouse events emitted by window manager

MVC’ problem statement

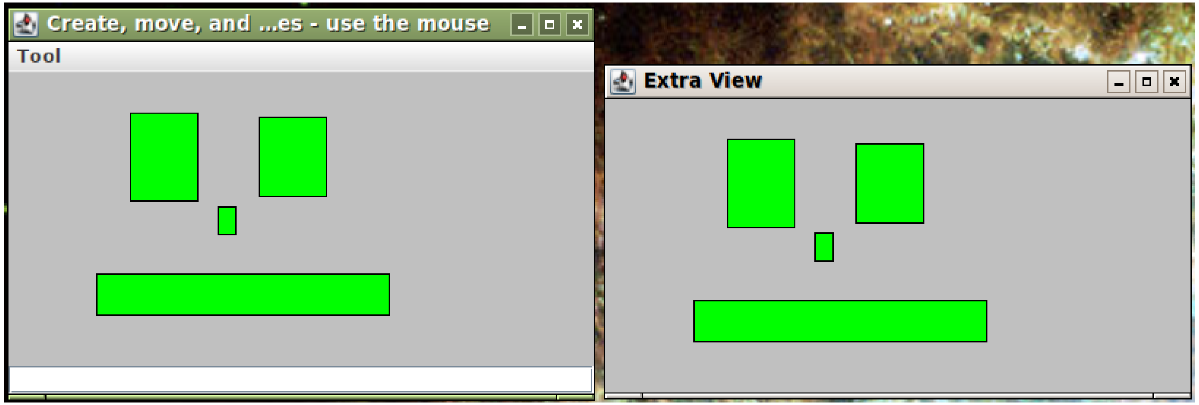
- Challenge:
  - *writing programs with a graphical user interface*



History [edit]

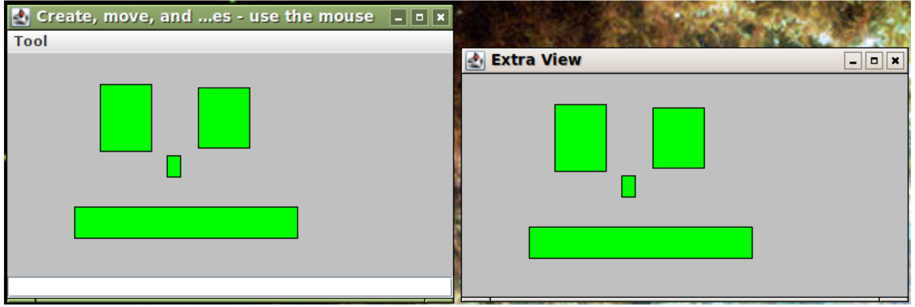
One of the seminal insights in the early development of graphical user interfaces, MVC became one of the first approaches to describe and implement software constructs in terms of their responsibilities.[10]

Trygve Reenskaug introduced MVC into Smalltalk-79 while visiting the Xerox Palo Alto Research Center (PARC)[11][12] in the 1970s. In the 1980s, Jim Althoff and others implemented a version of MVC for the Smalltalk-80 class library. Only later did a 1988 article in *The Journal of Object Technology* (JOT) express MVC as a general concept.[13]



MVC' problem statement

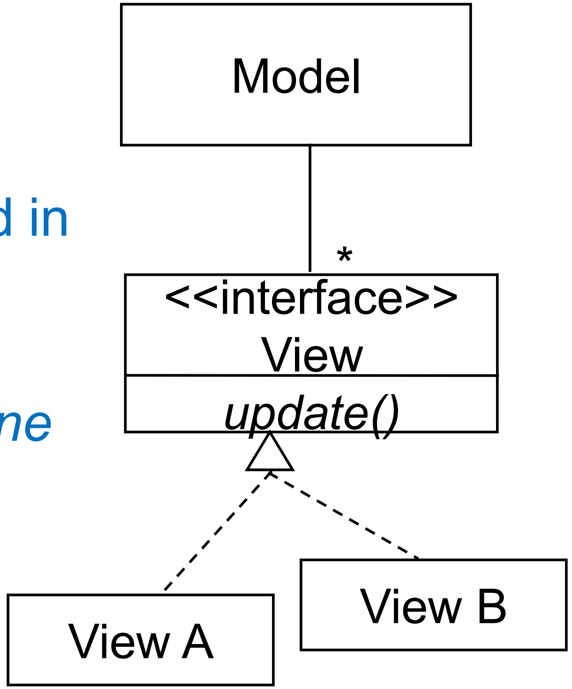
- Challenge:
  - *writing programs with a graphical user interface*
  - 1) multiple open windows showing the same data – keeping them consistent



  - 2) manipulating data in many different ways by direct manipulation (eg. move, resize, delete, create, ...)
    - i.e. switching *tool* will switch the object manipulation

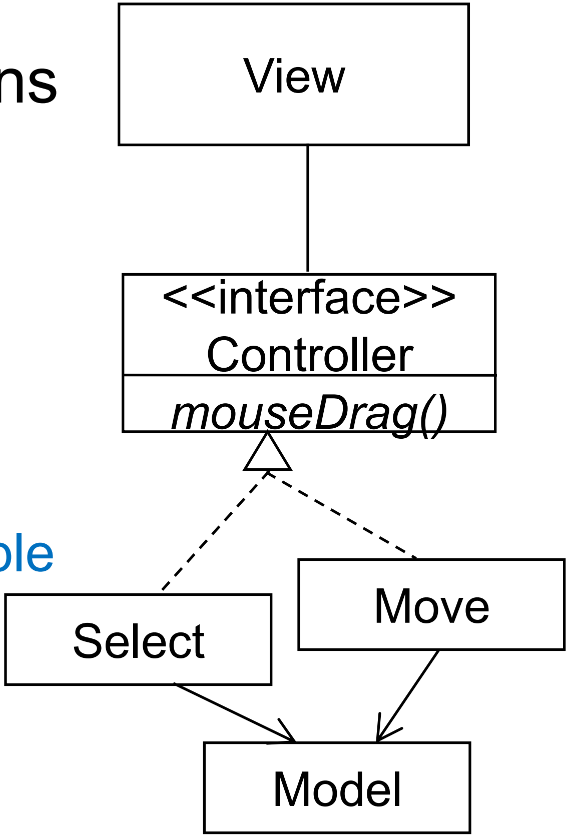
Challenge 1

- Keeping multiple windows consistent?
- Analysis:
  - **Data** is shared but **visualization** is variable!
  - ③ Data **visualization** is variable behavior
  - ① Responsibility to visualize data is expressed in interface: *View*
  - ② Instead of data object (model) itself is responsible for drawing graphics it *lets someone else do the job: the views*



Challenge 2

- 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 behavior
  - 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*



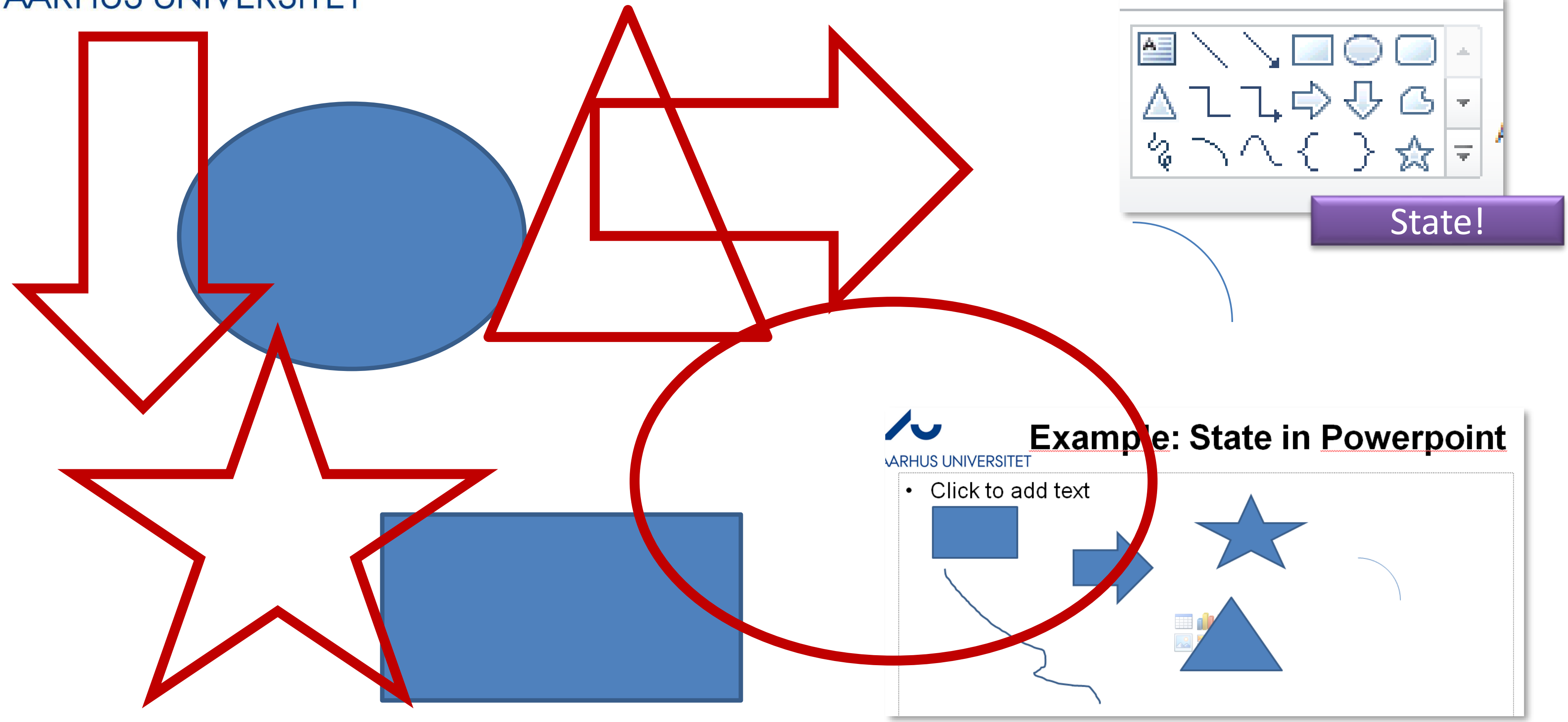
Observer

- 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*.
- *We covered Observer in Week 7* ☺

State

- Challenge 2:
  - Also known as ***state*** *pattern*
- *Intent*
  - *Allow an object to alter its behavior 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…

Example: State in Powerpoint



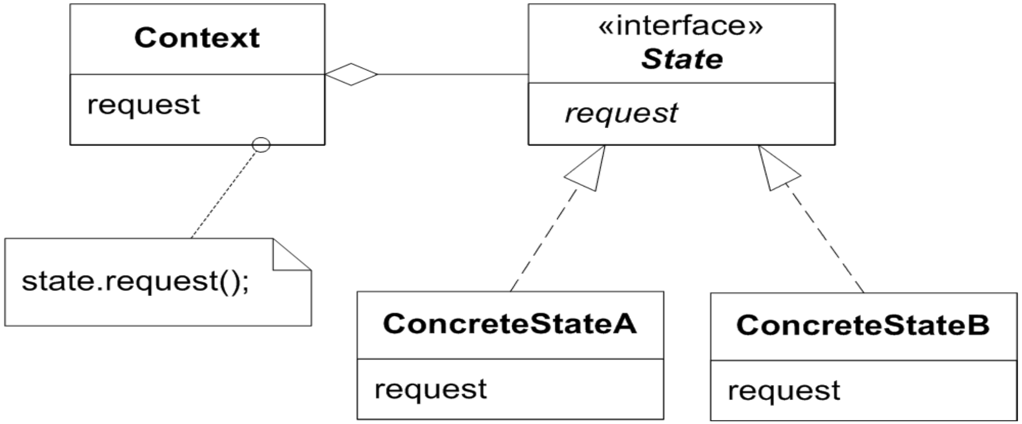
AARHUS UNIVERSITET

Example: State in Powerpoint

- Click to add text

State

- Consequences
  - the manipulation that is active, determines 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...



Architectural Pattern: MVC

- 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.

Static view

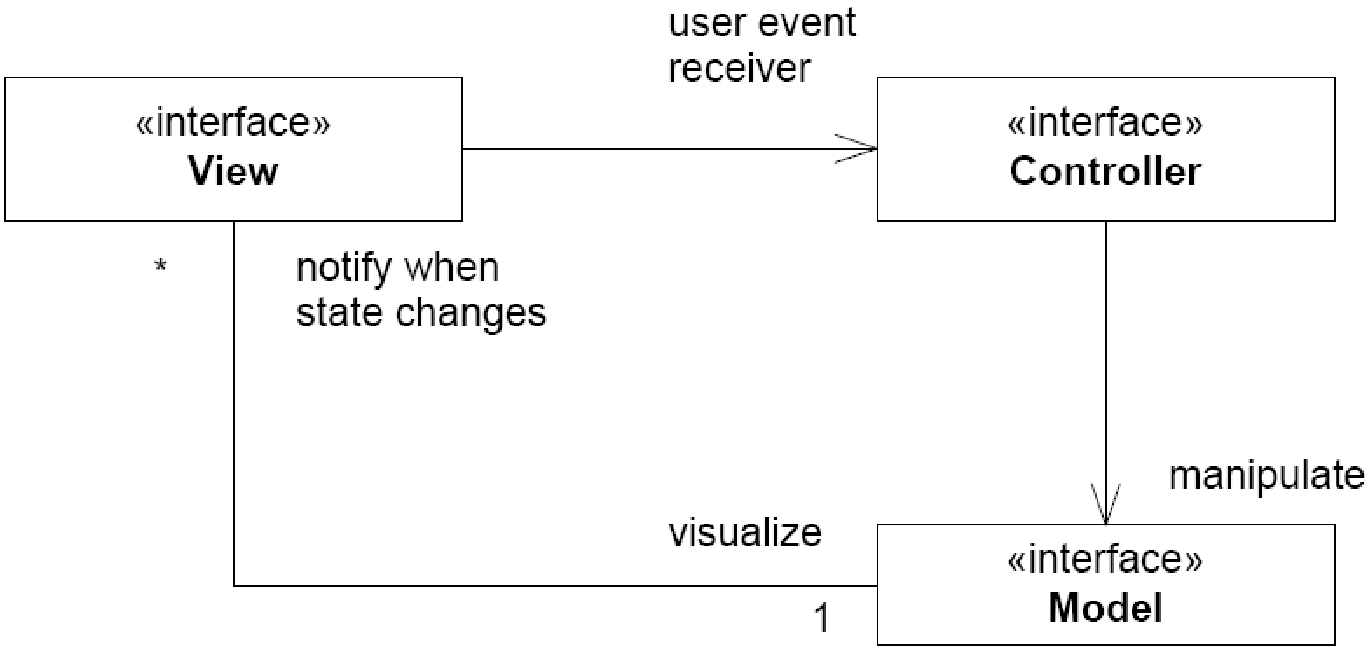
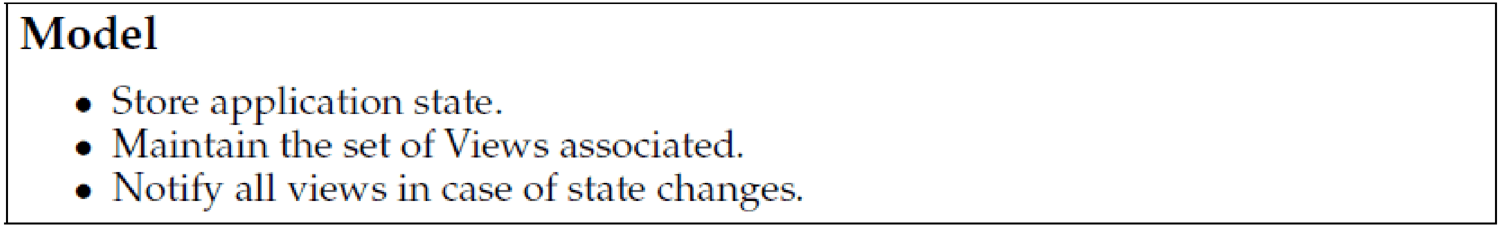


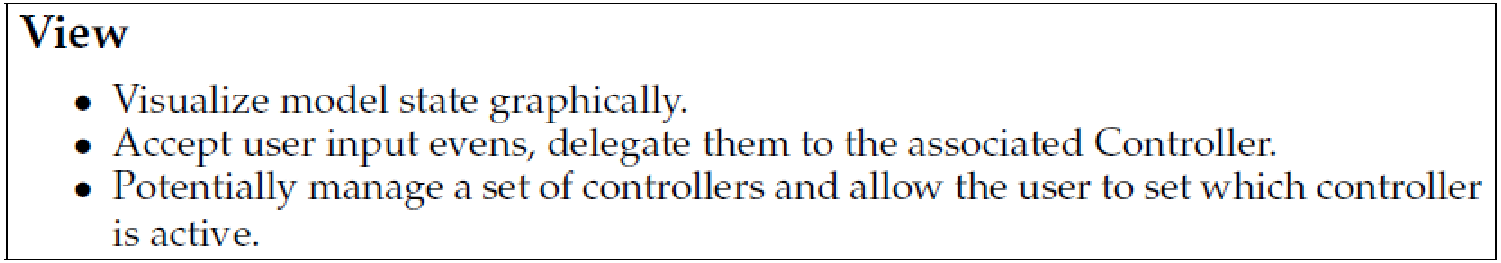
Figure 29.2: MVC role structure.

Responsibilities



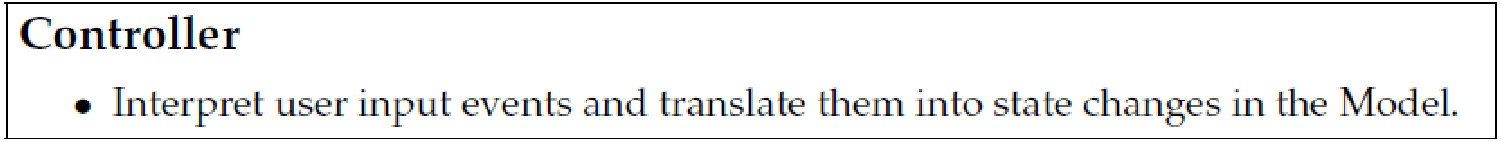
Model

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



View

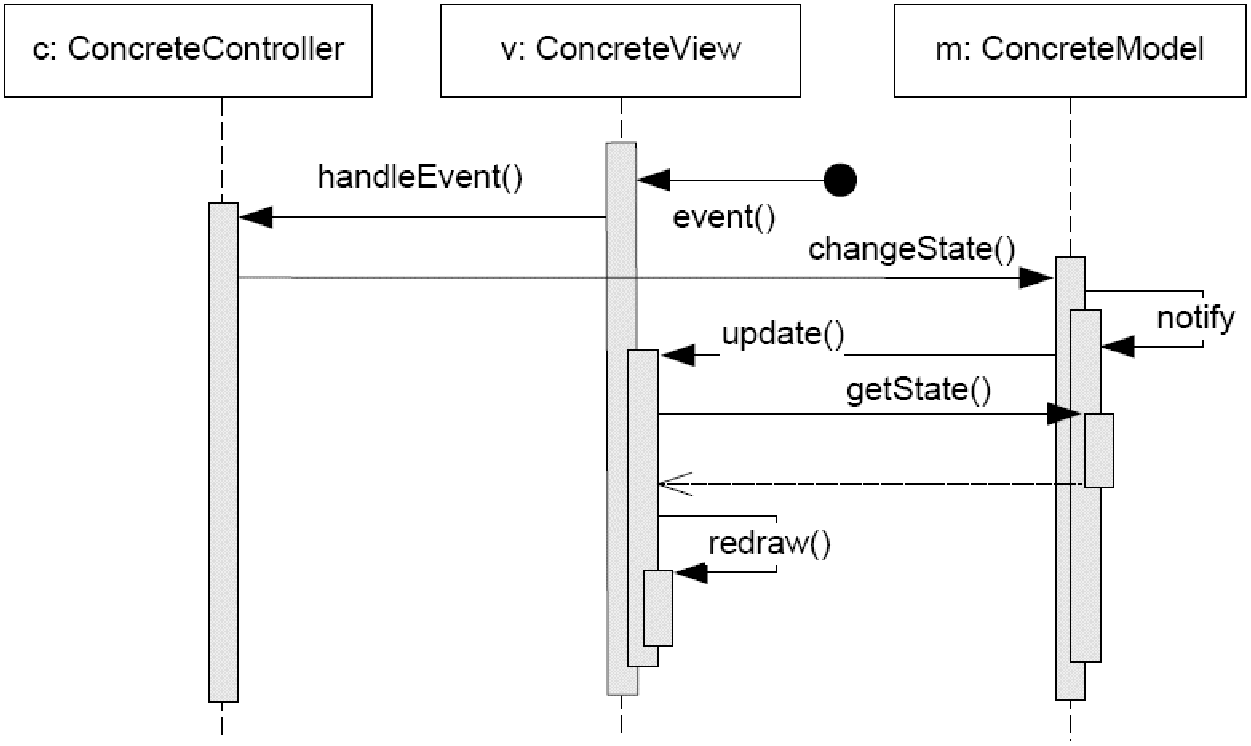
- Visualize model state graphically.
- Accept user input evens, delegate them to the associated Controller.
- Potentially manage a set of controllers and allow the user to set which controller is active.



Controller

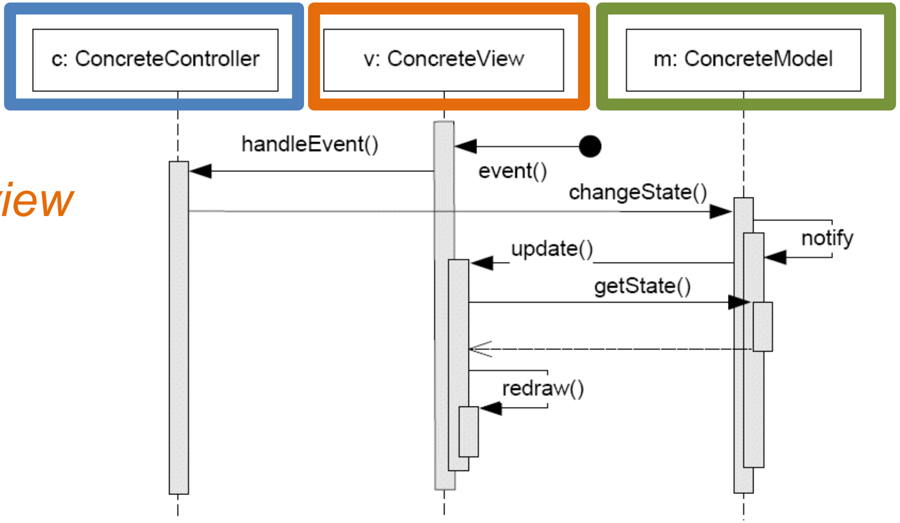
- Interpret user input events and translate them into state changes in the Model.

Dynamics


c: ConcreteController
v: ConcreteView
m: ConcreteModel
handleEvent()
event()
changeState()
notify
update()
getState()
redraw()

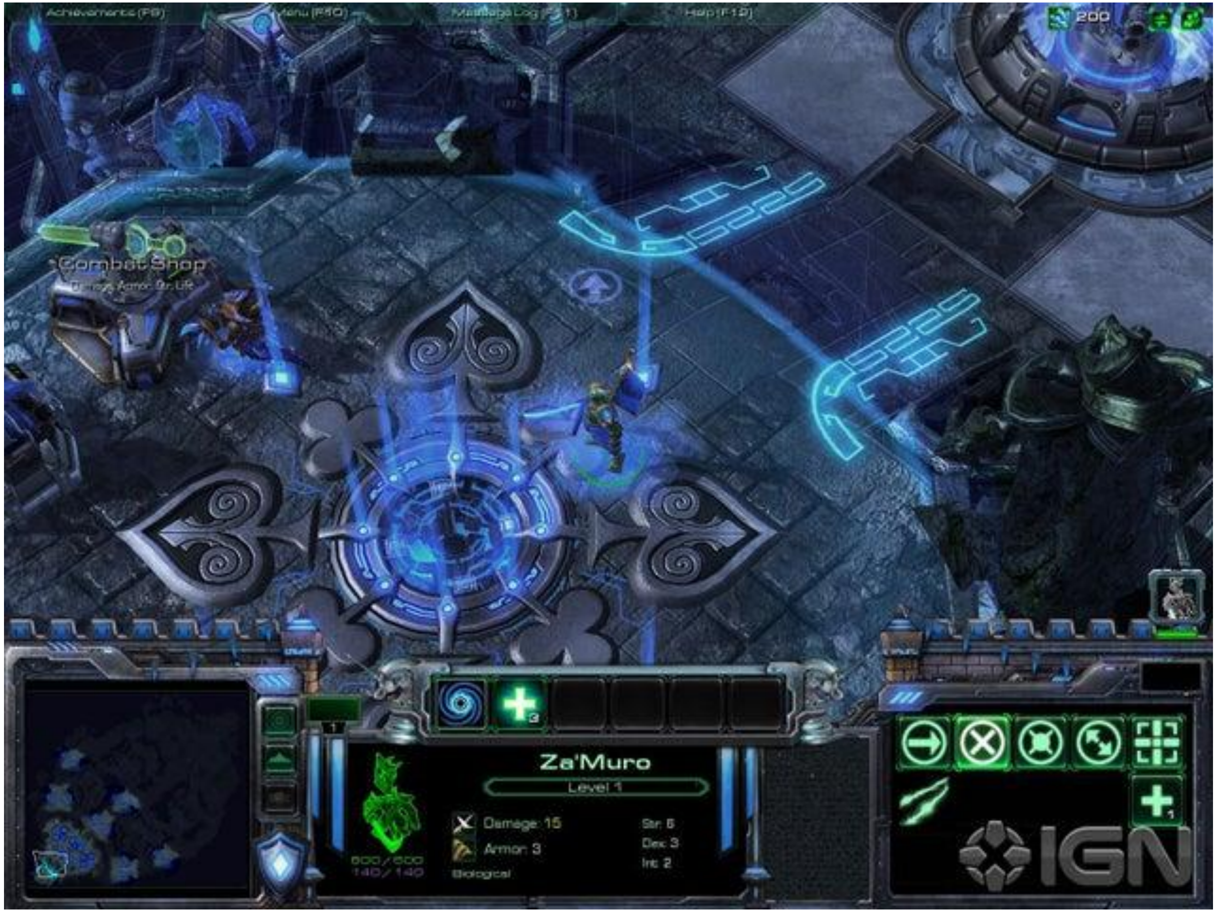
Discussion

- So much *pain* for so little???
  - To draw one lousy pixel with the mouse…
  - I have to code
    - *A tool/controller to intercept mouse events*
    - *Send it to the model*
      - *That does state change*
      - *That notifies…*
    - *Some registered observers/view*
      - *That receives the event*
      - *And then finally draw stuff*
- *Exercise:*
  - *Why all this pain???*



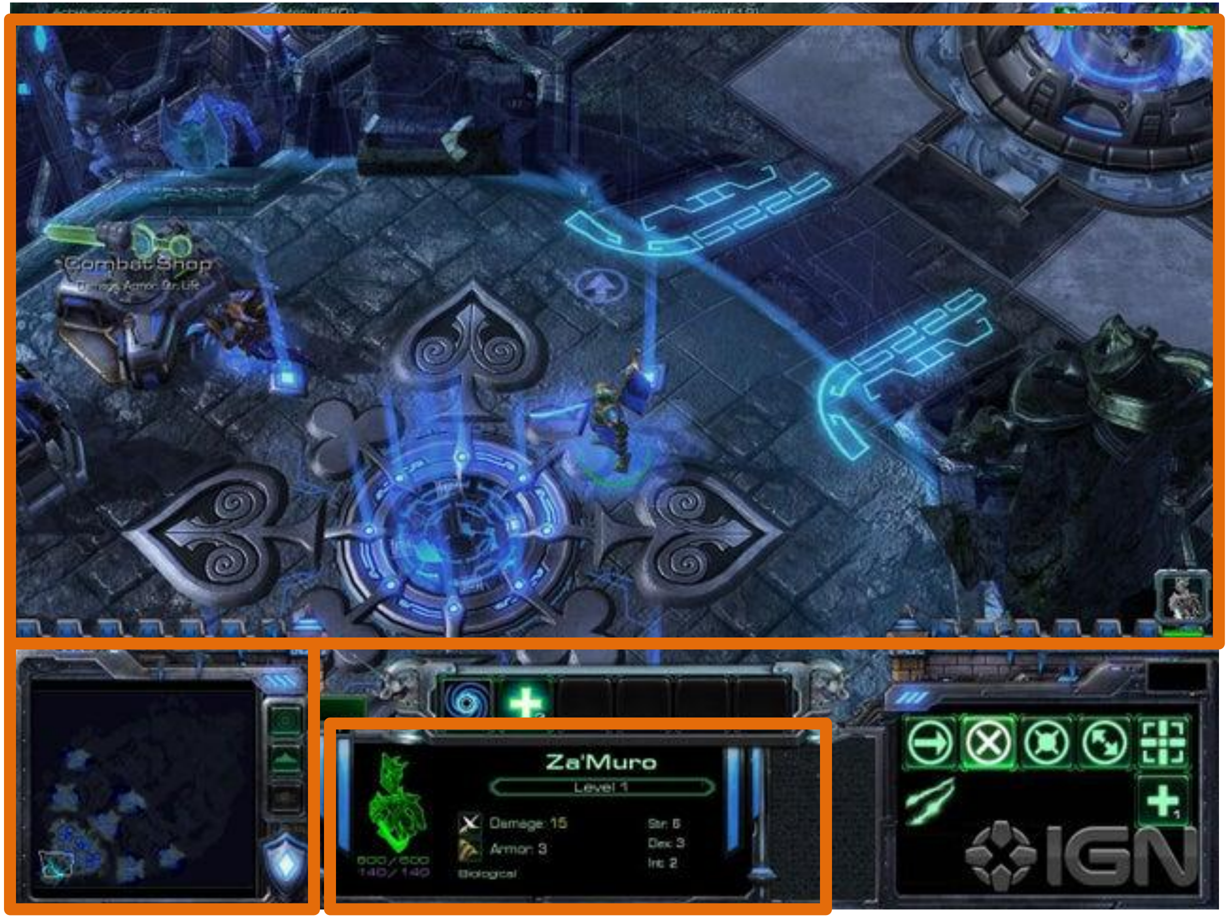
Visualizing MVC

Starcraft II

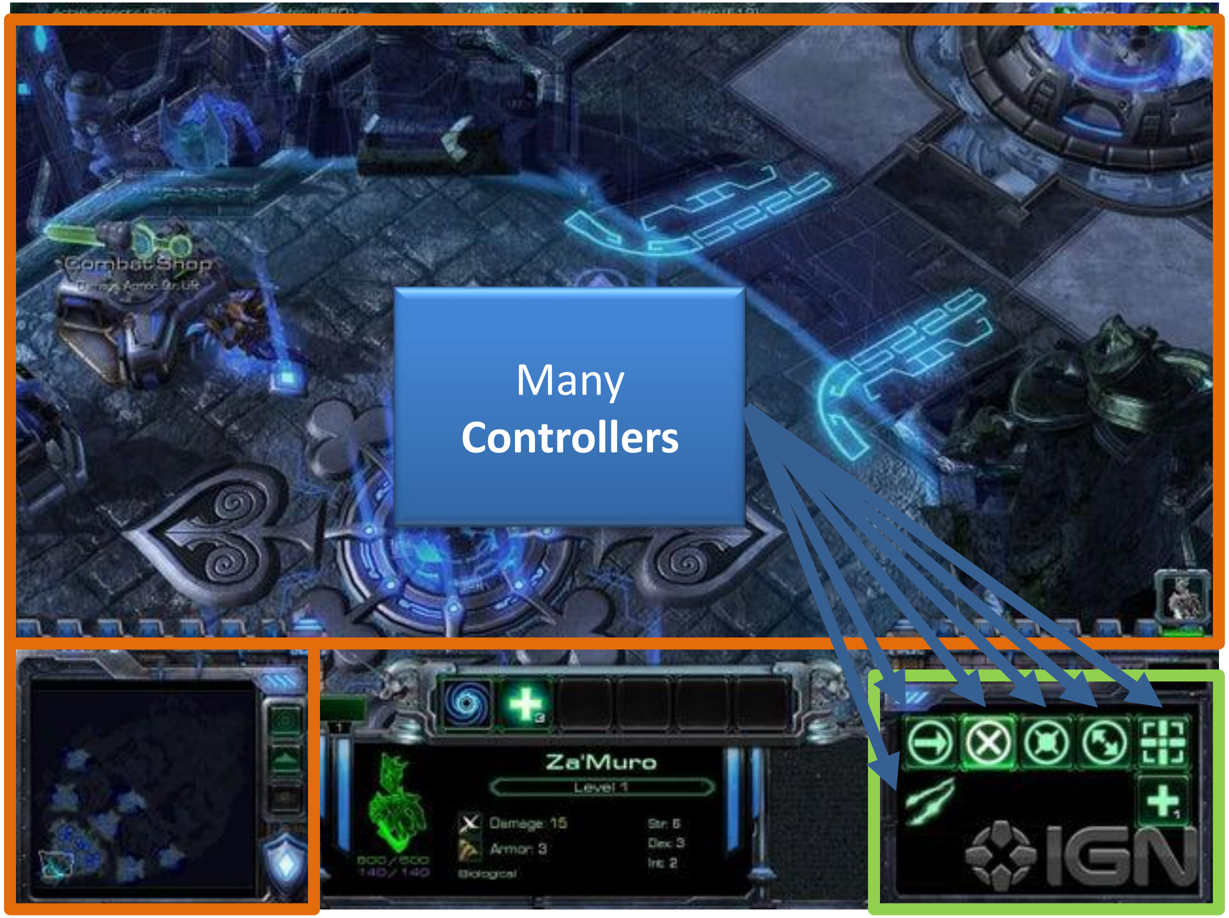


Visualizing MVC

Three **Views**



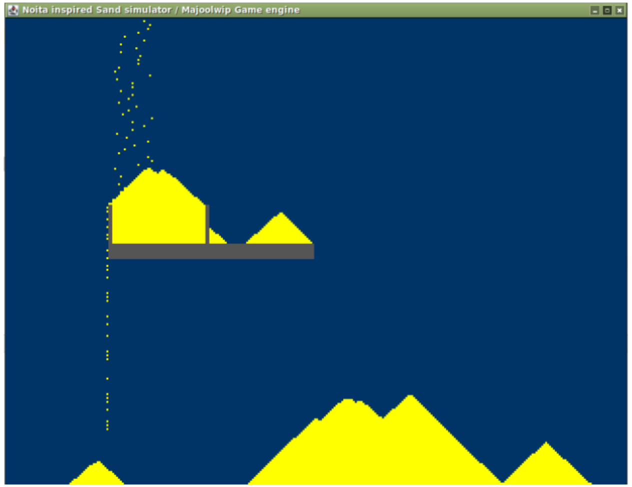
Visualizing MVC



Many **Controllers**

(Side note)

- Never too old to play ☺…
- In the 80'ies, there was not enough memory for
  - The Model
  - The Gfx that rendered the model
- So
  - The Model **was** the pixels drawn!
- *Noita* follows this tradition ☺

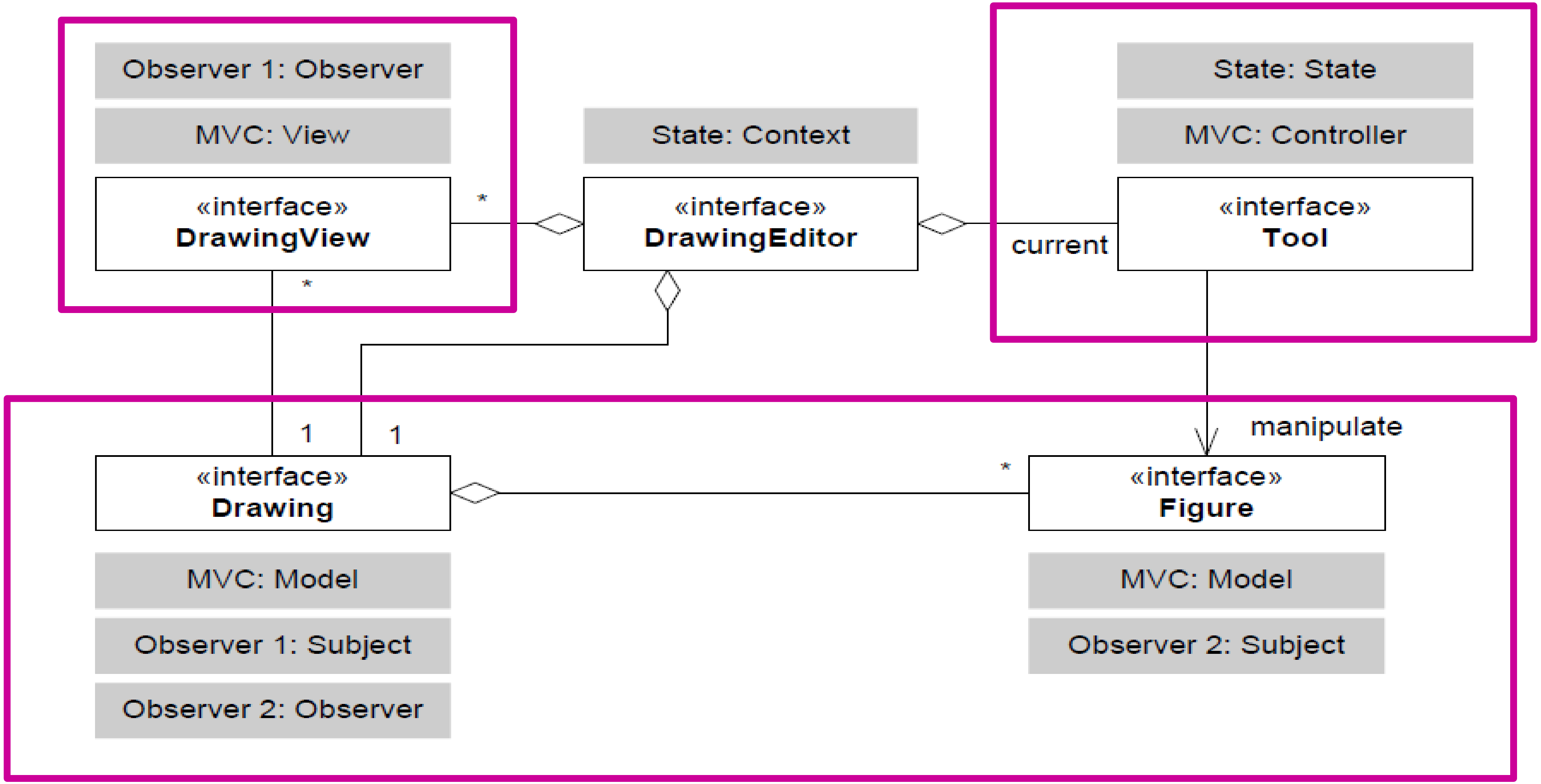


AARHUS UNIVERSITET

MiniDraw

Outline of its Architecture

MiniDraw: Role Diagram

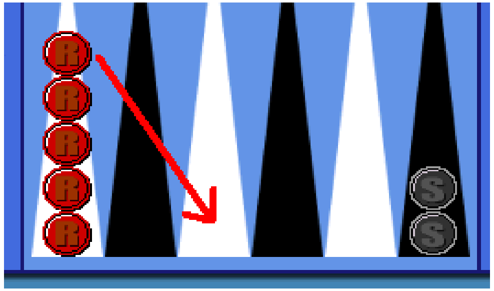


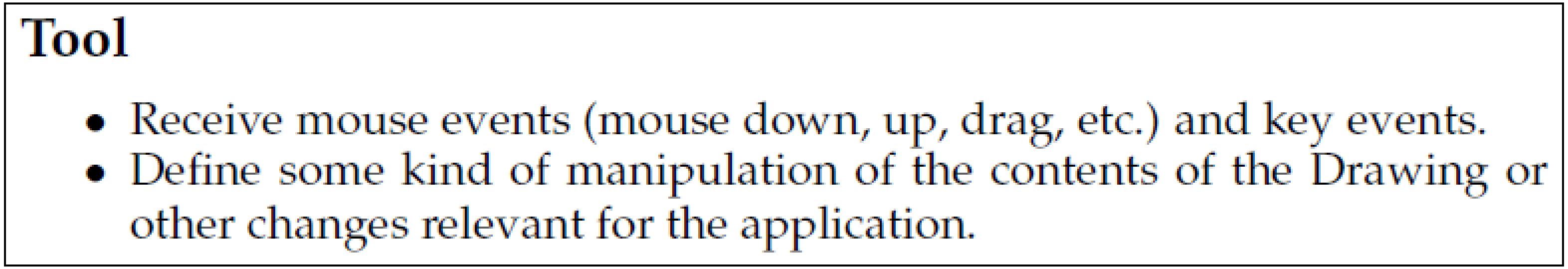
Tool: The Controller role

MiniDraw: Tool Interaction

- Basic paradigm: *Direct Manipulation*

- *[Demo: puzzle]*



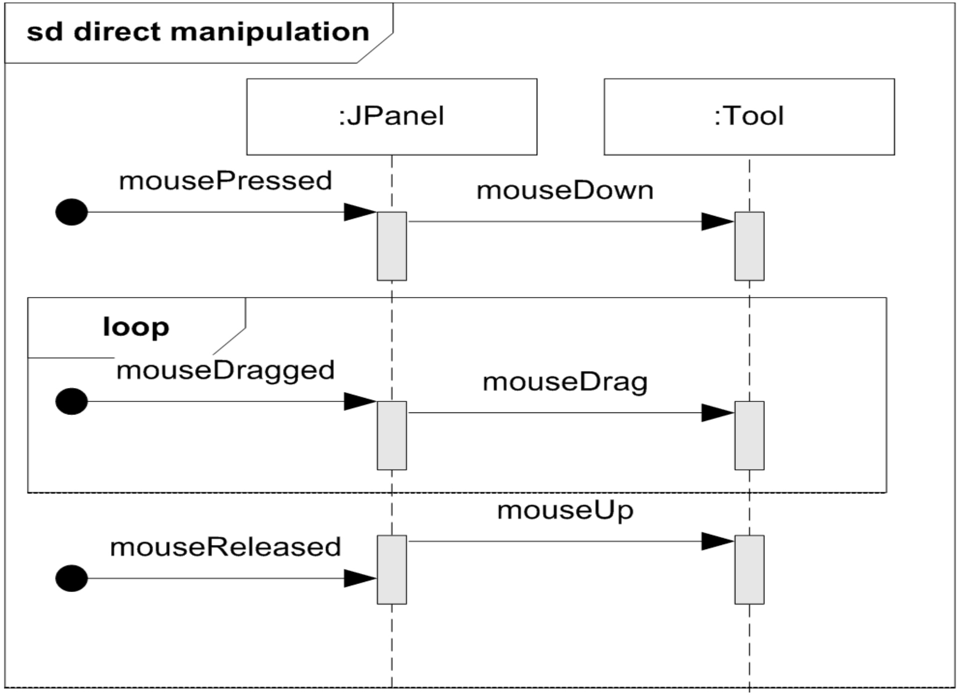


Tool

- Receive mouse events (mouse down, up, drag, etc.) and key events.
- Define some kind of manipulation of the contents of the Drawing or other changes relevant for the application.

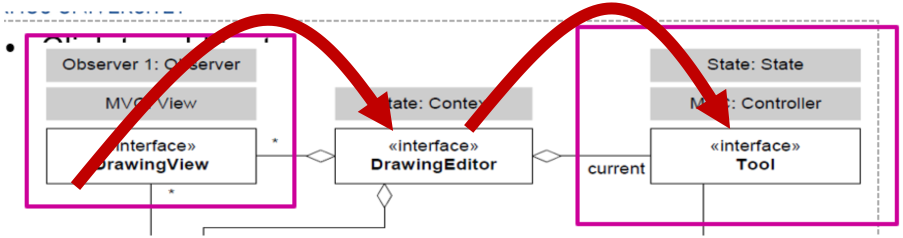
View -> Controller interaction

- Mouse events *do* hit the Swing JPanel, but MiniDraw simply delegates to its active tool...
  - The State pattern in action
    - *Let the tool do the job*

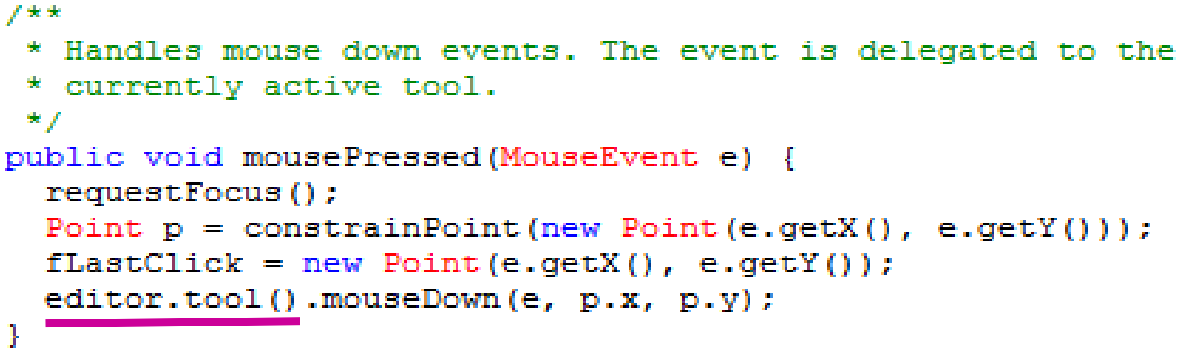


MiniDraw vrs MVC

- MiniDraw uses a ‘middle man’: The Editor



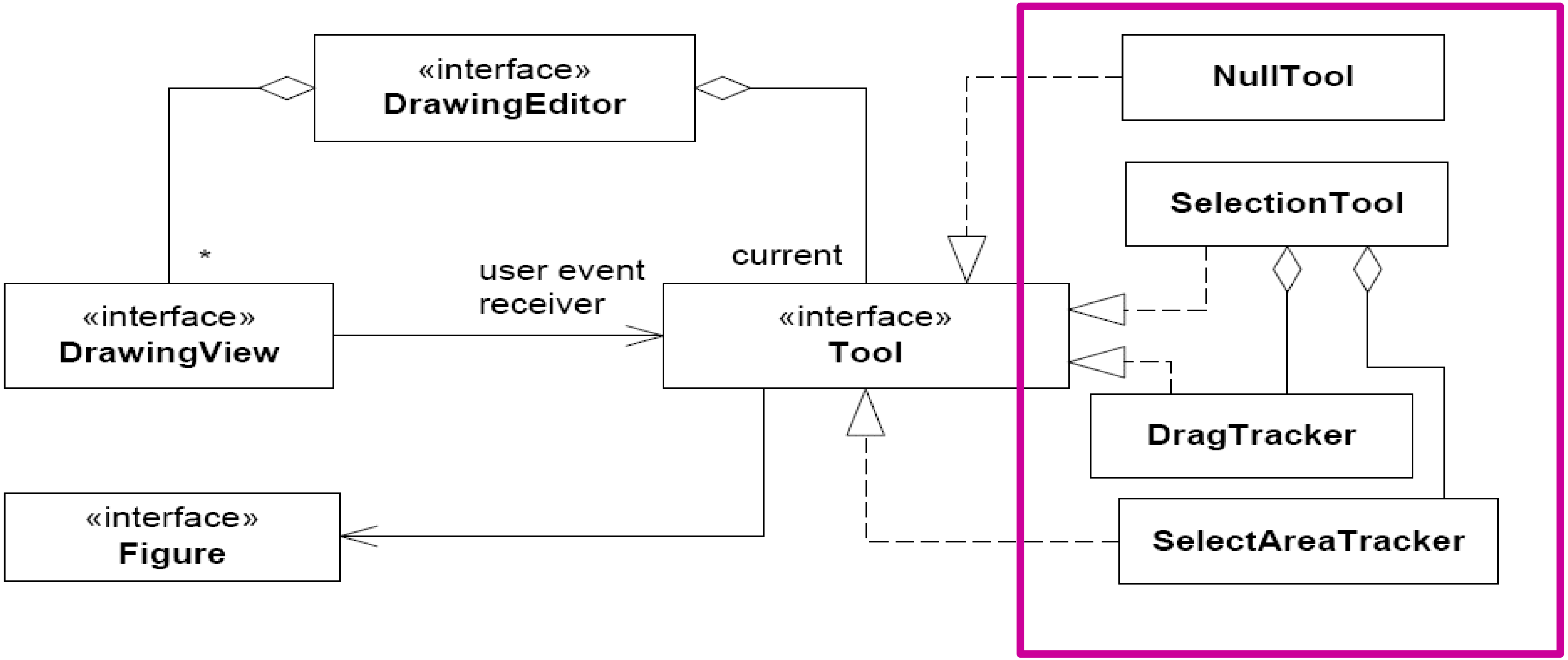
  – The view requests access to the *editor’s current tool*
    - *Aka: delegating the request to state.request()*

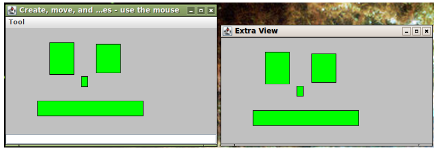


```
/**
 * Handles mouse down events. The event is delegated to the
 * currently active tool.
 */
public void mousePressed(MouseEvent e) {
  requestFocus();
  Point p = constrainPoint(new Point(e.getX(), e.getY()));
  fLastClick = new Point(e.getX(), e.getY());
  editor.tool().mouseDown(e, p.x, p.y);
}
```

Tools

- MiniDraw has some simple tools defined

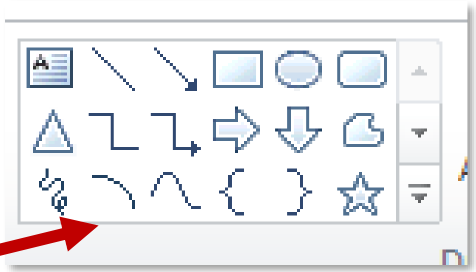




Code view

- It is very simple to set a new tool:
- editor.setTool(t);

This is the code equivalent of this UI tool box



- where t is the tool you want to become active.
- **Framework: You can define your own tool types!**
  - **A framework hotspot**

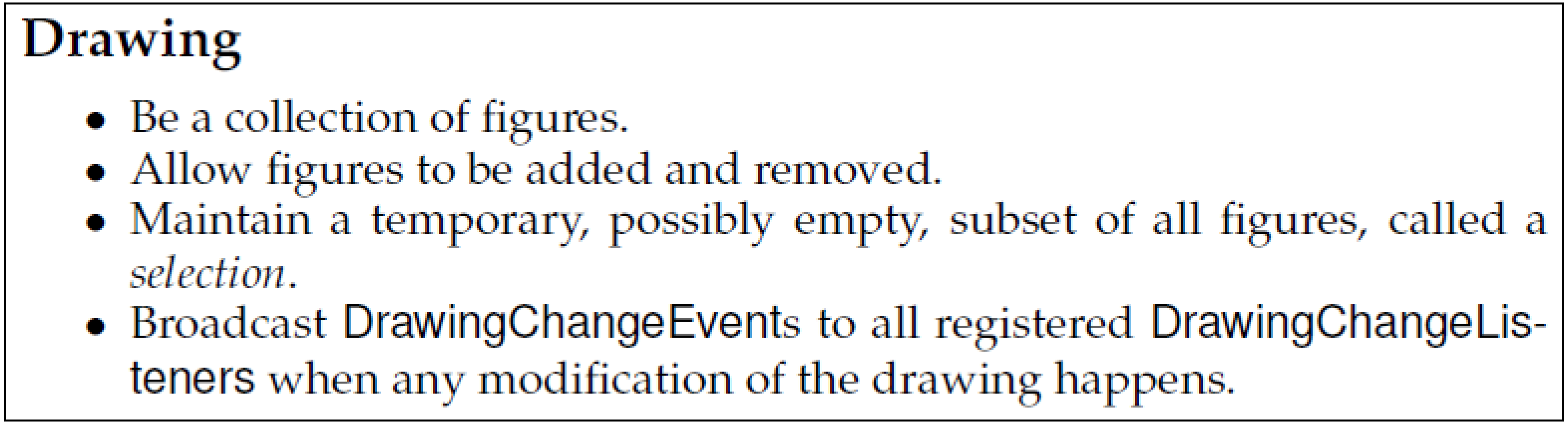
Drawing: The Model role

MiniDraw 3.x rewrote the code base to be purely compositional.

AARHUS UNIVERSITET

Drawing

- Drawing – is responsible for quite a lot…



Drawing

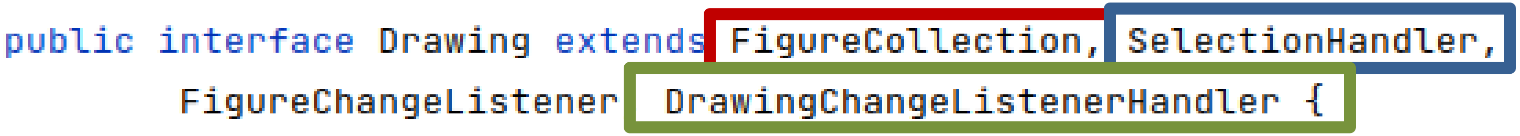
- Be a collection of figures.
- Allow figures to be added and removed.
- Maintain a temporary, possibly empty, subset of all figures, called a *selection*.
- Broadcast DrawingChangeEvents to all registered DrawingChangeListeners when any modification of the drawing happens.

- How to model that in the *compositional paradigm?*
  - ***By composition, of course!***

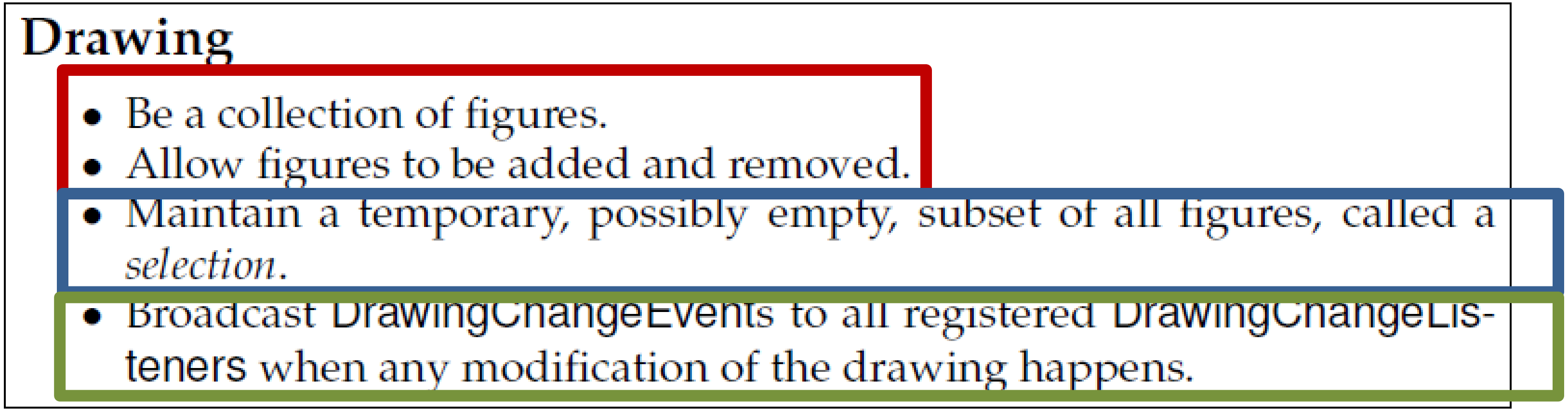
… and we partially covered that in Week 6: Compositional Design

The Interface in MiniDraw

- So – it is defined in terms of *fine-grained roles*
  - *Role interfaces*



```
public interface Drawing extends FigureCollection, SelectionHandler,
        FigureChangeListener  DrawingChangeListenerHandler {
```

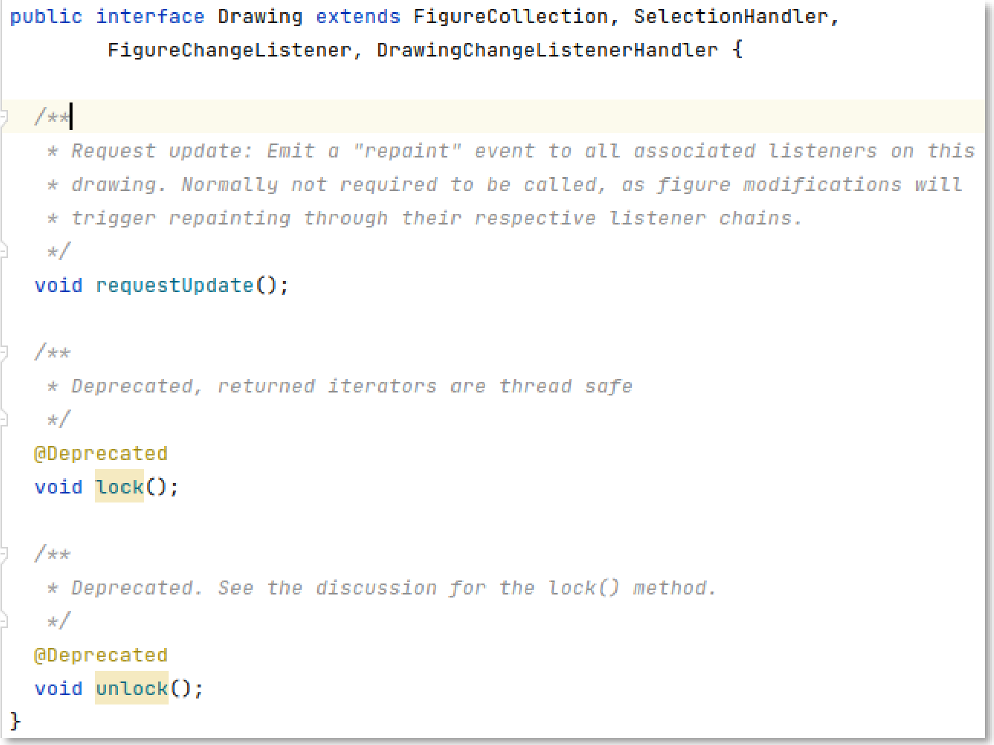


Drawing

- Be a collection of figures.
- Allow figures to be added and removed.
- Maintain a temporary, possibly empty, subset of all figures, called a *selection*.
- Broadcast DrawingChangeEvents to all registered DrawingChangeListeners when any modification of the drawing happens.

And the Interface is nearly Empty

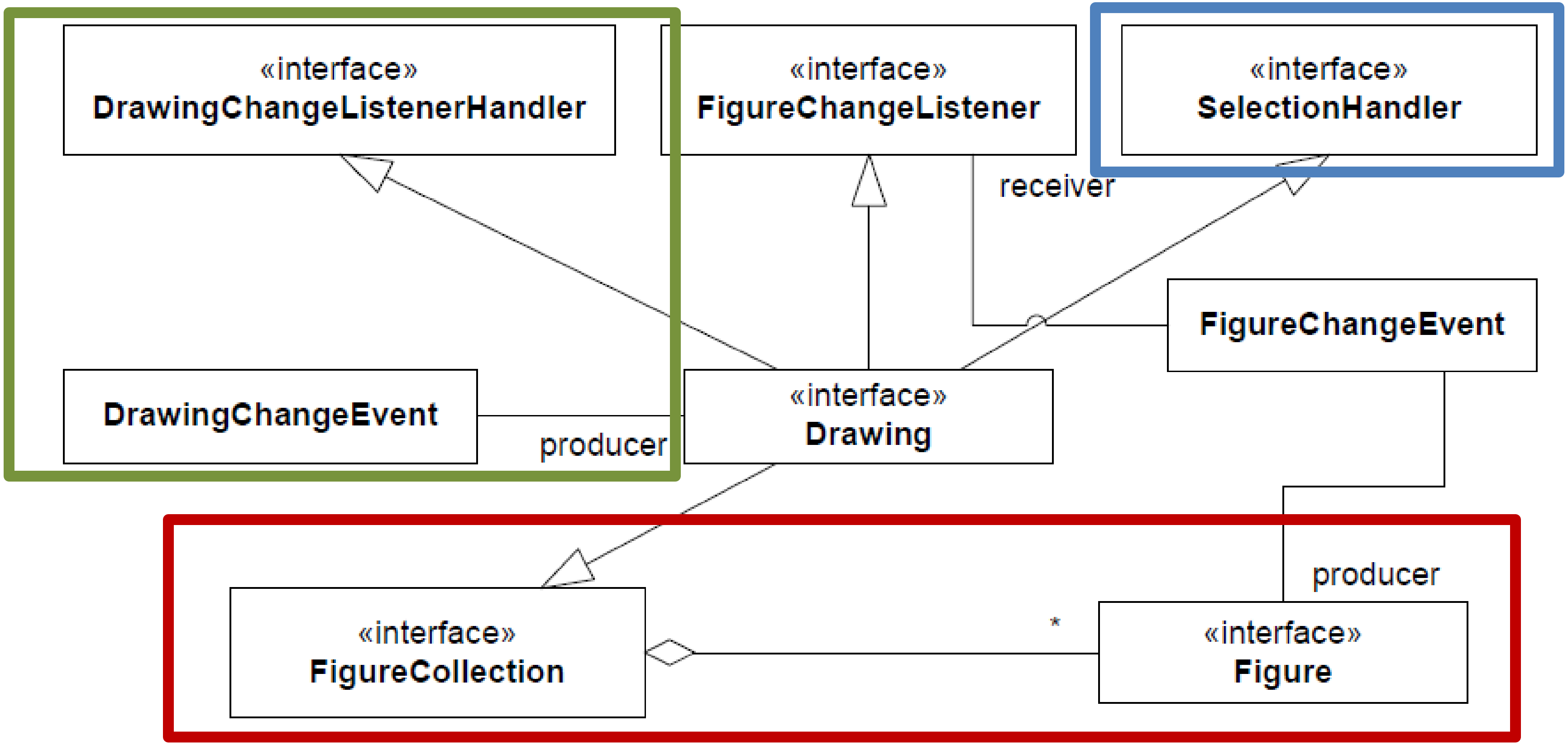
- One little extra responsibility is all there is…



```java
public interface Drawing extends FigureCollection, SelectionHandler,
        FigureChangeListener, DrawingChangeListenerHandler {

  /**
   * Request update: Emit a "repaint" event to all associated listeners on this
   * drawing. Normally not required to be called, as figure modifications will
   * trigger repainting through their respective listener chains.
   */
  void requestUpdate();

  /**
   * Deprecated, returned iterators are thread safe
   */
  @Deprecated
  void lock();

  /**
   * Deprecated. See the discussion for the lock() method.
   */
  @Deprecated
  void unlock();
}
```

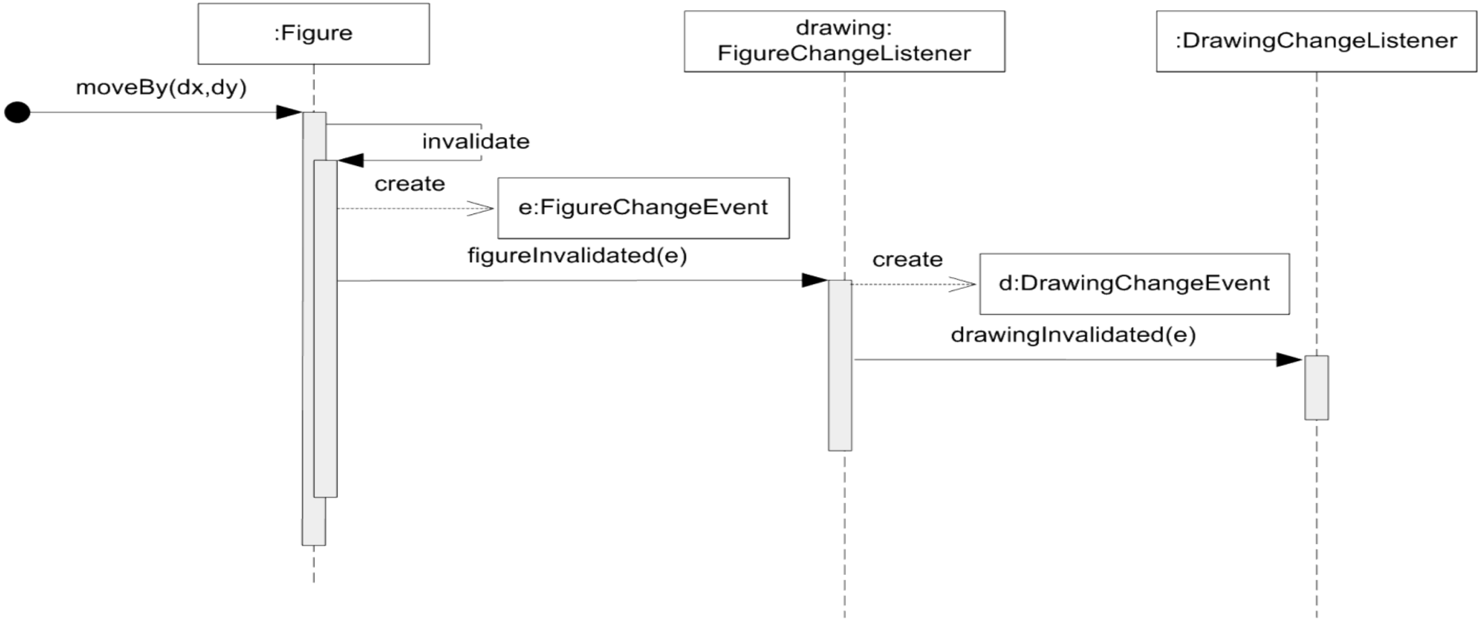
MiniDraw: Drawing

- Static view



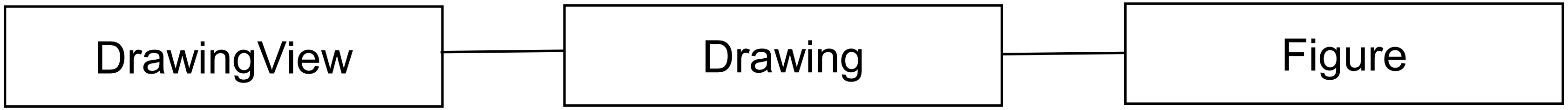
MiniDraw: Drawing

- But how does the view get repainted?
  - *Double* observer chain
    - Figure *notifies* drawing, which again *notifies* drawing view.



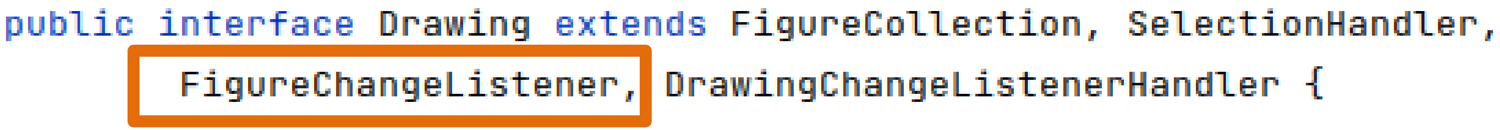
Exercise:

- Observer pattern has two *roles*
  - *Subject*: Container of data
  - *Observer*: Object to notify upon data changes
- Who are who here???



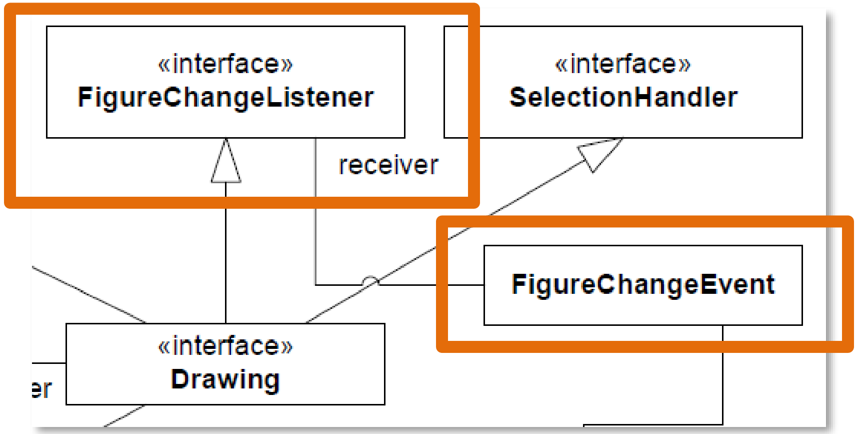
So the last role

- The last role that the Drawing serves…



```
public interface Drawing extends FigureCollection, SelectionHandler,
        FigureChangeListener, DrawingChangeListenerHandler {
```

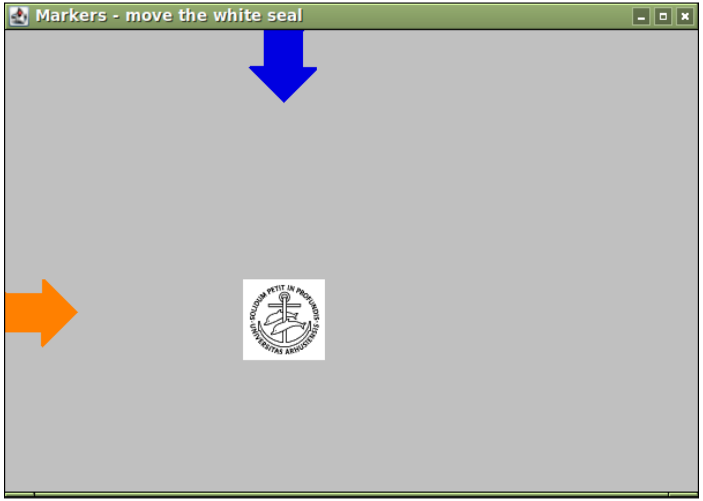
- … is to *listen to any change events from the figures it contains* in order to be able to fire the drawing event…



Flexibility

- As any object, including a Figure itself, can listen to FigureChangs…
  - We can create *semantic bindings*

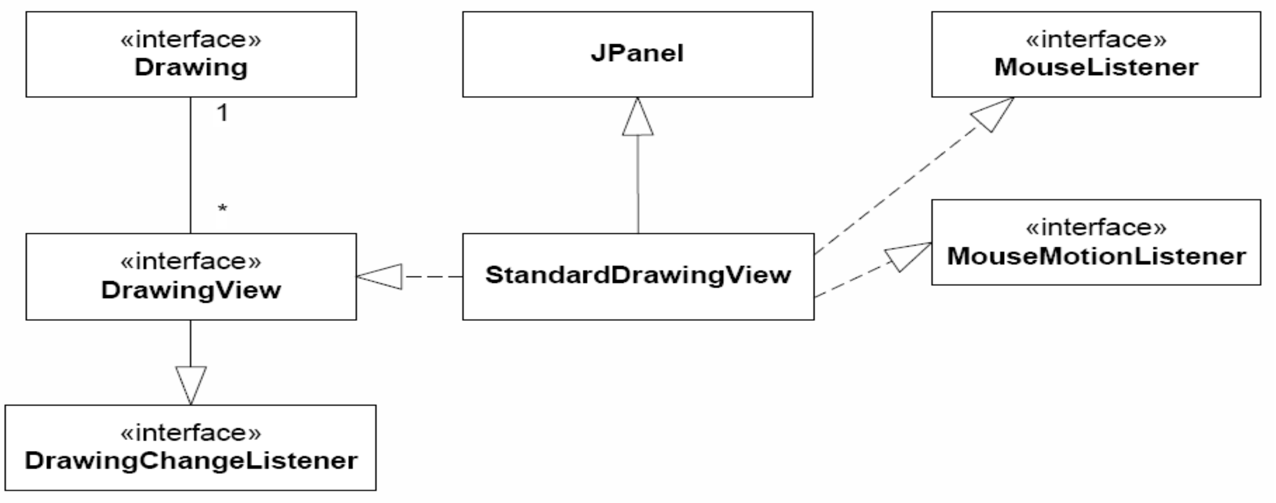
- MiniDraw can be used for to create a UML diagram editor…



DrawingView: The View role

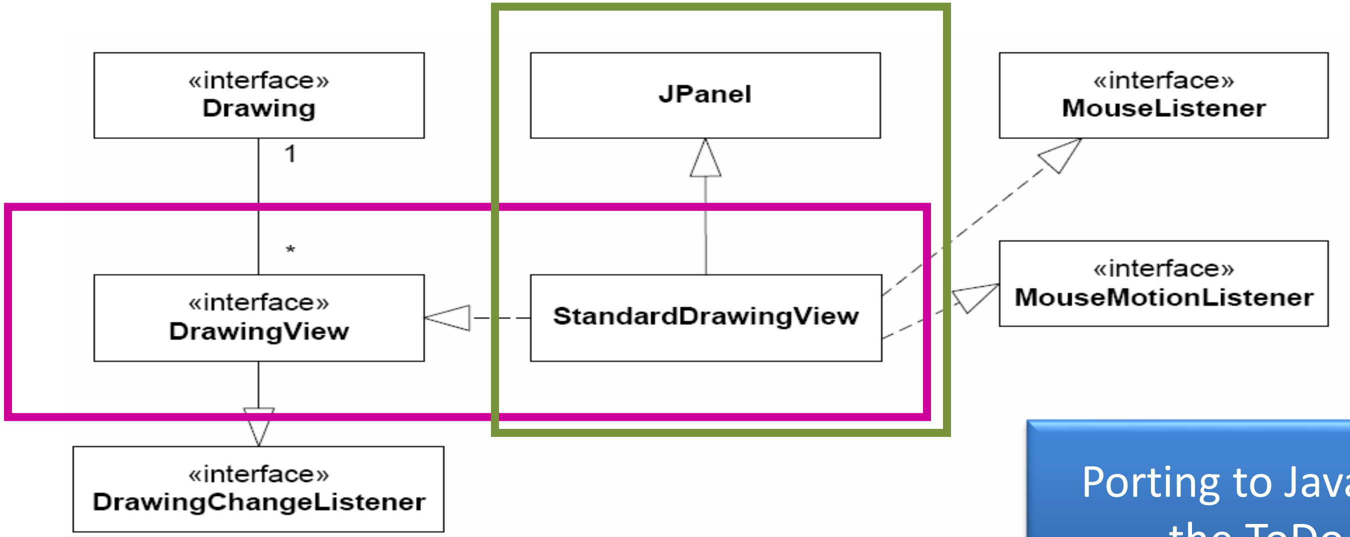
View

- 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

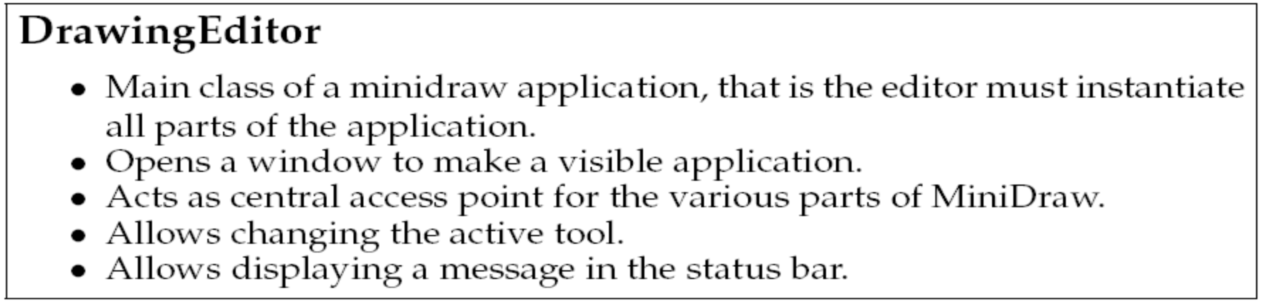
- Note that this design **combines two frameworks**
  - MiniDraw and Swing
  - If DrawingView was *not* an interface then ☠



Porting to JavaFX is on the ToDo list!

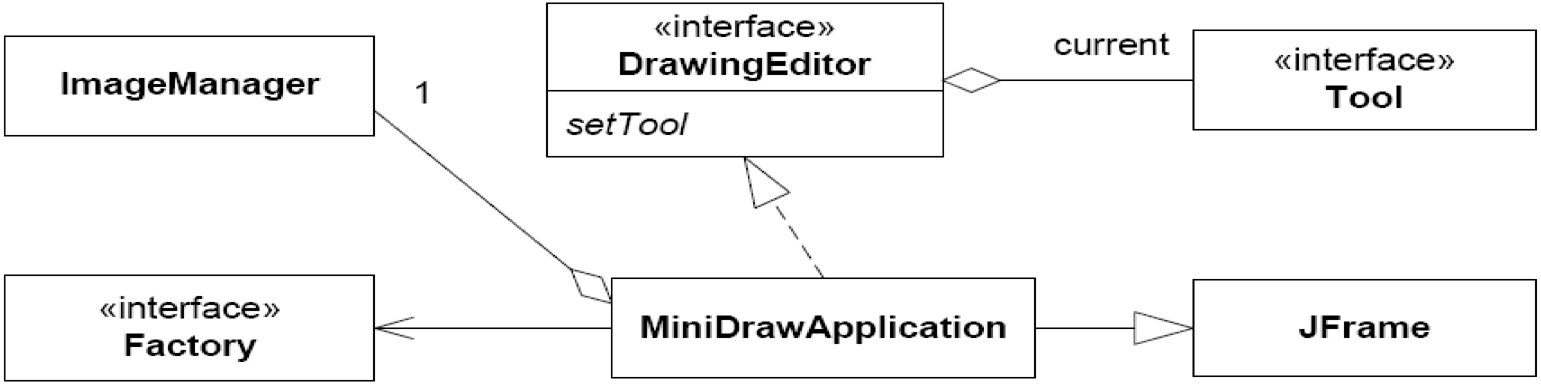
DrawingEditor: The Coordinator

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

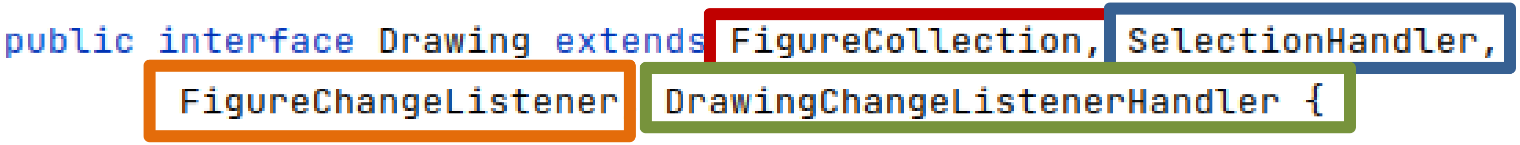
- Most MiniDraw roles have default implementations:
  - Interface `X` has default implementation `StandardX`
  - `DrawingView -> StandardDrawingView`

- There are also some partial implementations:
  - Interface `X` has partial implementation `AbstractX`
  - `Tool -> AbstractTool`
  - `Figure -> AbstractFigure`

Compositional Design

- *Complex behaviour as a result of combining simple behaviour...*

- Example:
  - CompositionalDrawing implements Drawing

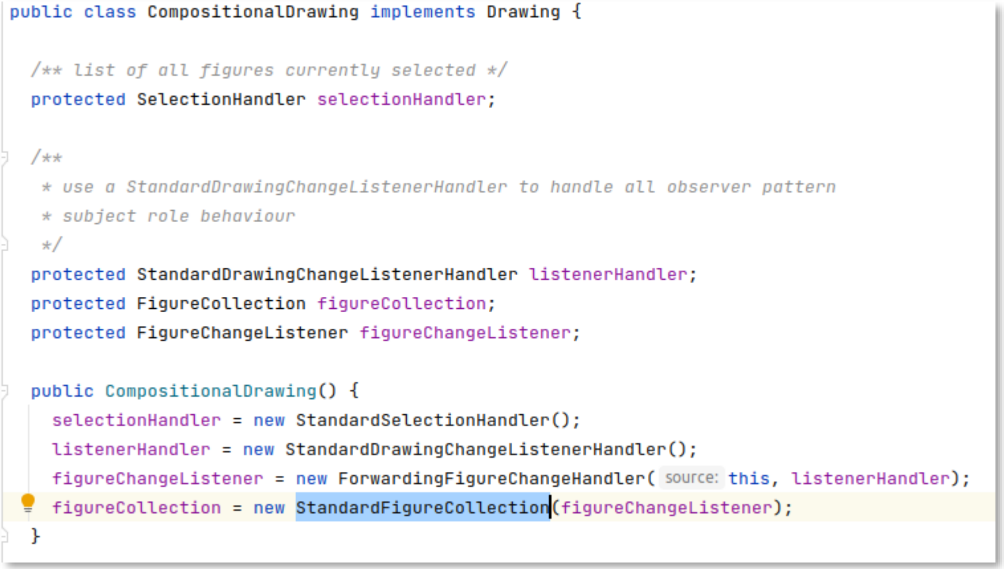


```
public interface Drawing extends FigureCollection, SelectionHandler,
        FigureChangeListener DrawingChangeListenerHandler {
```

How do we do that?

- Proposal 1:
  - *implement ahead...*
- Proposal 2:
  - *encapsulate major responsibilities in separate objects and compose behavior*

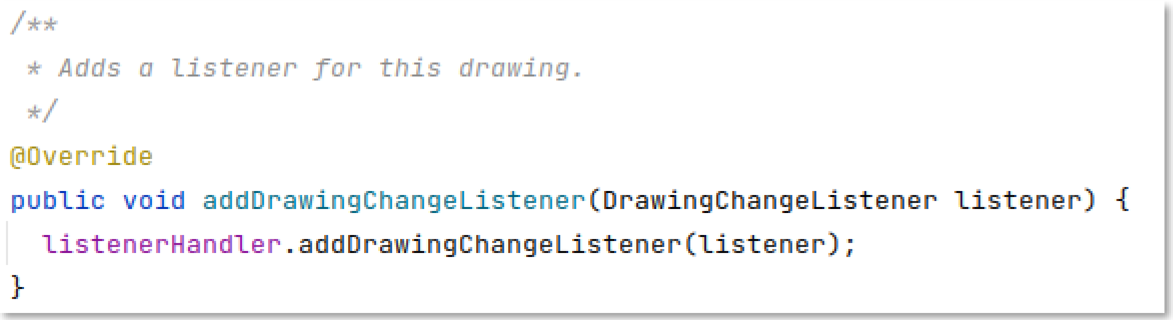
… and we partially covered that in Week 6: Compositional Design



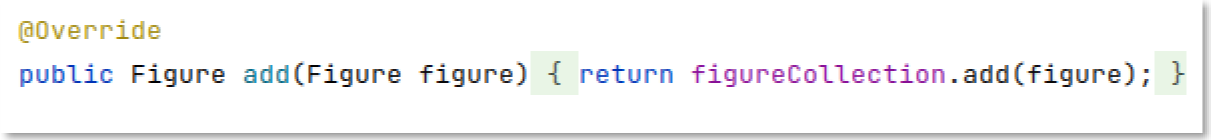
```java
public class CompositionalDrawing implements Drawing {

  /** list of all figures currently selected */
  protected SelectionHandler selectionHandler;

  /**
   * use a StandardDrawingChangeListenerHandler to handle all observer pattern
   * subject role behaviour
   */
  protected StandardDrawingChangeListenerHandler listenerHandler;
  protected FigureCollection figureCollection;
  protected FigureChangeListener figureChangeListener;

  public CompositionalDrawing() {
    selectionHandler = new StandardSelectionHandler();
    listenerHandler = new StandardDrawingChangeListenerHandler();
    figureChangeListener = new ForwardingFigureChangeHandler( source: this, listenerHandler);
    figureCollection = new StandardFigureCollection(figureChangeListener);
  }
```

Code view: delegations!

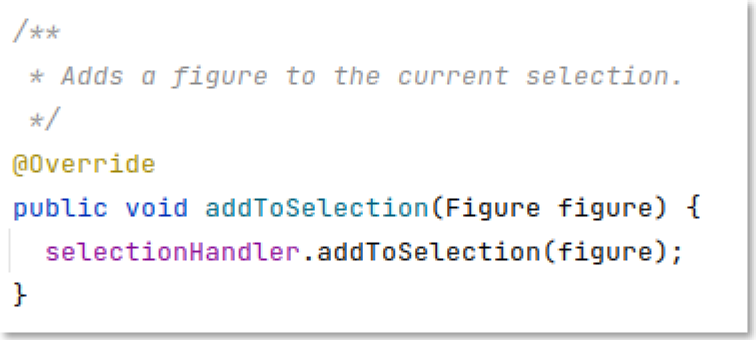
- Examples:



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



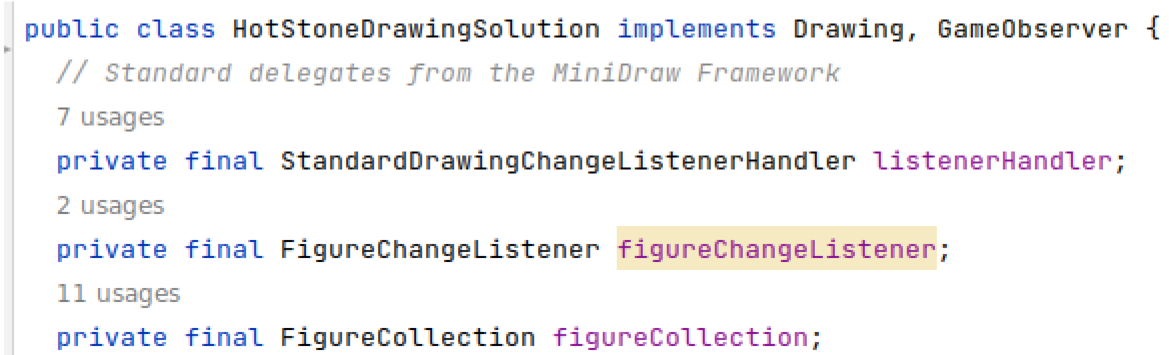
```java
@Override
public Figure add(Figure figure) { return figureCollection.add(figure); }
```



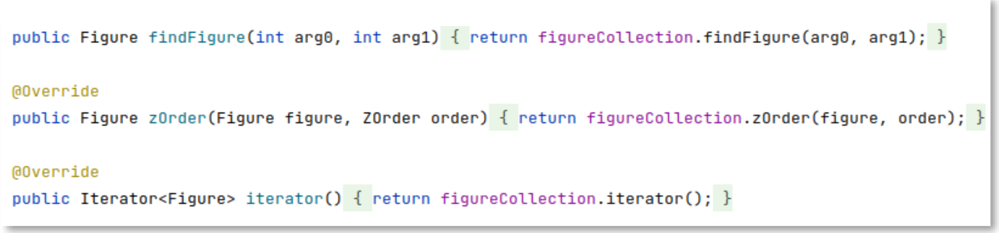
```java
/**
 * Adds a figure to the current selection.
 */
@Override
public void addToSelection(Figure figure) {
  selectionHandler.addToSelection(figure);
}
```

What do I achieve?

- Implementing a custom Drawing
  - In which the figure collection works differently…
    - As in our **HotStoneDrawing**
  - but I can *reuse* the collection, the selection and drawing-change handler behavior directly!



```
public class HotStoneDrawingSolution implements Drawing, GameObserver {
  // Standard delegates from the MiniDraw Framework
  7 usages
  private final StandardDrawingChangeListenerHandler listenerHandler;
  2 usages
  private final FigureChangeListener figureChangeListener;
  11 usages
  private final FigureCollection figureCollection;
```



```
public Figure findFigure(int arg0, int arg1) { return figureCollection.findFigure(arg0, arg1); }

@Override
public Figure zOrder(Figure figure, ZOrder order) { return figureCollection.zOrder(figure, order); }

@Override
public Iterator<Figure> iterator() { return figureCollection.iterator(); }
```

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

Summary

- 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.