
# Programming and Algorithms II

## Lecture 10: Composite and MVC

**Nicolas Wu**

**nicolas.wu@bristol.ac.uk**

Department of Computer Science
University of Bristol

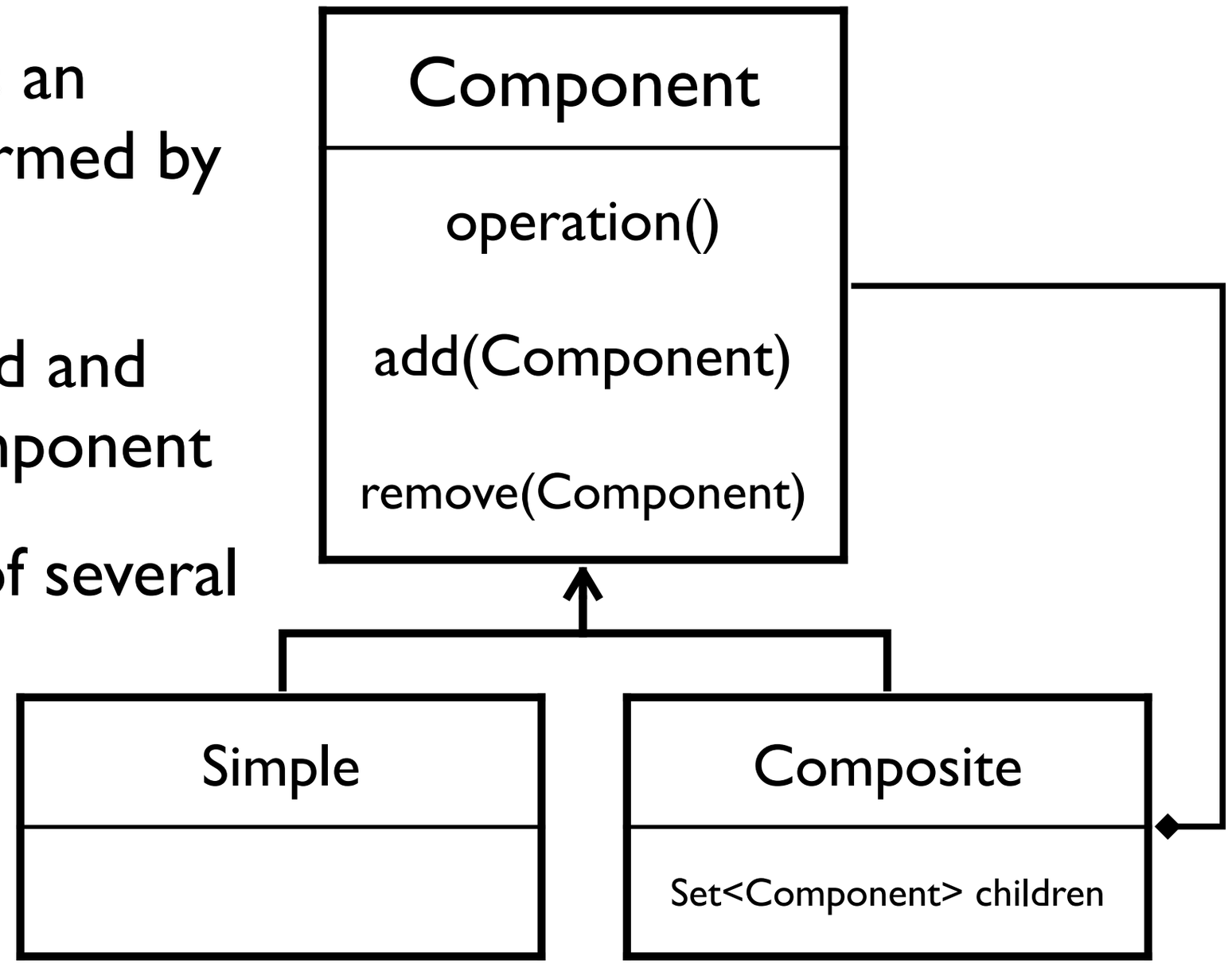
# Composite Pattern

# Composite Pattern

- GoF: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly
- Example: Your GUI window contains panes, which contain of buttons, which contain text, all of which need to be drawn

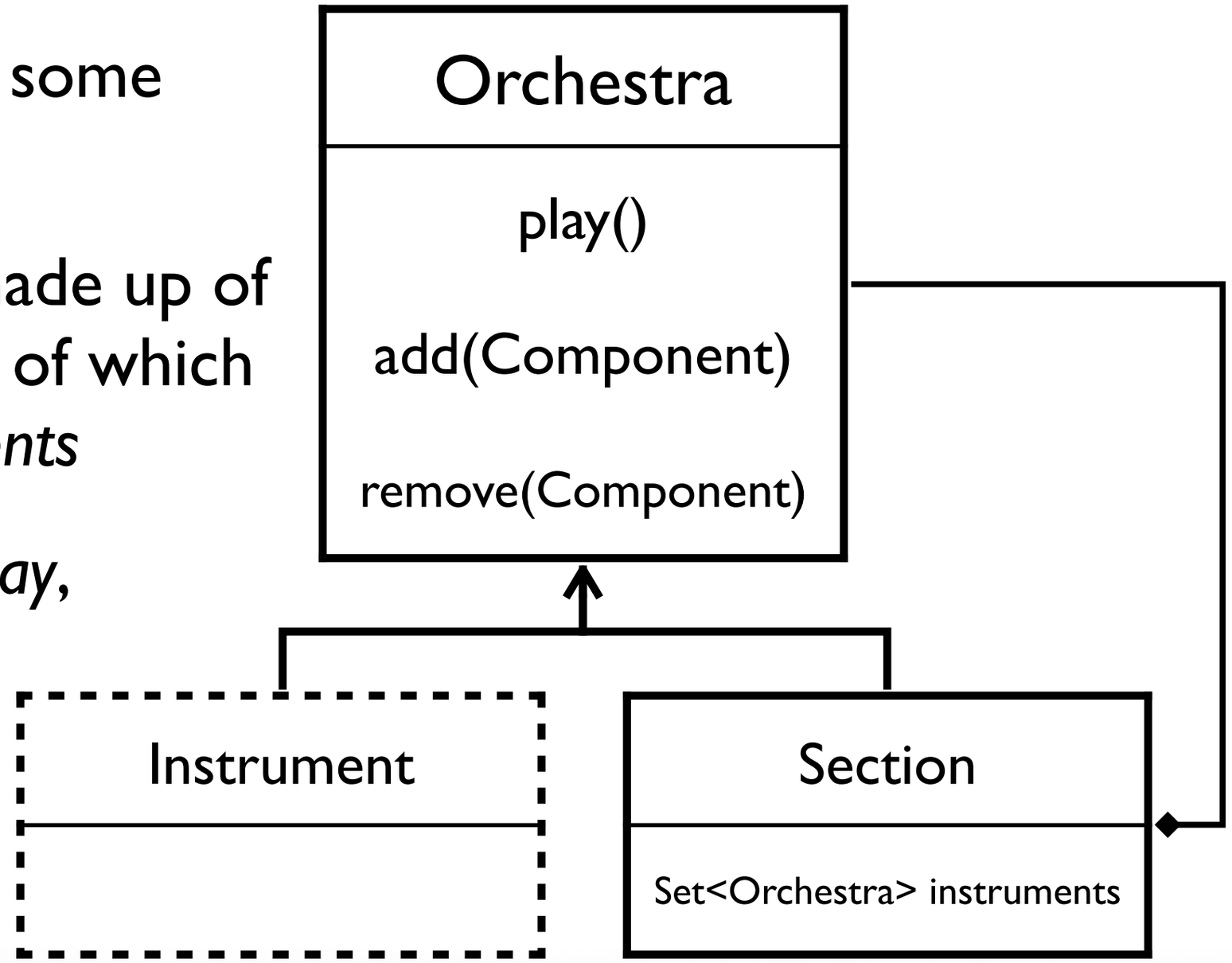
# Composite Pattern

- A *component* supports an *operation* that is performed by it and all its children
- Children may be *add*ed and *remov*ed from the component
- A *composite* consists of several *children* components
- *A simple* component has no children



# Composite Pattern

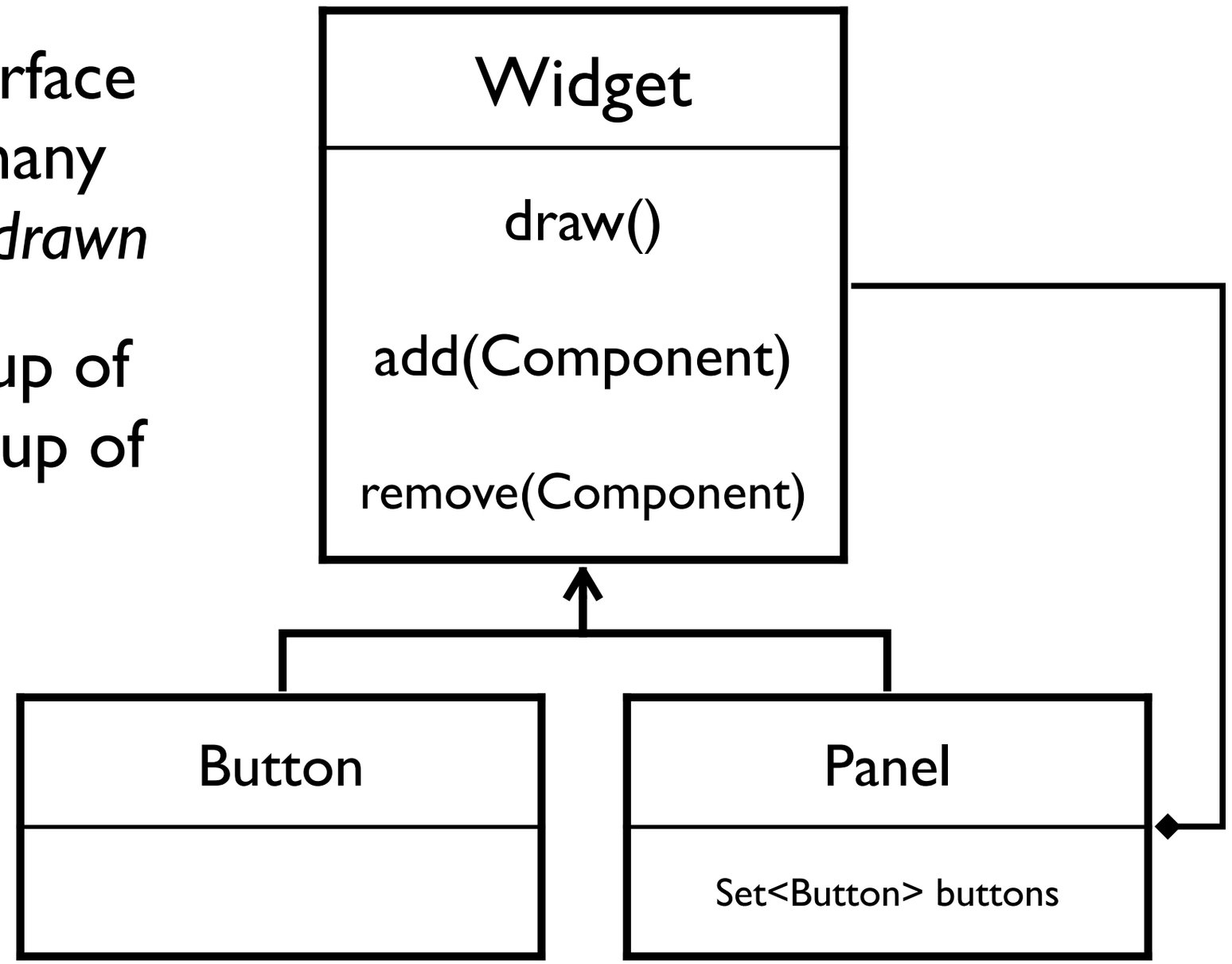
- An orchestra will *play* some music
- A large orchestra is made up of different *sections*, each of which is made up of *instruments*
- For an orchestra to *play*, so must all its composite parts




An instrument is the interface of a hierarchy containing Violin, Piano etc. These could have alternatively all been examples of a Simple class

# Composite Pattern

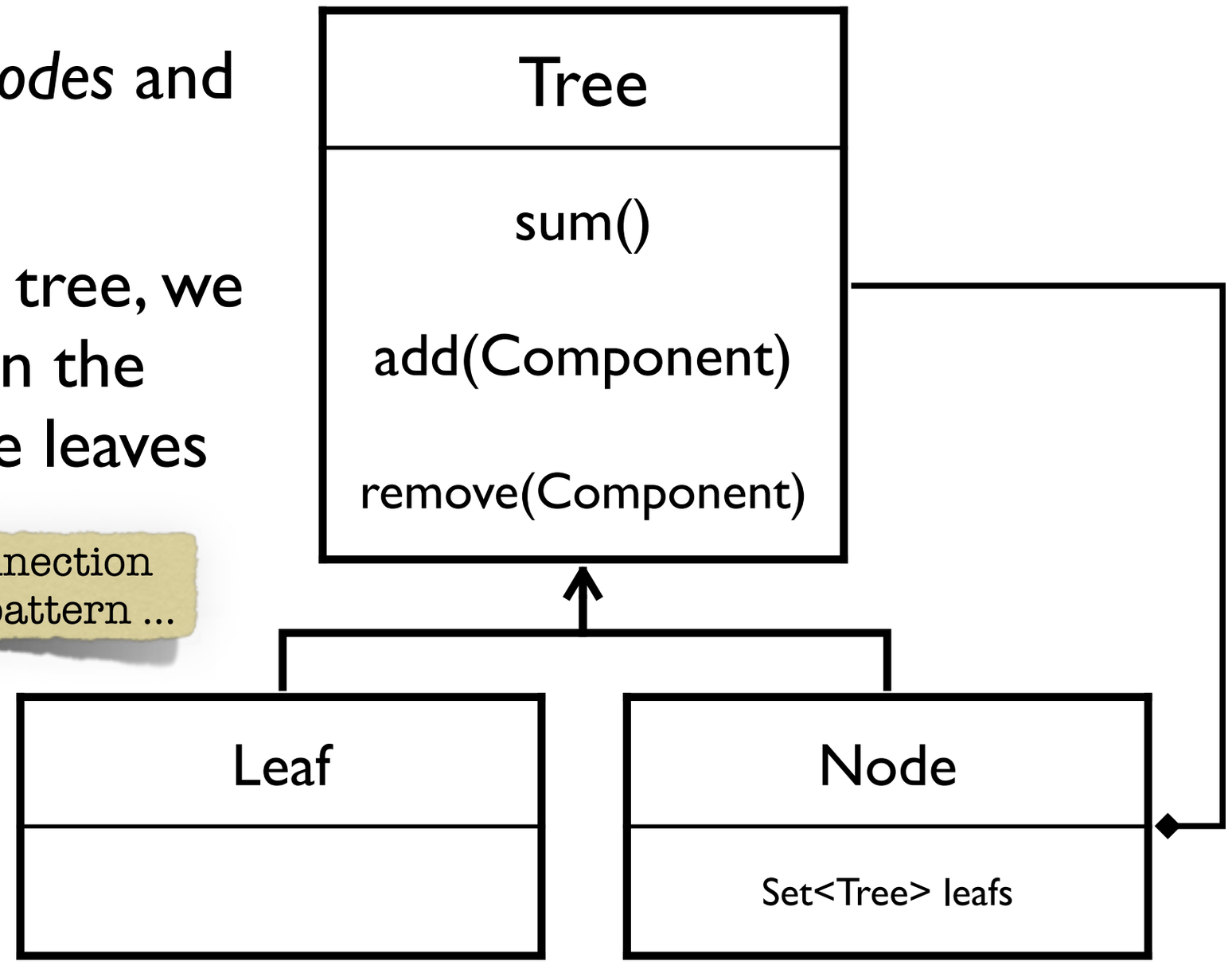
- A Graphical User Interface (GUI) is made up of many widgets that must be *drawn*
- The window is made up of panels, which is made up of buttons
- Asking a window to *draw* triggers a cascade of *draw* commands



# Composite Pattern

- A *tree* is made up of *nodes* and *leaves*
- To *sum* the values in a tree, we must *sum* the values in the nodes and those in the leaves

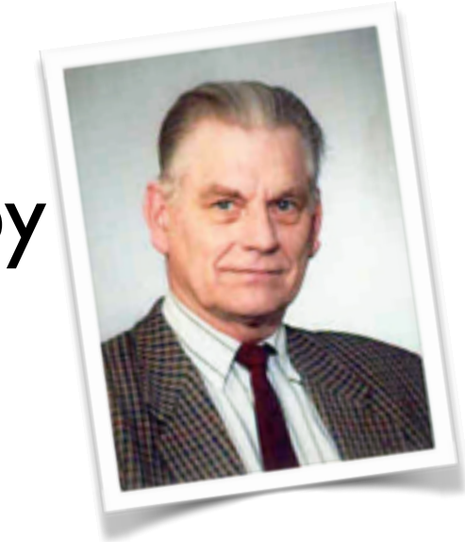
Usually there's a very close connection between Composite and Visitor pattern ...



# Model-View-Controller

# Model-View-Controller

- Model View Controller (MVC) is a pattern-of-patterns: it describes a general architecture for Graphical User Interface (GUI) design
- Originally called Thing-Model-View-Editor, by Trygve Reenskaug 1978-1979
- There are a *lot* of variations on this pattern
- The *intent* is to separate display (the view) from domain (the model)
- MVC is a pattern of patterns, and no well established "correct" definition



# Model-View-Controller

The Domain is the model, or data, or state of your application

Domain

Display

The Display is the GUI that your user interacts with

# Model-View-Controller

The Domain is the model, or data, or state of your application

Domain

Display

The Display is the GUI that your user interacts with

Keeping the two apart makes your code easier to manage

But keeping them truly separate would mean they never interact!

# Model-View-Controller

The Domain is the model, or data, or state of your application

Domain

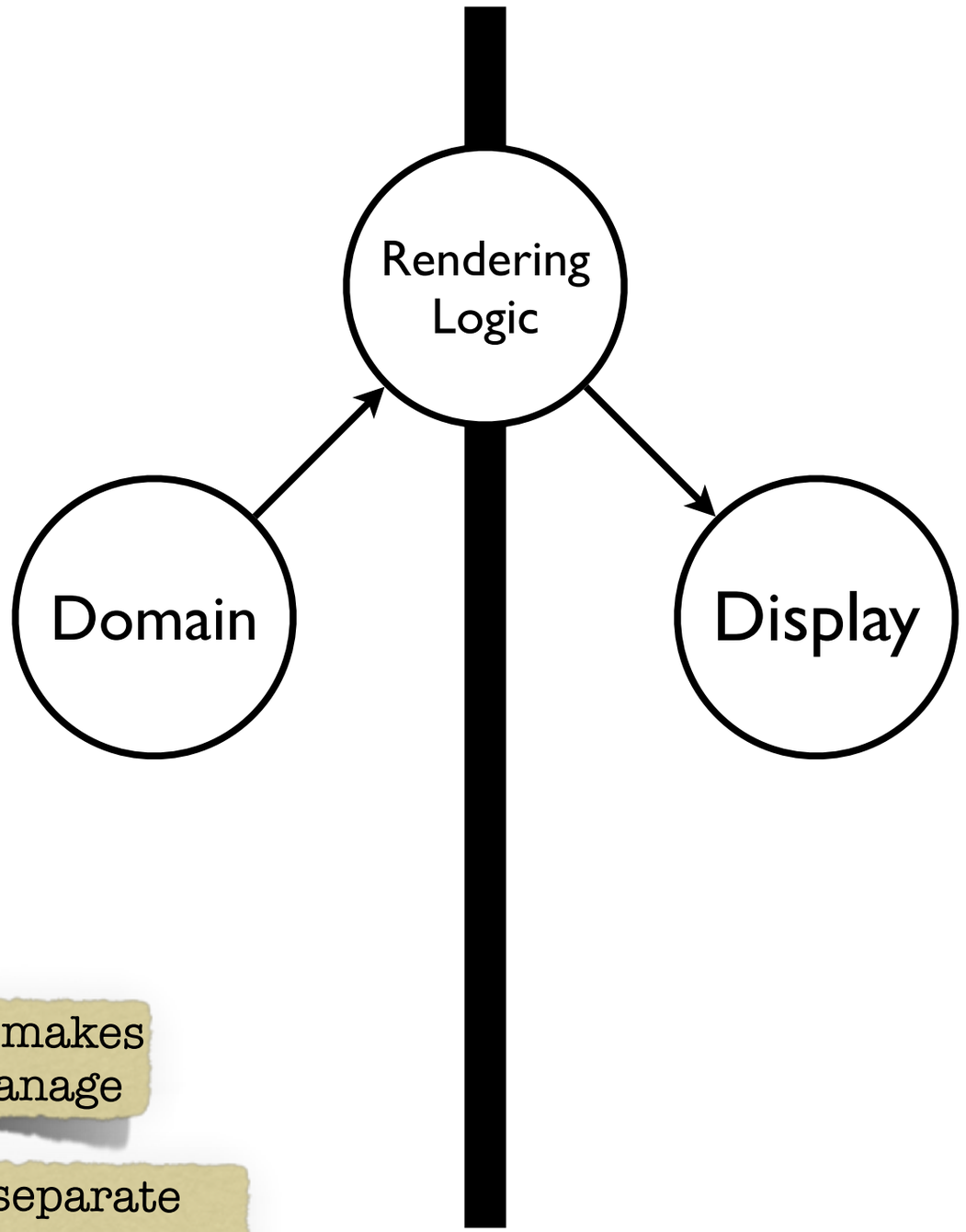
Display

The Display is the GUI that your user interacts with

Keeping the two apart makes your code easier to manage

But keeping them truly separate would mean they never interact!

# Model-View-Controller



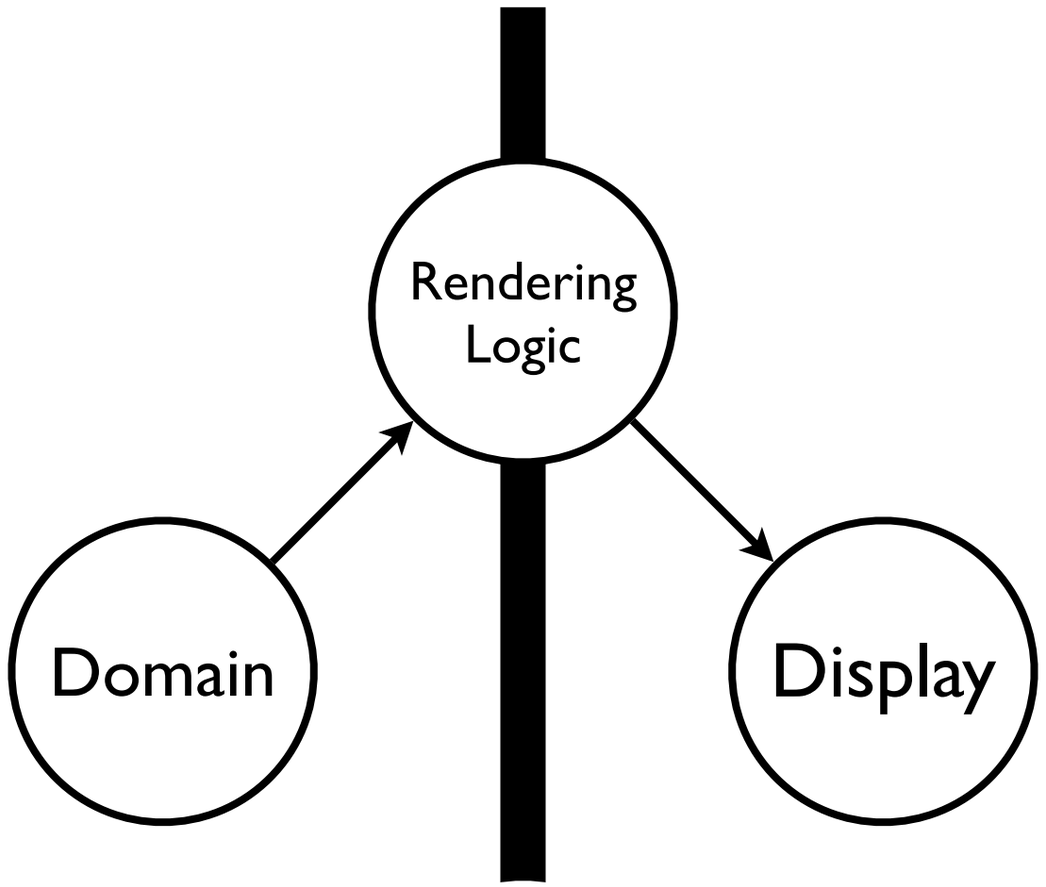

The Domain is the model, or data, or state of your application

The Display is the GUI that your user interacts with

Keeping the two apart makes your code easier to manage

But keeping them truly separate would mean they never interact!

# Model-View-Controller



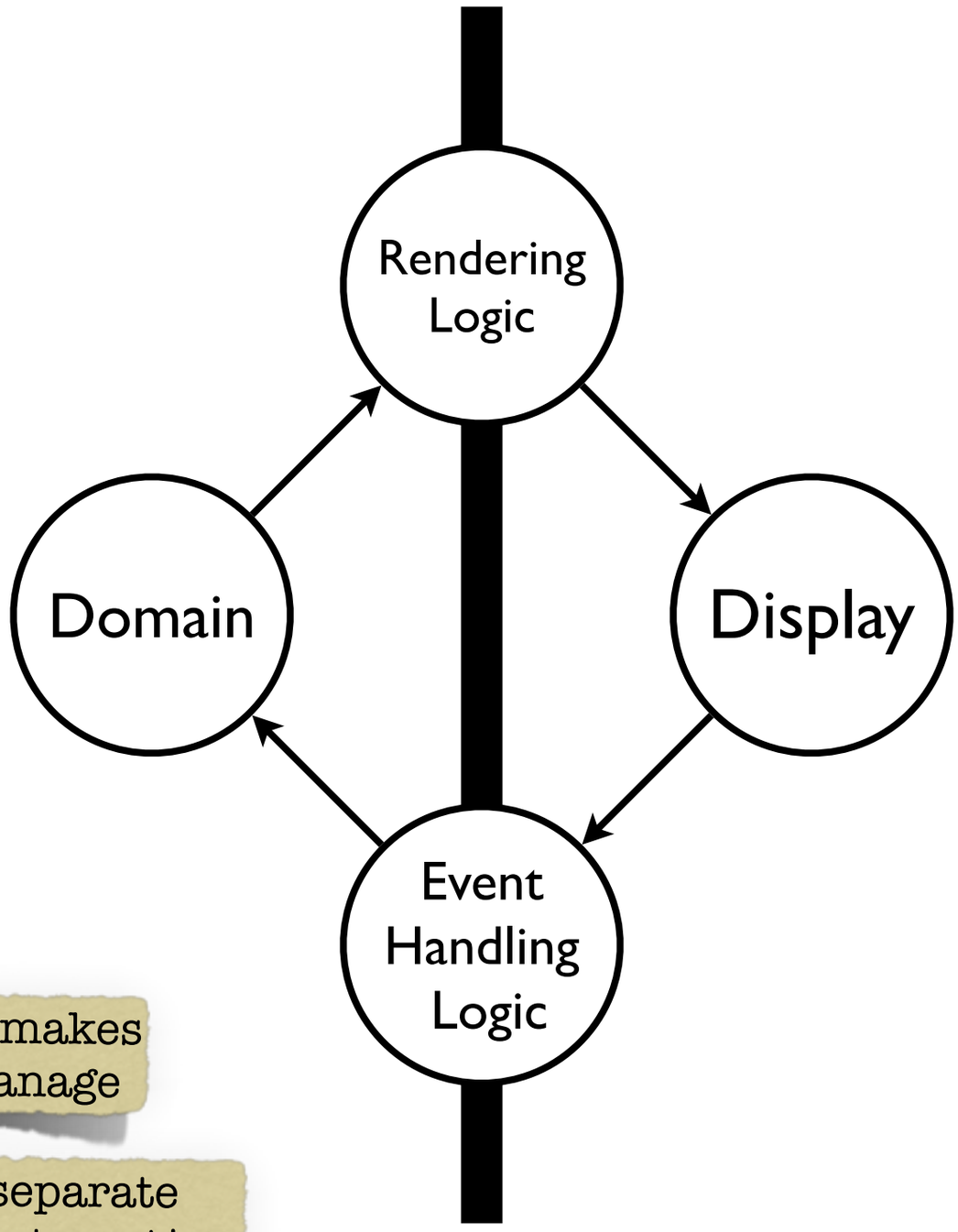

The Domain is the model, or data, or state of your application

The Display is the GUI that your user interacts with

Keeping the two apart makes your code easier to manage

But keeping them truly separate would mean they never interact!

# Model-View-Controller



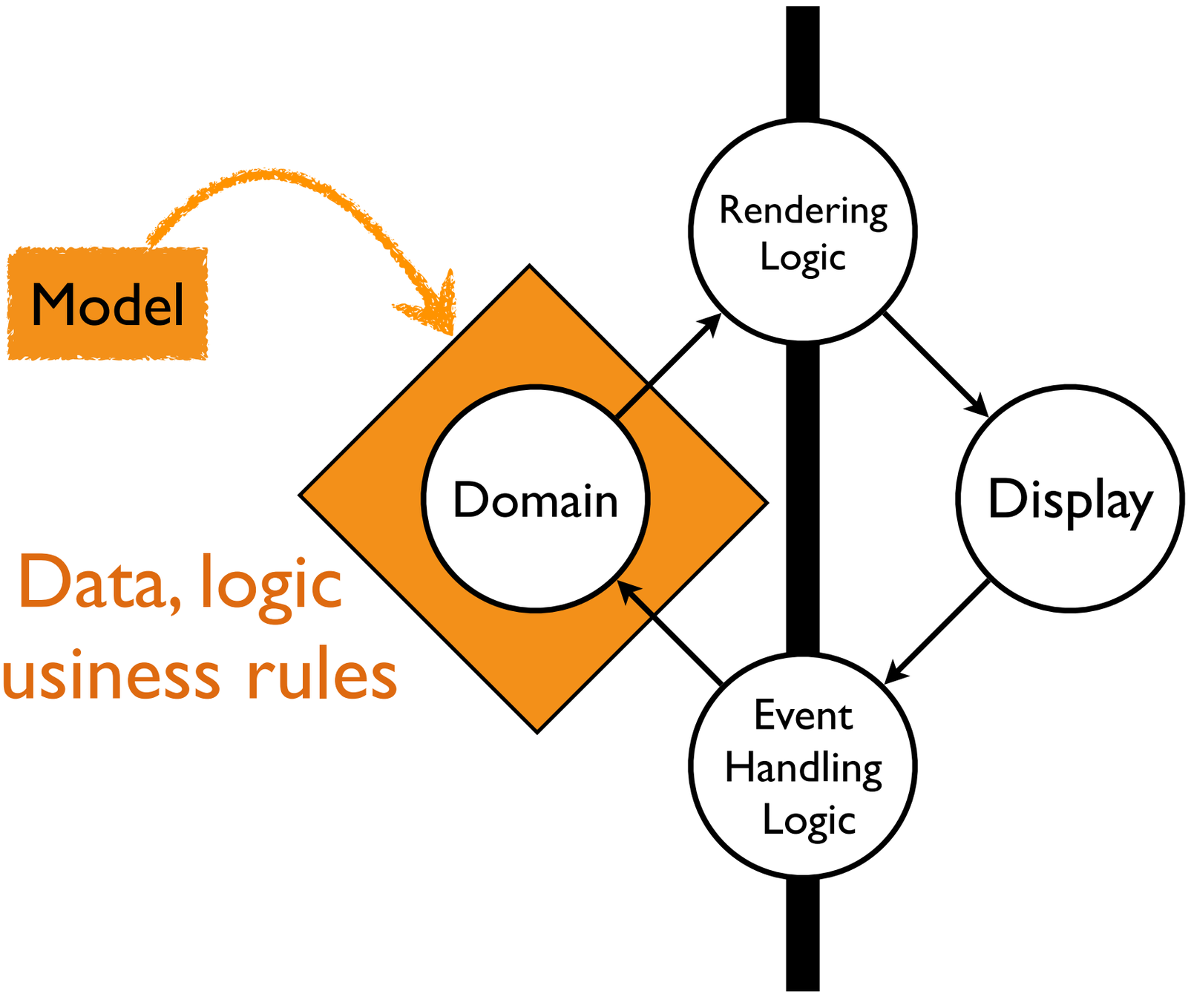

The Domain is the model, or data, or state of your application

The Display is the GUI that your user interacts with

Keeping the two apart makes your code easier to manage

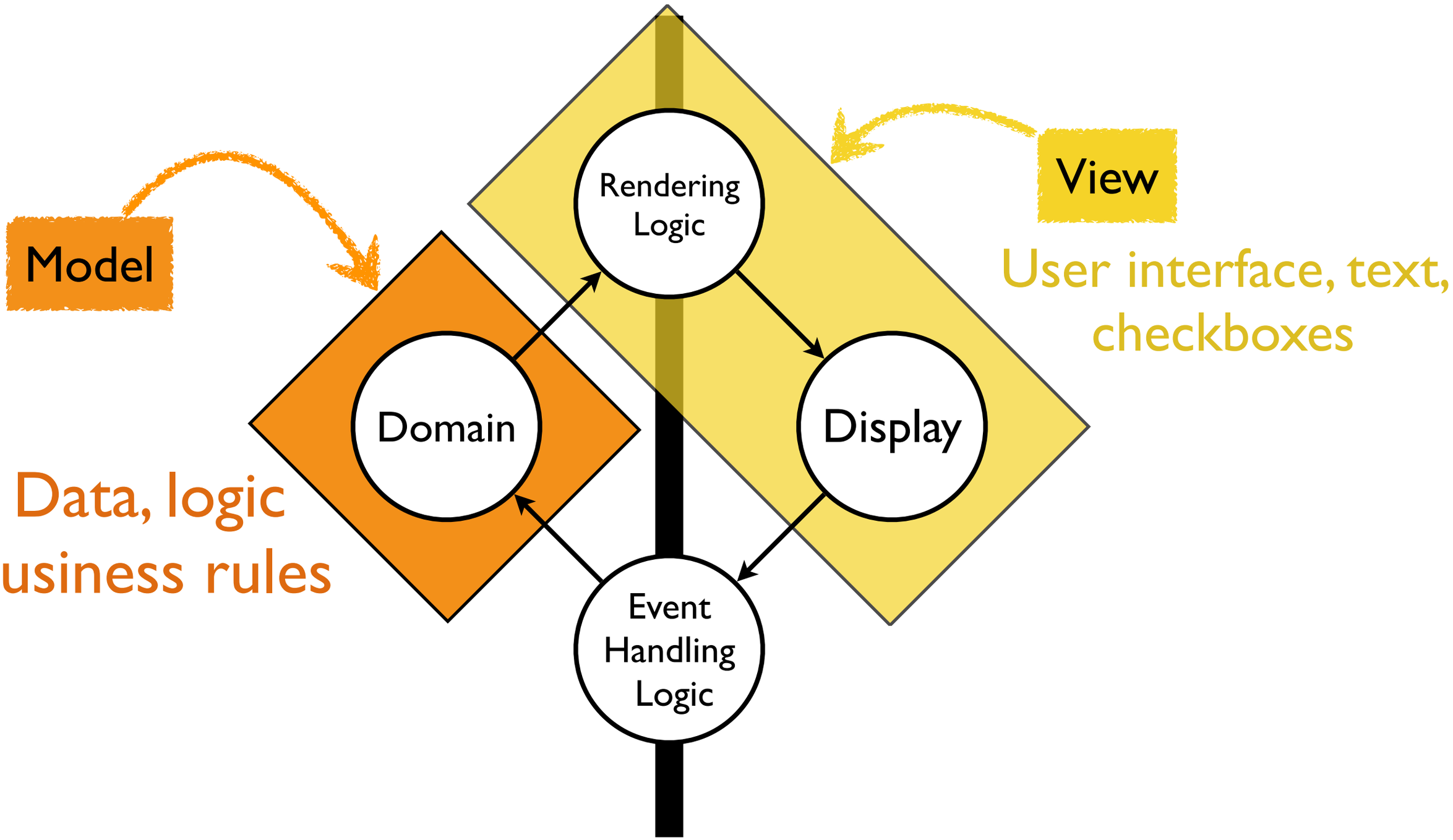
But keeping them truly separate would mean they never interact!

# Model-View-Controller

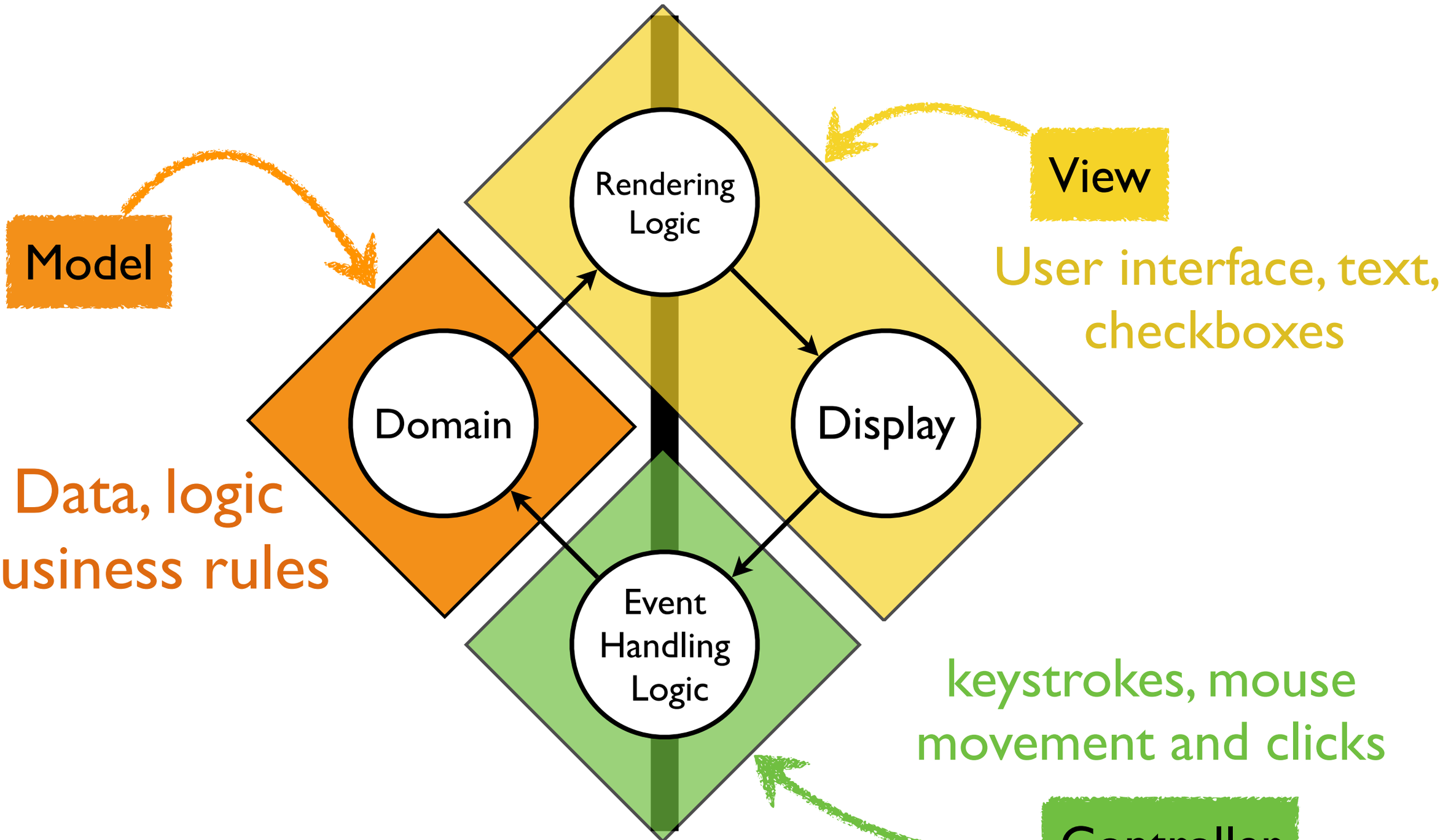



Data, logic
business rules

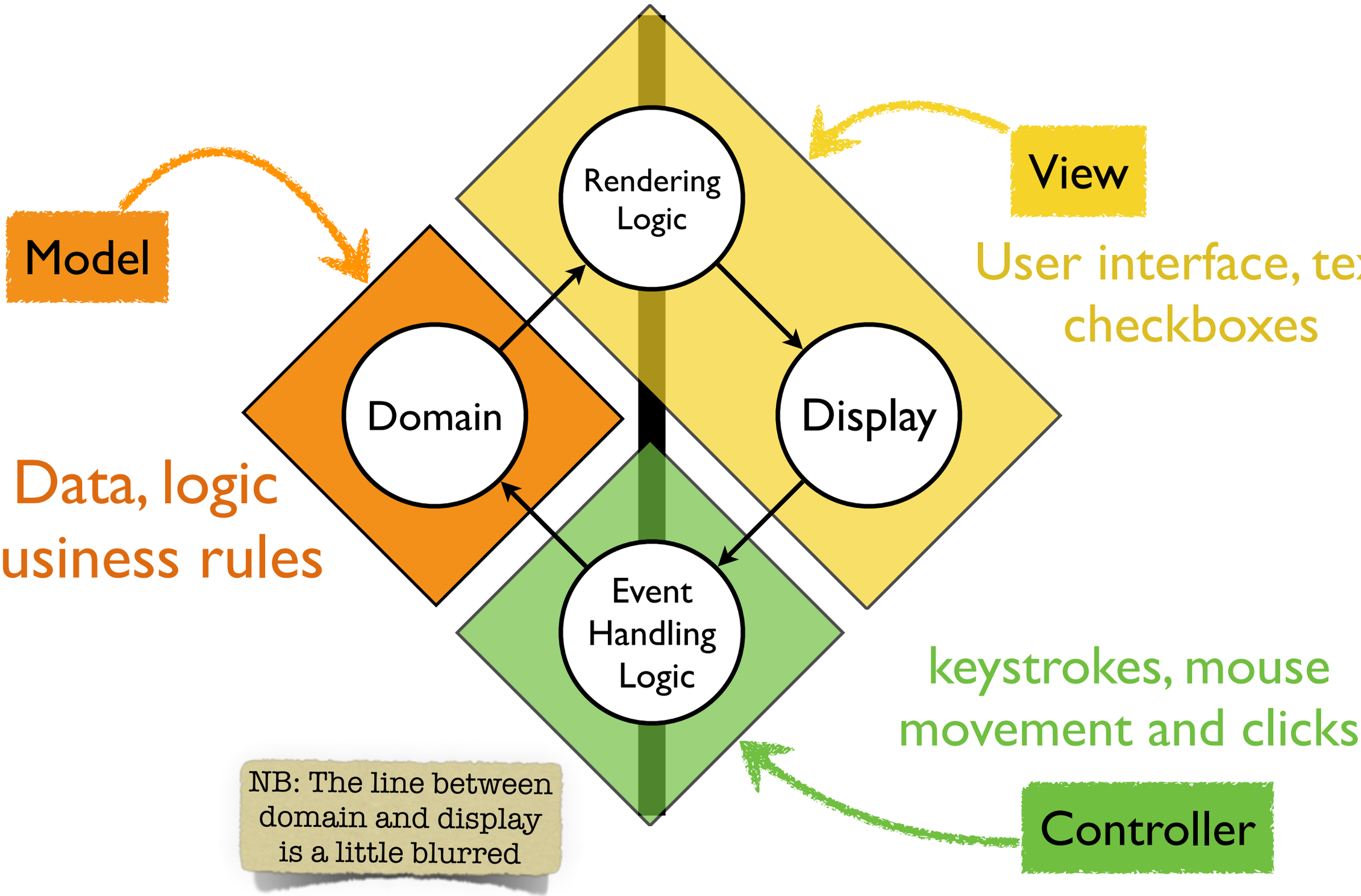
# Model-View-Controller



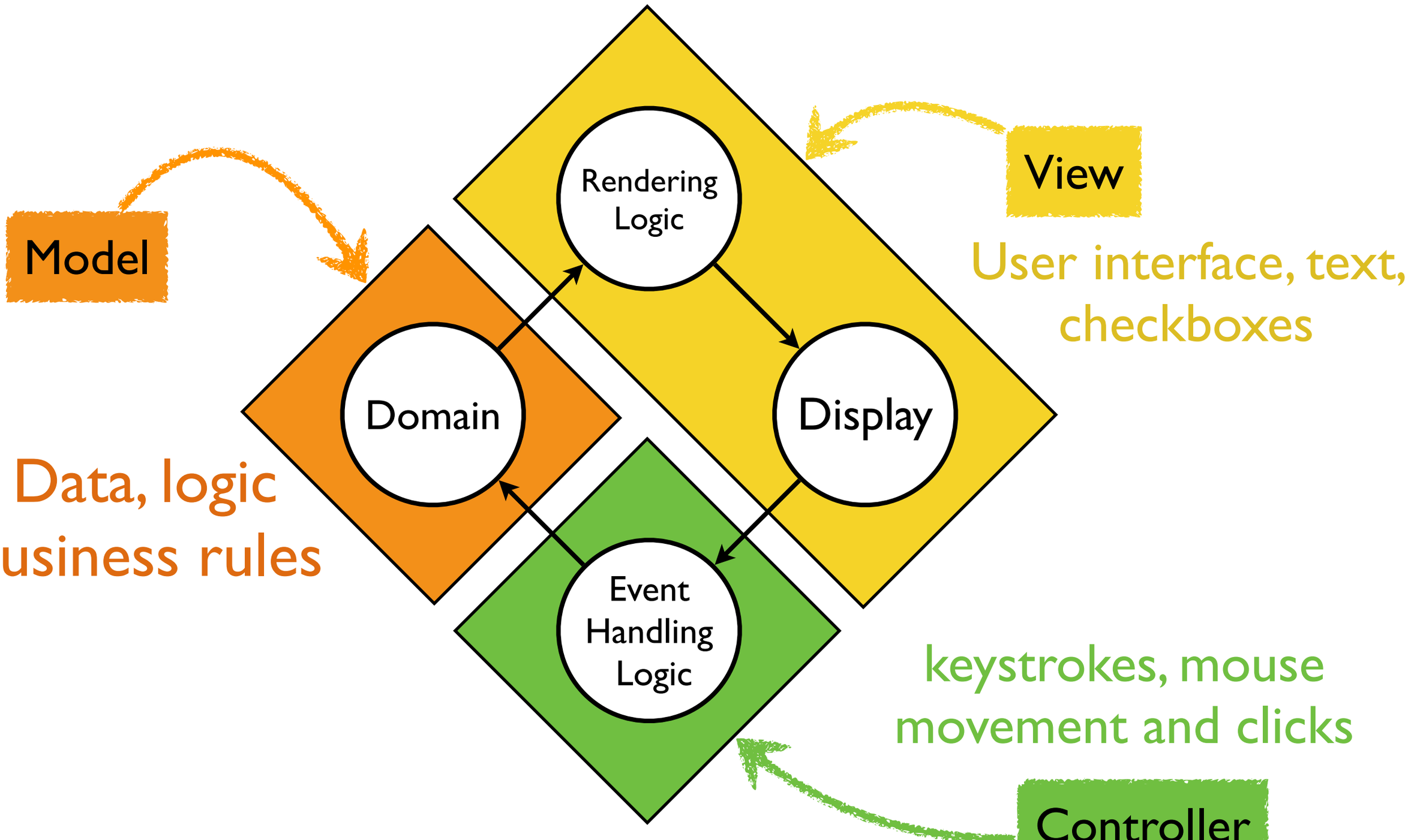
# Model-View-Controller



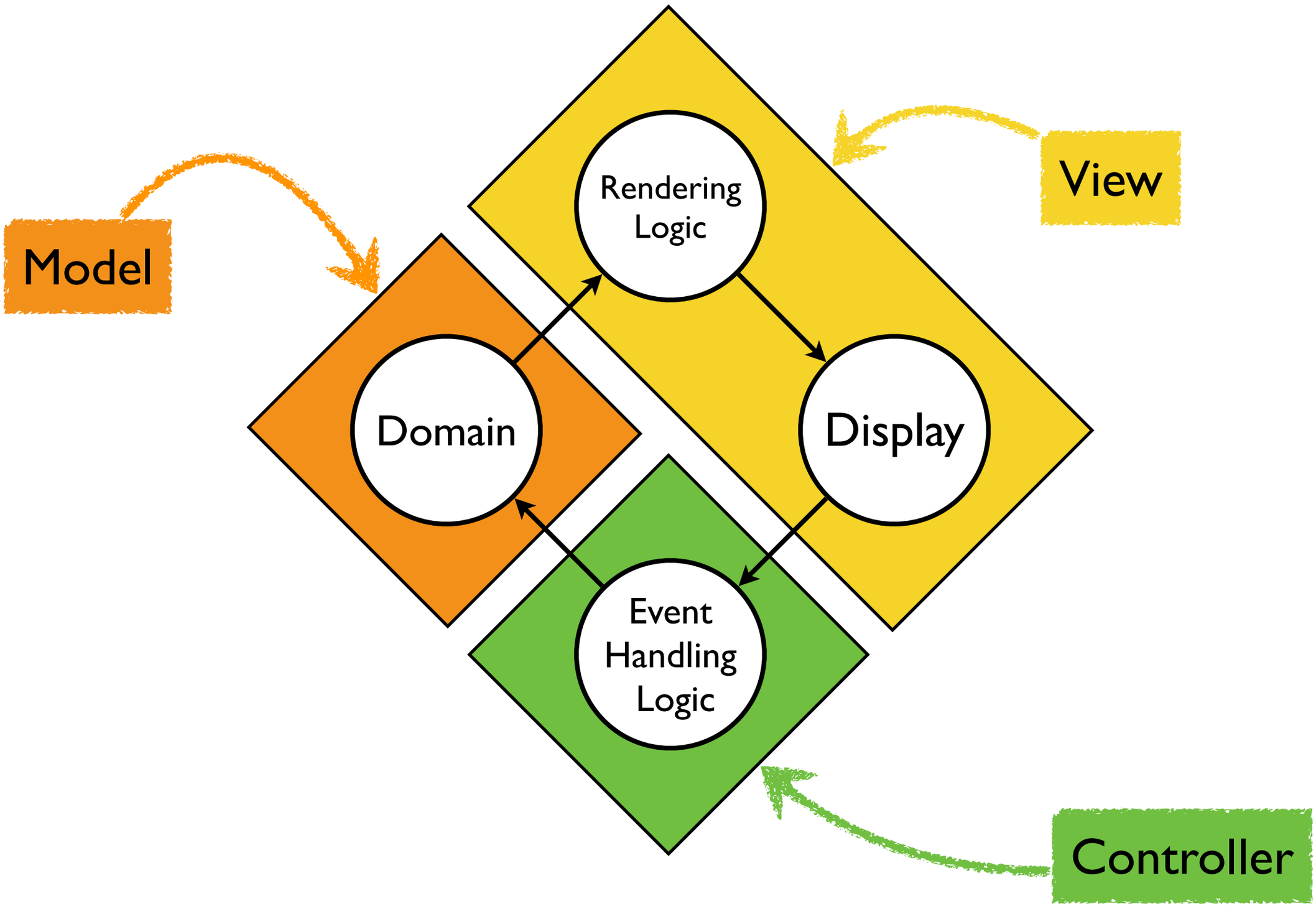
# Model-View-Controller



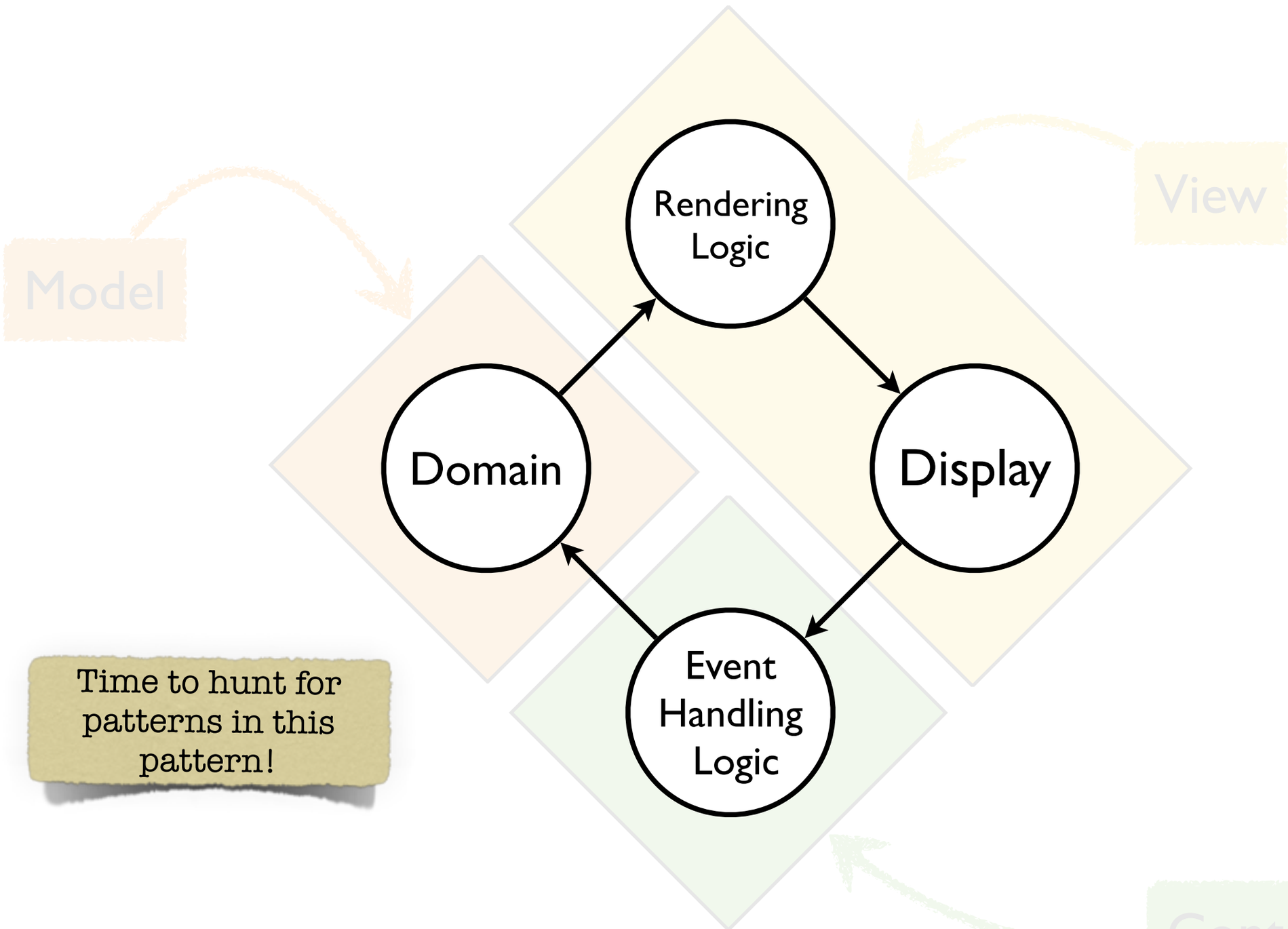
# Model-View-Controller



# Model-View-Controller



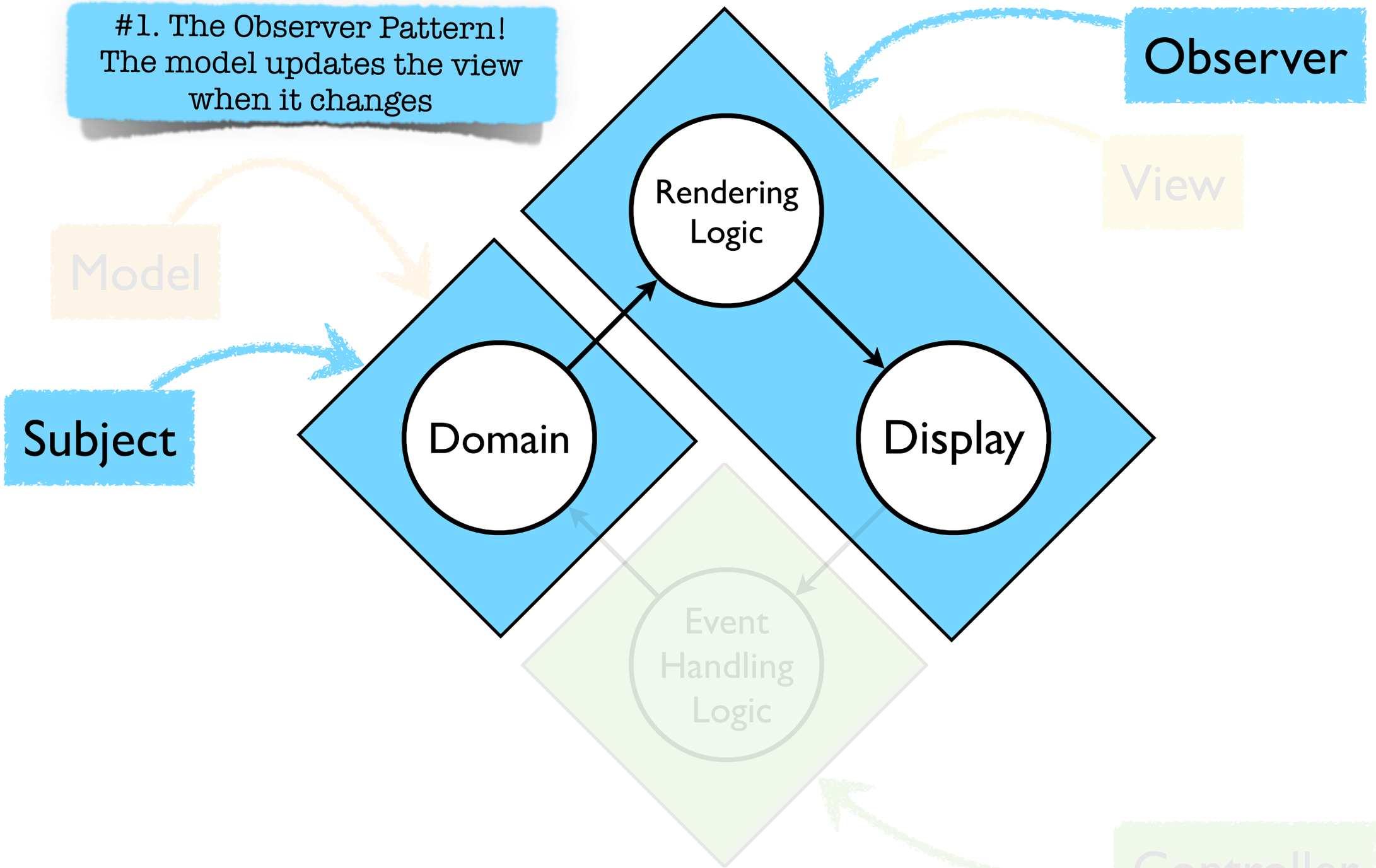
# Model-View-Controller




Time to hunt for patterns in this pattern!

# Model-View-Controller

#1. The Observer Pattern!
The model updates the view when it changes



# Model-View-Controller

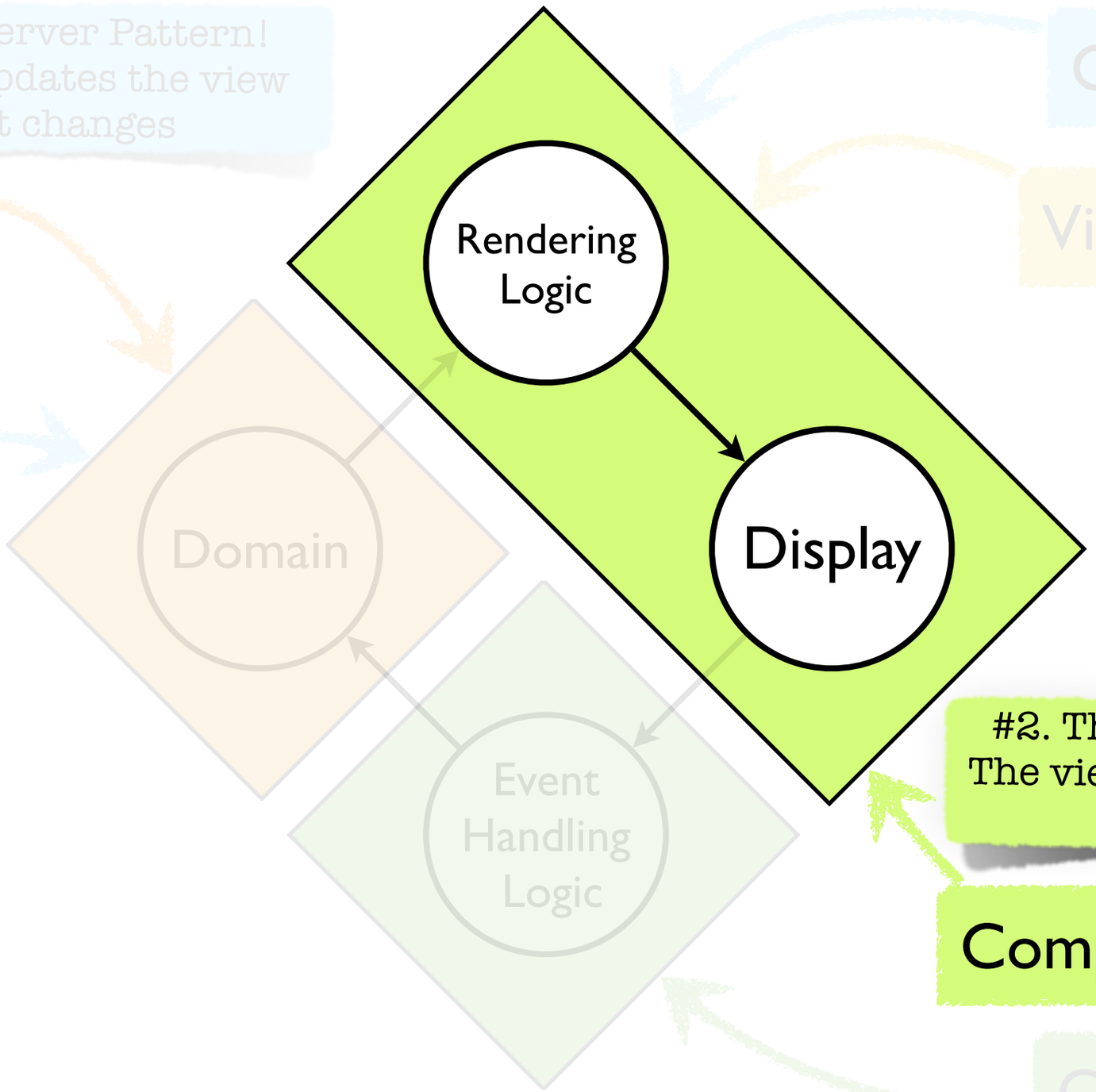
#1. The Observer Pattern!
The model updates the view
when it changes

Observer

View

Model

Subject




#2. The Composite Pattern!
The view is made up of nested
GUI elements

Composite

Controller

# Model-View-Controller

#1. The Observer Pattern!
The model updates the view when it changes

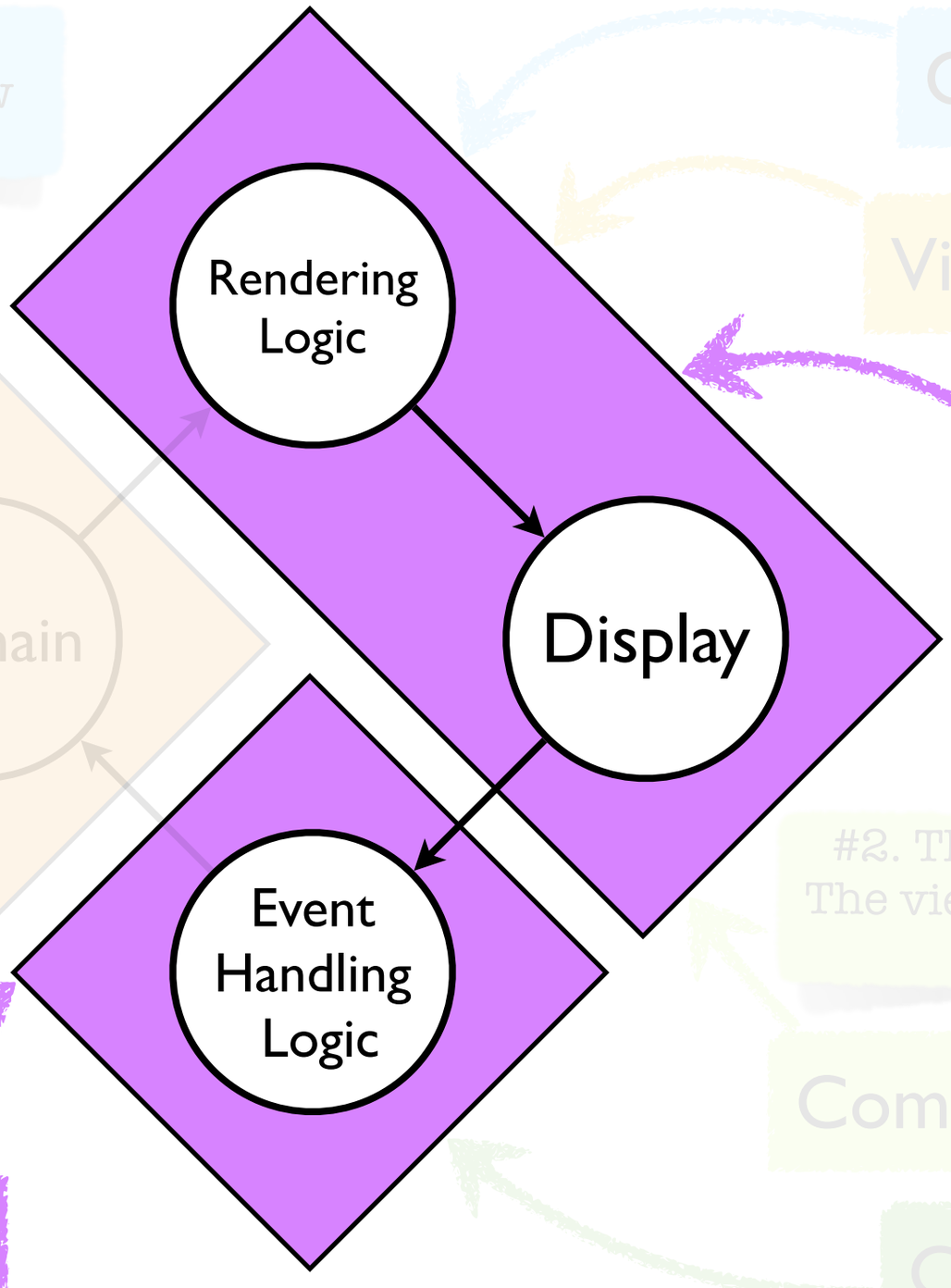
Observer

View

Model

Subject

Domain




Context

#2. The Composite Pattern!
The view is made up of nested GUI elements

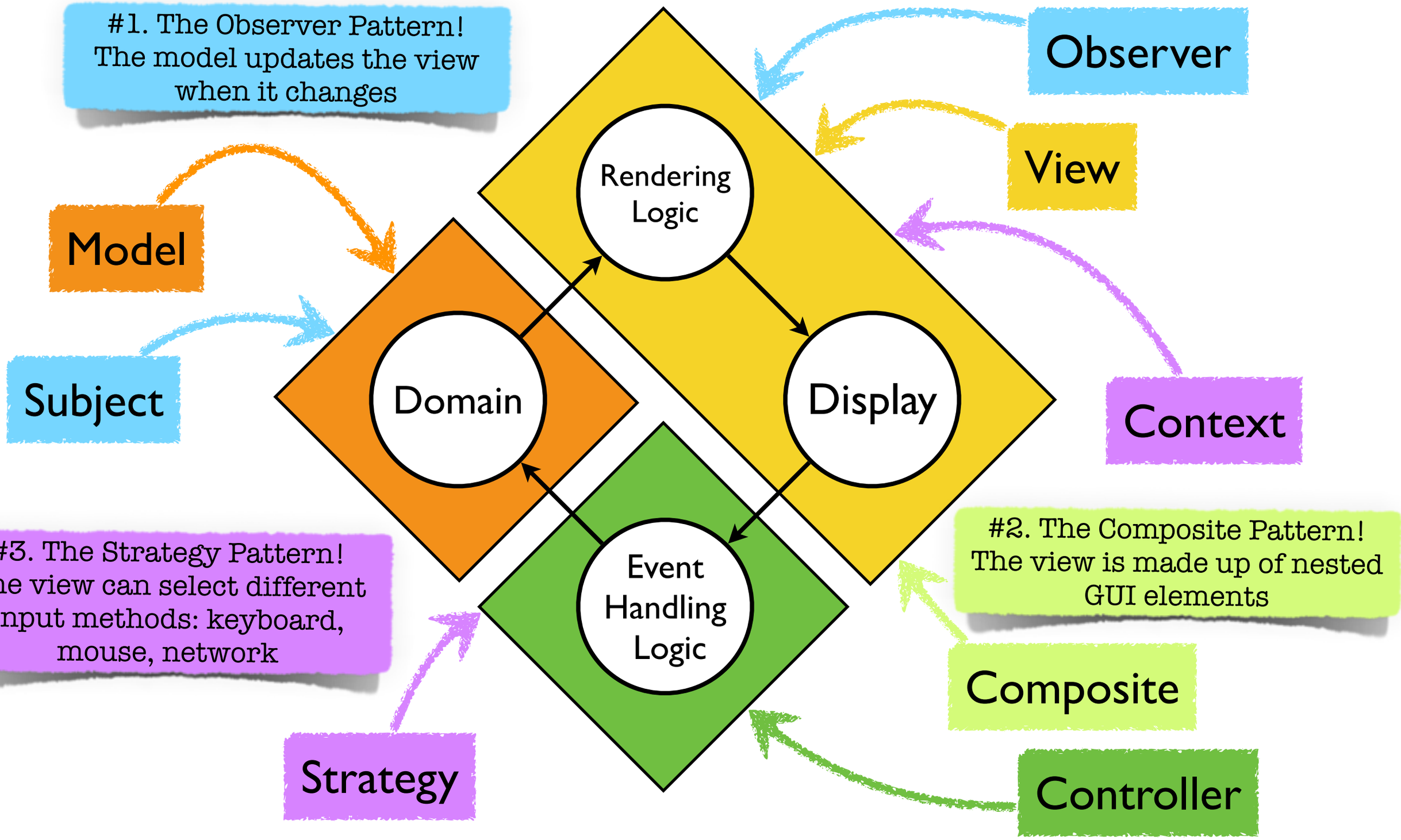
Composite

#3. The Strategy Pattern!
The view can select different input methods: keyboard, mouse, network

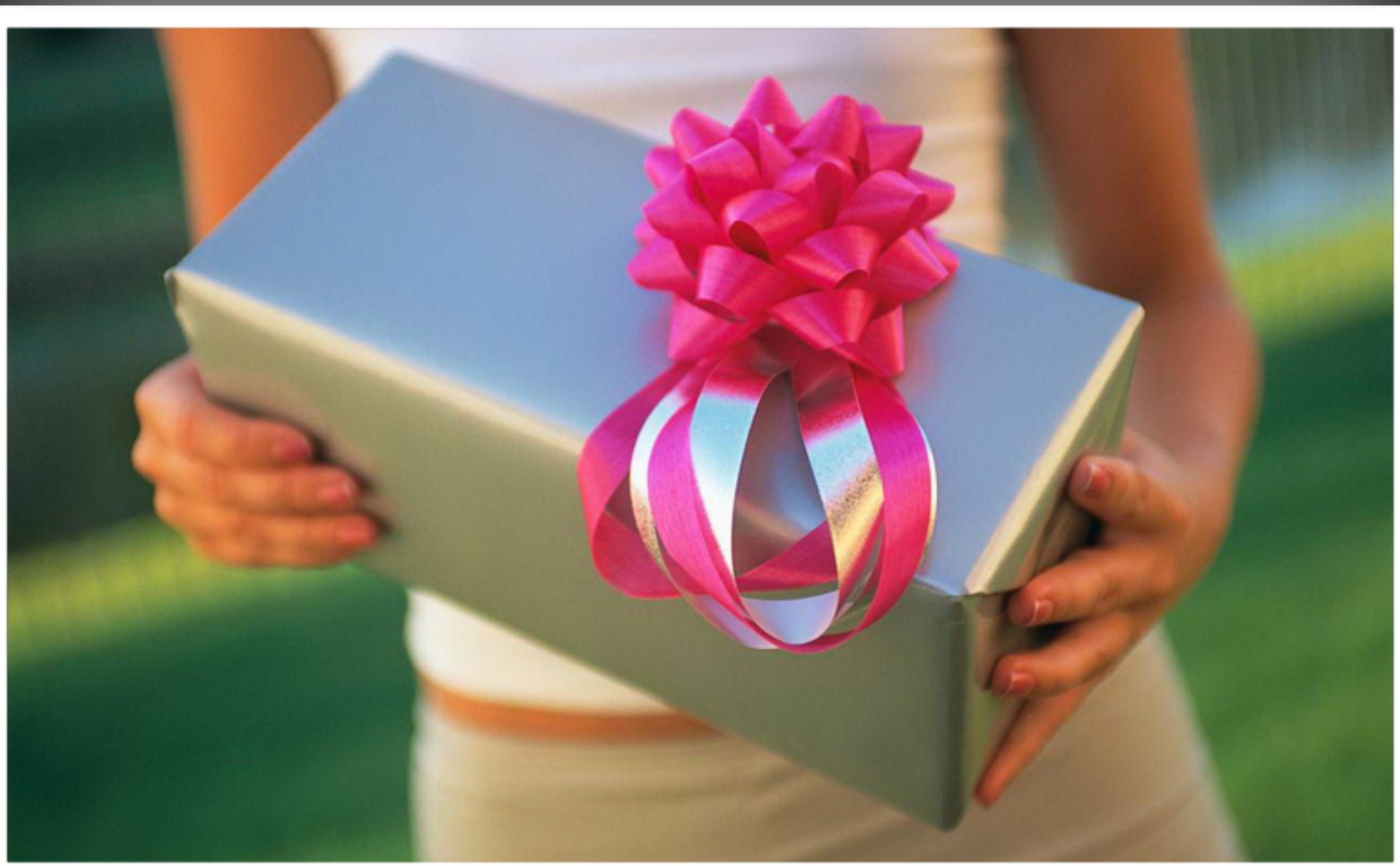
Strategy

Controller

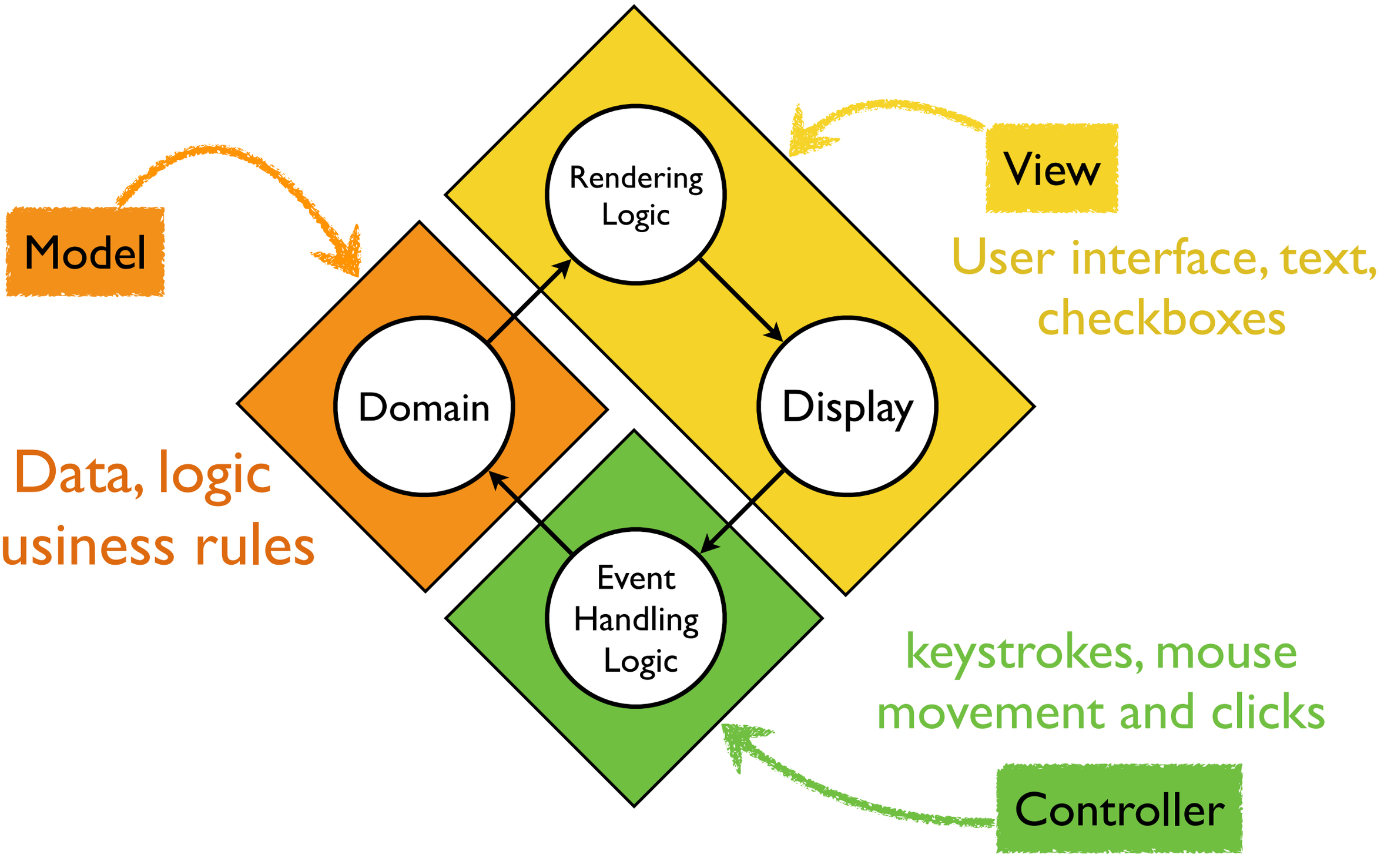
# Model-View-Controller



# Model-View-Presenter



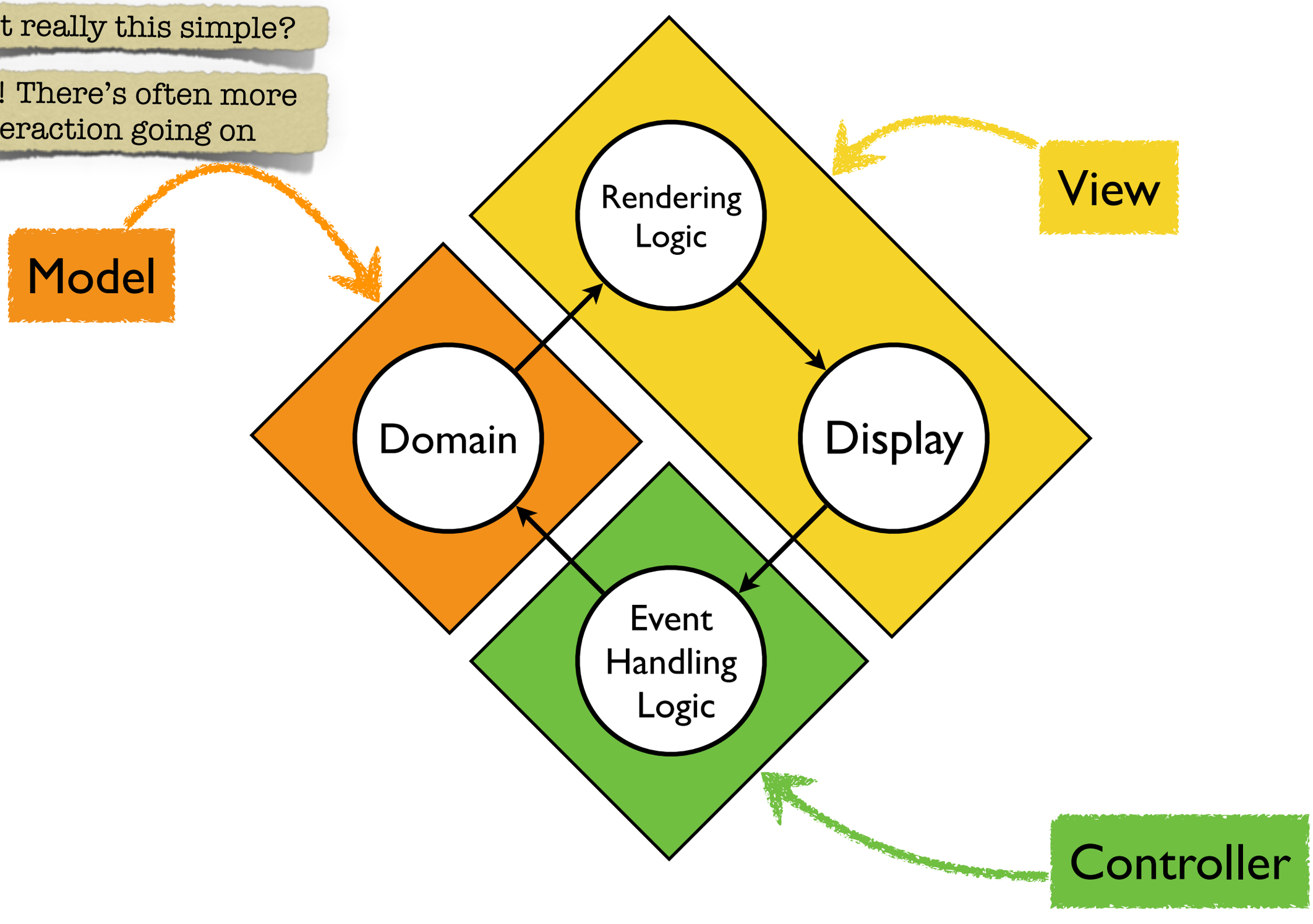
# Model-View-Controller



# Model-View-Controller

Q. Is it really this simple?

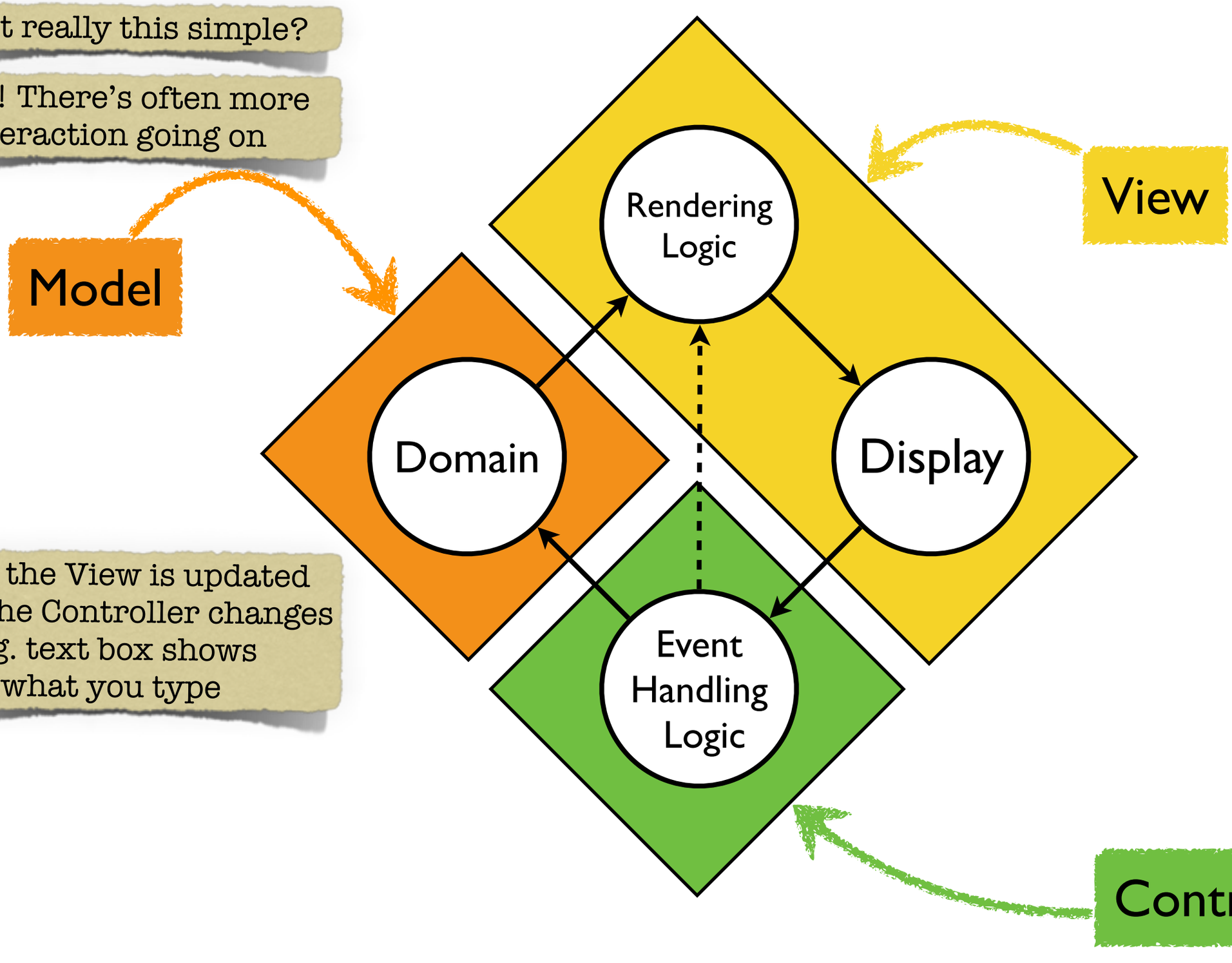
A. No! There's often more interaction going on



# Model-View-Controller

Q. Is it really this simple?

A. No! There's often more interaction going on

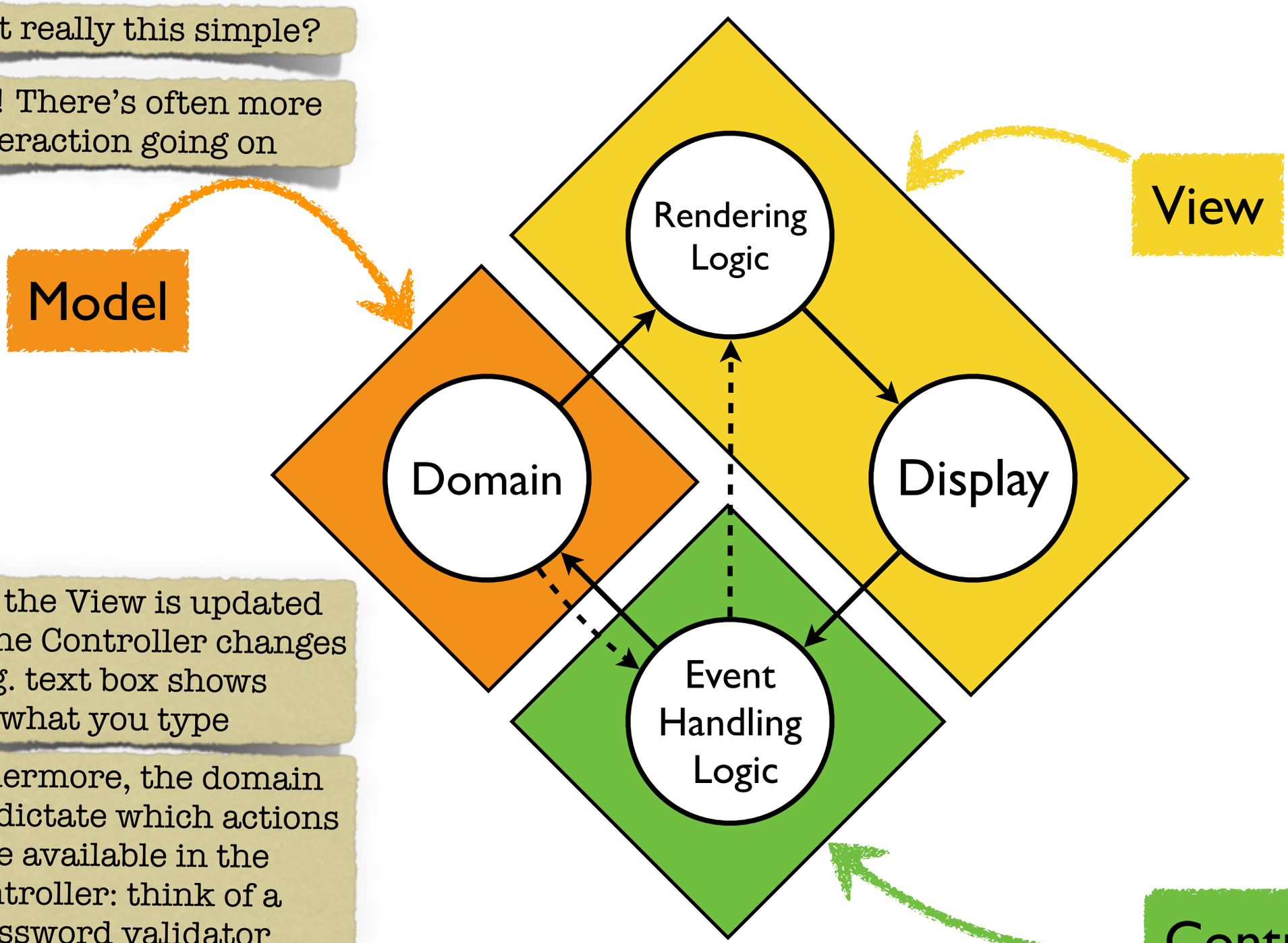



Often the View is updated when the Controller changes e.g. text box shows what you type

# Model-View-Controller

Q. Is it really this simple?

A. No! There's often more interaction going on



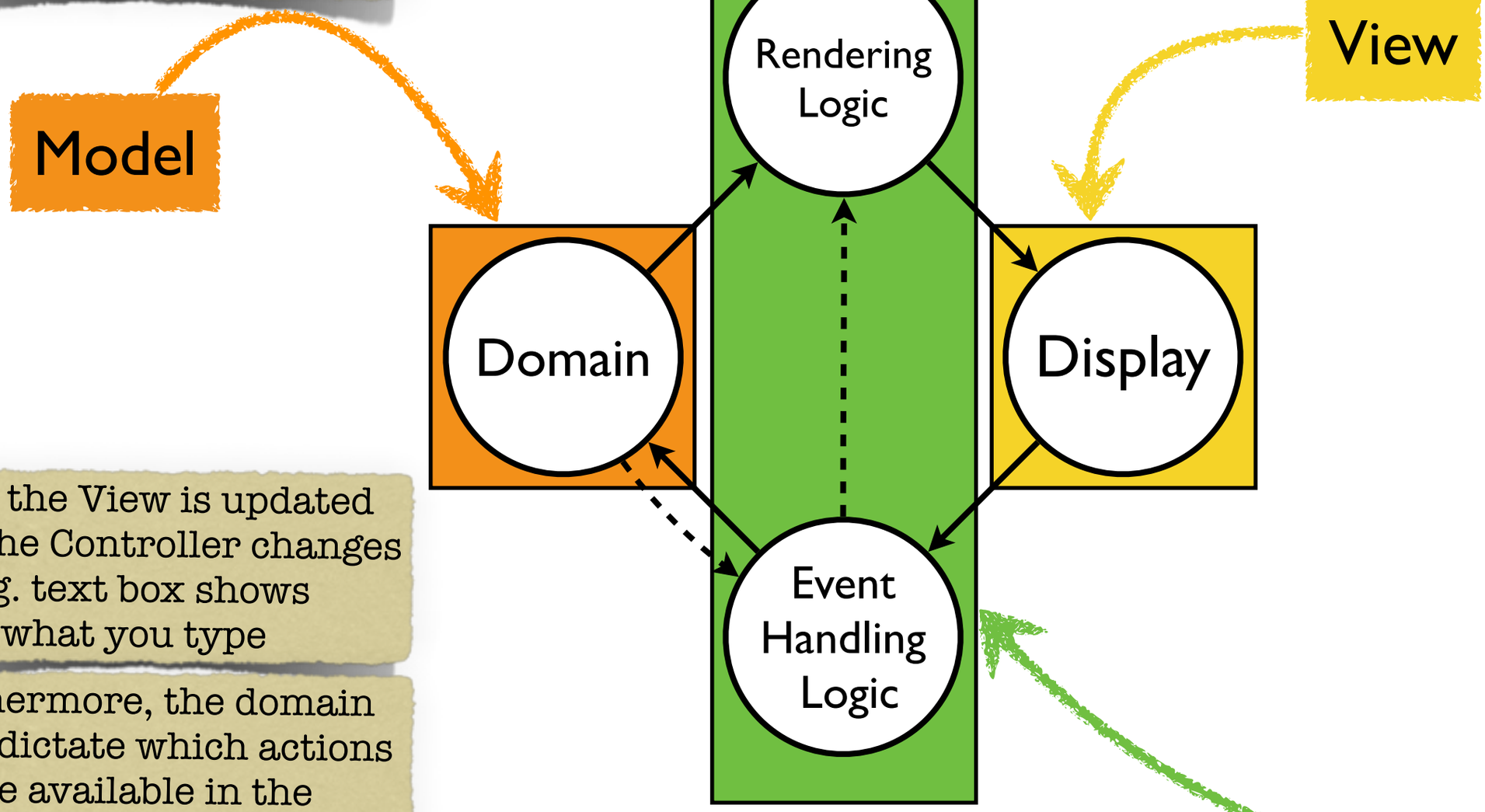

Often the View is updated when the Controller changes e.g. text box shows what you type

Furthermore, the domain might dictate which actions are available in the controller: think of a password validator

# Model-View-~~Controller~~

Q. Is it really this simple?

A. No! There's often more interaction going on



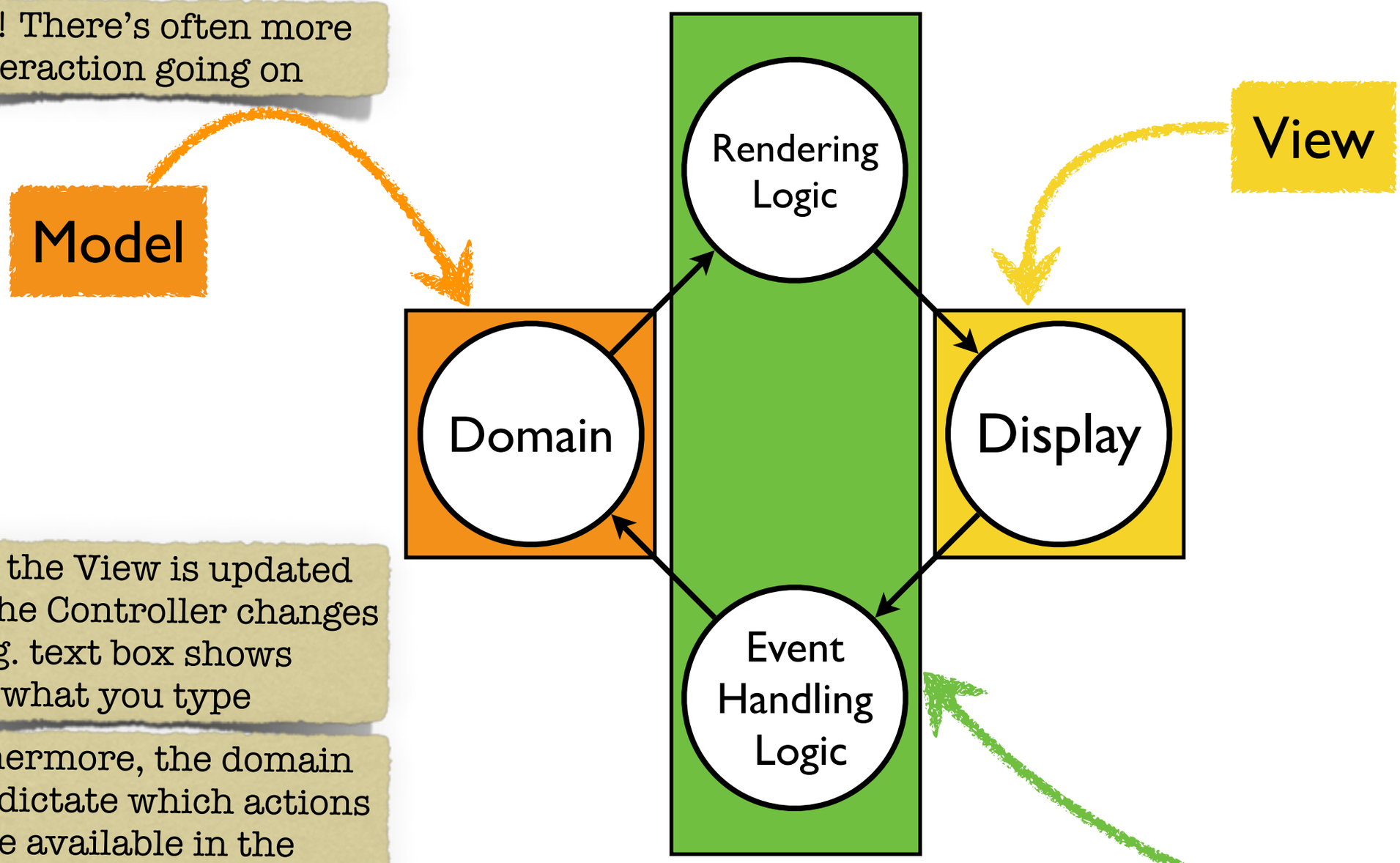

Often the View is updated when the Controller changes e.g. text box shows what you type

Furthermore, the domain might dictate which actions are available in the controller: think of a password validator

# Model-View-Presenter

Q. Is it really this simple?

A. No! There's often more interaction going on



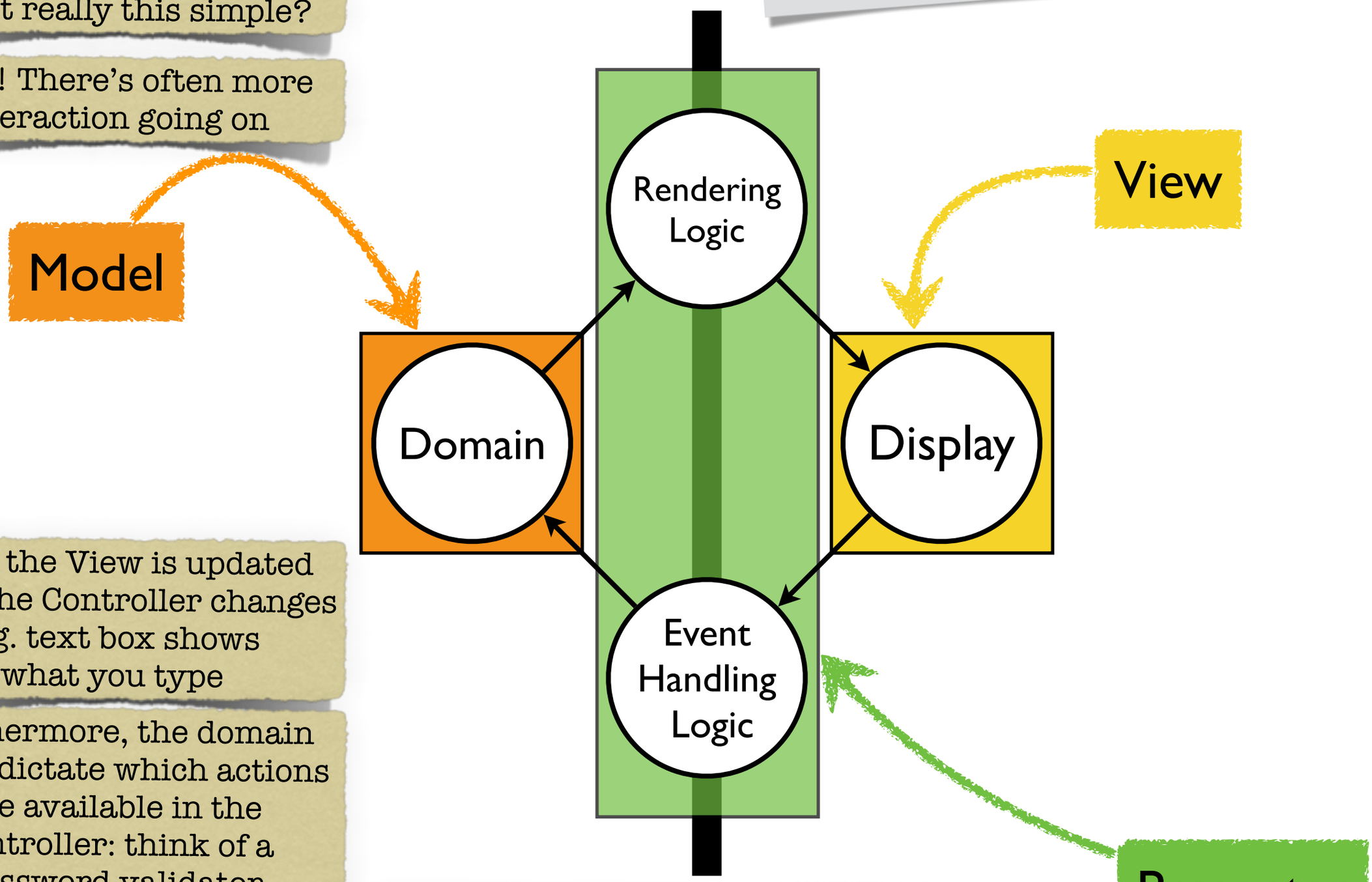

Often the View is updated when the Controller changes e.g. text box shows what you type

Furthermore, the domain might dictate which actions are available in the controller: think of a password validator

# Model-View-Presenter

Q. Is it really this simple?

A. No! There's often more interaction going on

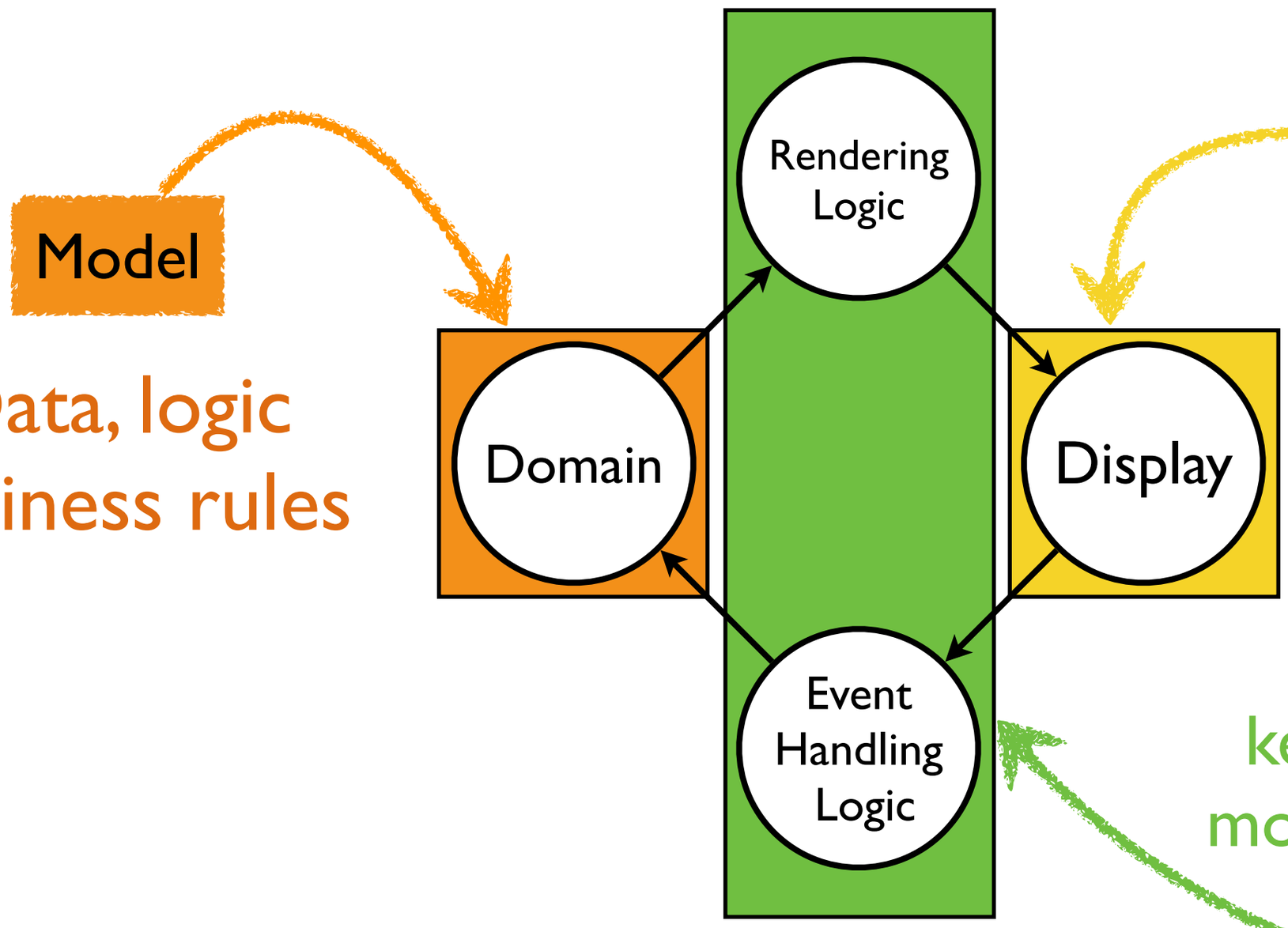



Often the View is updated when the Controller changes e.g. text box shows what you type

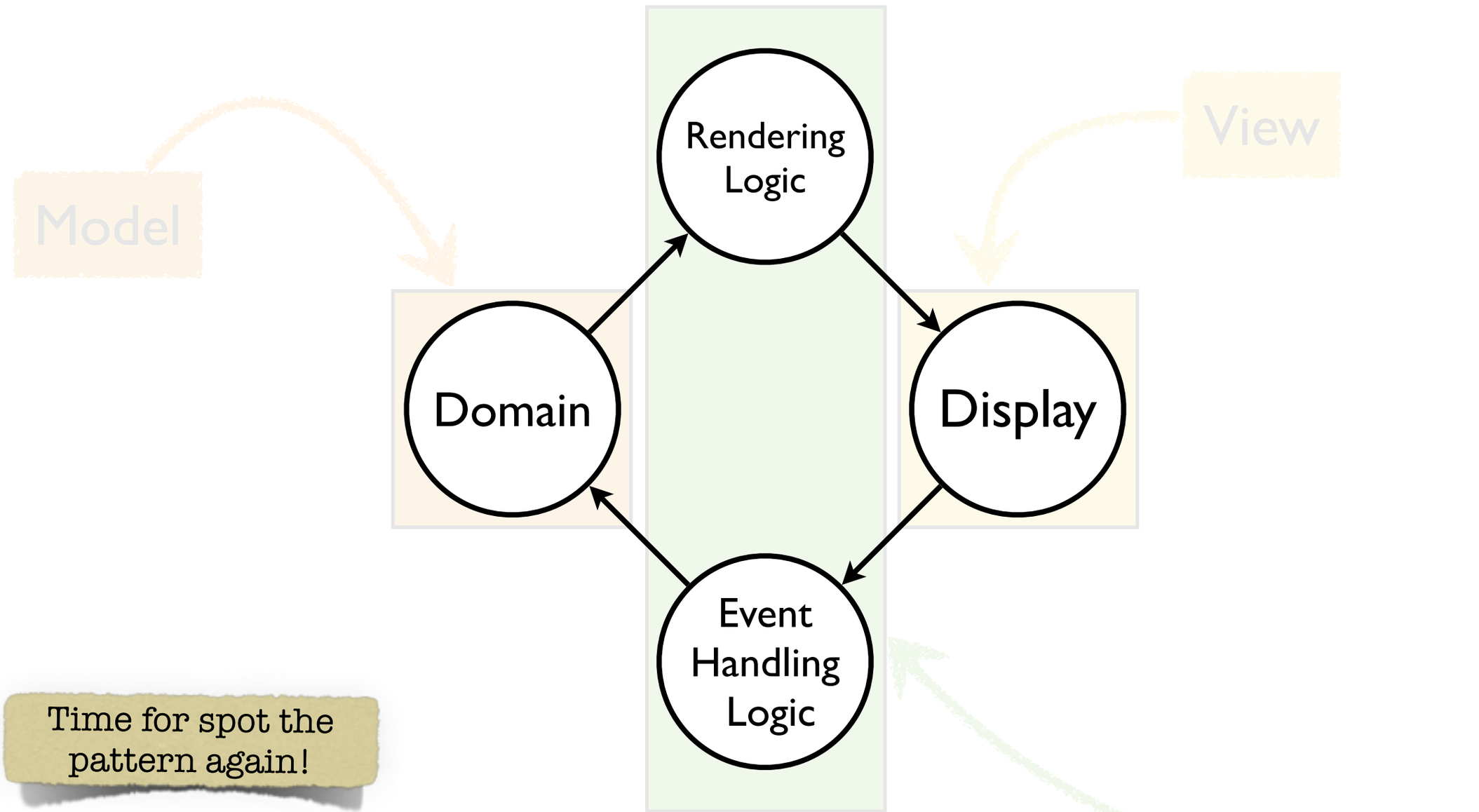
Furthermore, the domain might dictate which actions are available in the controller: think of a password validator

Our dividing line makes more sense again!

# Model-View-Presenter



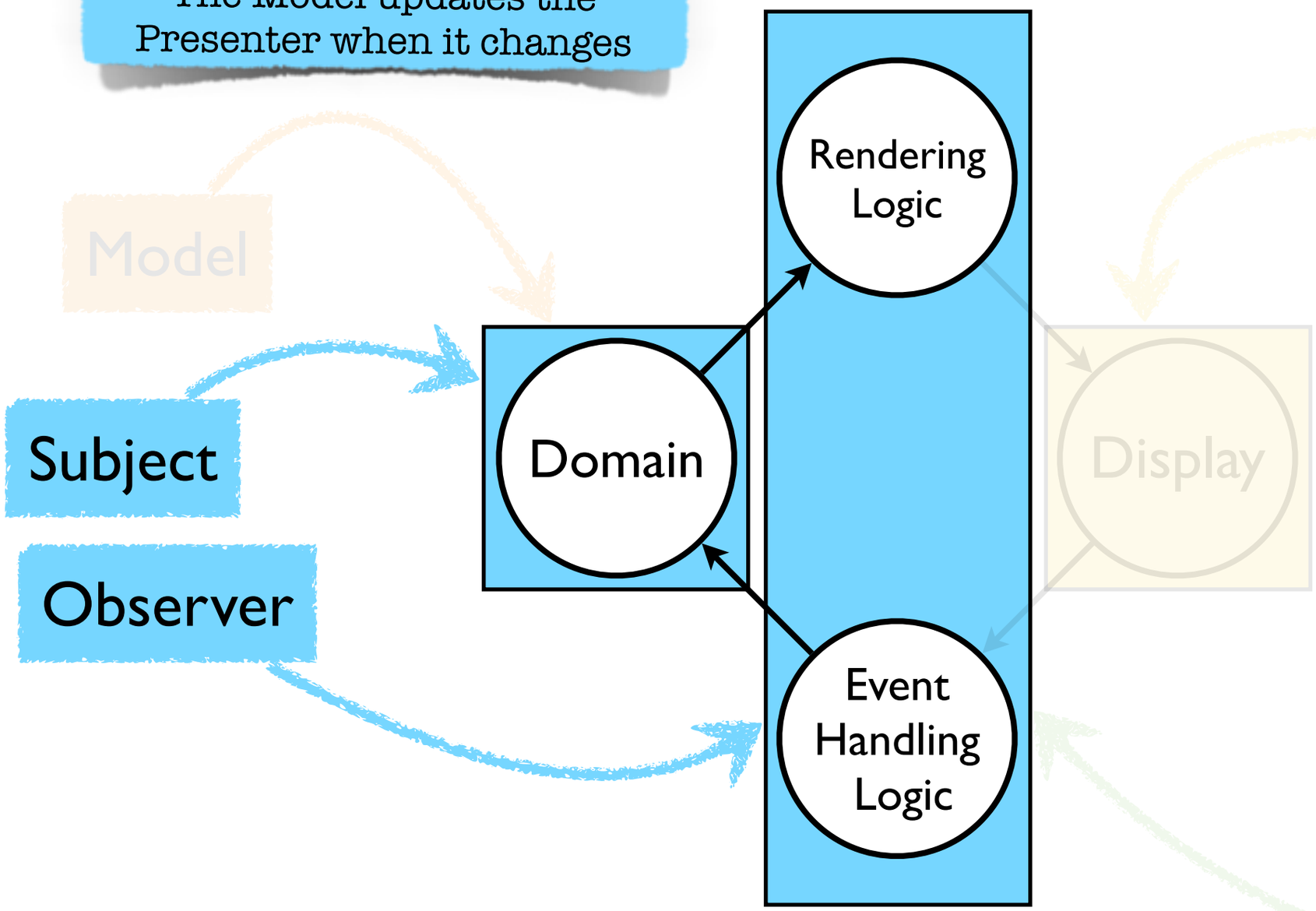
# Model-View-Presenter




Time for spot the pattern again!

# Model-View-Presenter

#1. The Observer Pattern!
The Model updates the Presenter when it changes



# Model-View-Presenter

#1. The Observer Pattern!
The Model updates the
Presenter when it changes

Model

Subject

Observer

Rendering
Logic

Domain

Event
Handling
Logic

Composite

View

Display

#2. The Composite Pattern!
The view is still made up of a
cascade of widgets

Presenter

# Model-View-Presenter

#1. The Observer Pattern!
The Model updates the Presenter when it changes

Model

Subject

Observer

Domain

Rendering Logic

Display

Event Handling Logic

Composite

View

#2. The Composite Pattern!
The view is still made up of a cascade of widgets

Subject

Observer

#3. The Observer Pattern!
The Presenter listens out for events from the View

Presenter

# Model-View-Presenter

#1. The Observer Pattern!
The Model updates the Presenter when it changes

Model

Subject

Observer

Rendering Logic

Domain

Event Handling Logic

Display

Composite

View

#2. The Composite Pattern!
The view is still made up of a cascade of widgets

Subject

Observer

#3. The Observer Pattern!
The Presenter listens out for events from the View

Presenter

# Swing

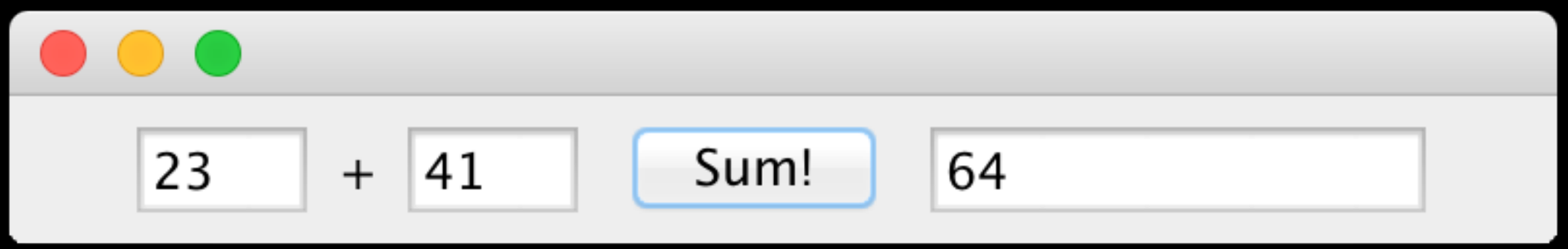
# Swing

- Swing uses elements of MVC in its design
- It implements the observer pattern through the *listener* family of classes


You'll probably stumble on the Abstract Window Toolkit (AWT) family of classes when digging around swing. This was the original windowing library for Java, and bits still hang around.

# Swing

- Implement an (Awe)sum Calculator
  - Implement a Model
  - Implement a View
  - Implement a Controller



# Swing

```
public class SumModel {
   private int sum;

   public void sum(int x, int y) {
      sum = x + y;
   }

   public int getSum() {
      return sum;
   }
}
```

The **model** is embarrassingly simple: it knows nothing except how to sum things

We could make this a **subject** in the observer pattern by allowing others to subscribe, but this is overkill for such a small example

# Swing

The **view** knows nothing of the **model** or the **controller**!

```
import java.awt.event.ActionListener;
import javax.swing.*;

public class SumView extends JFrame {
  private JTextField sumX;
  private JLabel sumLabel;
  private JTextField sumY;
  private JButton sumButton;
  private JTextField sumField;

  SumView() {
    sumX           = new JTextField(10);
    sumLabel       = new JLabel("+");
    sumY           = new JTextField(10);
    sumButton      = new JButton("Sum!");
    sumField       = new JTextField(10);
    JPanel sumPanel = new JPanel();

    this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    this.setSize(400, 80);

    sumPanel.add(sumX);
    sumPanel.add(sumLabel);
    sumPanel.add(sumY);
    sumPanel.add(sumButton);
    sumPanel.add(sumField);

    this.add(sumPanel);
  }
}
```

```
public int getSumX() {
  return Integer.parseInt(sumX.getText());
}

public int getSumY() {
  return Integer.parseInt(sumY.getText());
}

public int getSum() {
  return Integer.parseInt(sumField.getText());
}

public void setSum(int sum) {
  sumField.setText(Integer.toString(sum));
}

public void addSumListener(ActionListener listenSumButton) {
  sumButton.addActionListener(listenSumButton);
}

public void displayErrorMessage(String errorMessage) {
  JOptionPane.showMessageDialog(this, errorMessage);
}
```

Here we use **addSumListener** as the way for observers to subscribe

# Swing

```
import java.awt.event.*;

public class SumController {

  private SumView view;
  private SumModel model;

  public SumController(SumModel model, SumView view) {
    this.model = model;
    this.view = view;

    this.view.addSumListener(new SumListener());
  }

  class SumListener implements ActionListener {
    public void actionPerformed(ActionEvent actionEvent) {
      int x, y = 0;

      try {
        x = view.getSumX();
        y = view.getSumY();

        model.sum(x, y);

        view.setSum(model.getSum());
      }
      catch (NumberFormatException e) {
        view.displayErrorMessage("Enter two numbers!");
      }
    }
  }
}
```

The **controller**, I mean, **presenter**, rigs up the **view** and the **model**

It subscribes to the **view**

This **inner class** has access to all its parent features

The **notify** is implemented by **actionPerformed**, and hooks up to the **model** to ask for values to display on the **view**

There is also some feedback sent to the **view**

# Swing

We can test our application with a simple **main** that sets it all up

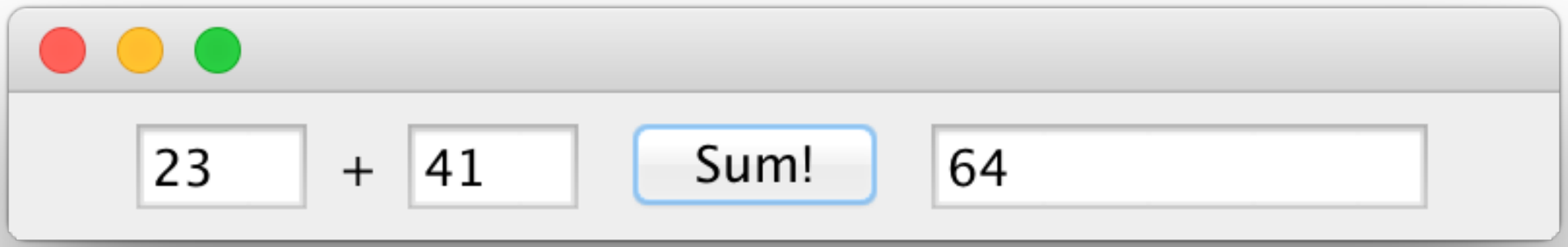
```
public class Sum {

  public static void main (String[] args) {
    SumView view   = new SumView();
    SumModel model = new SumModel();

    SumController controller = new SumController(model, view);

    view.setVisible(true);
  }
}
```

This is example is really **MVP**, in a so-called **encapsulated view style**



# Practicals


Scotland Yard
+ + + Die spannende Jagd nach Mister X quer durch London + + + Hunting Mister X + + +
Ravensburger

# Feedback