Duplicate Observed Data

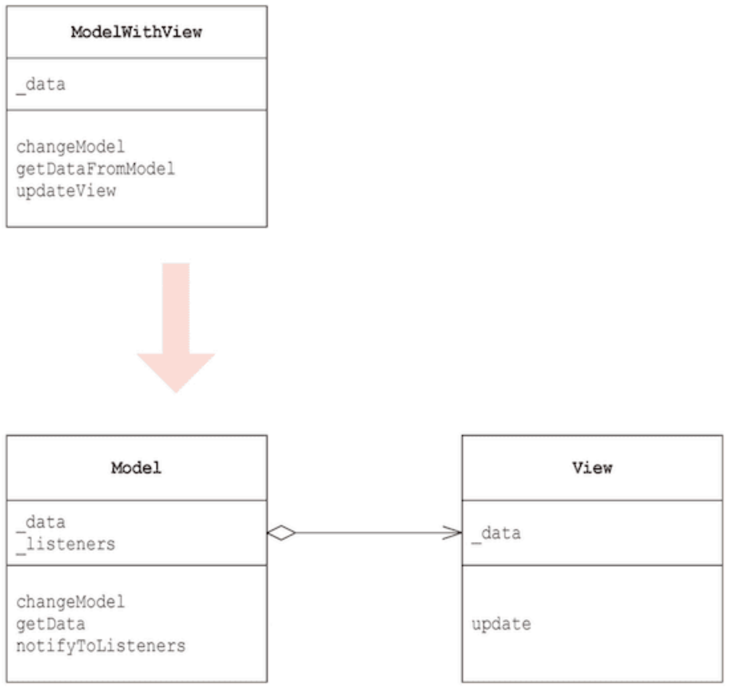
Copy **domain data** to a separate UI and buiness layer and keep it **synchronized with observers**.

- When a model and view are in the same class, we should separate them.
- Then, we should synchronize the model and view using the Observer design pattern.

For MVVM architecture (ASE456), we can think of this as:

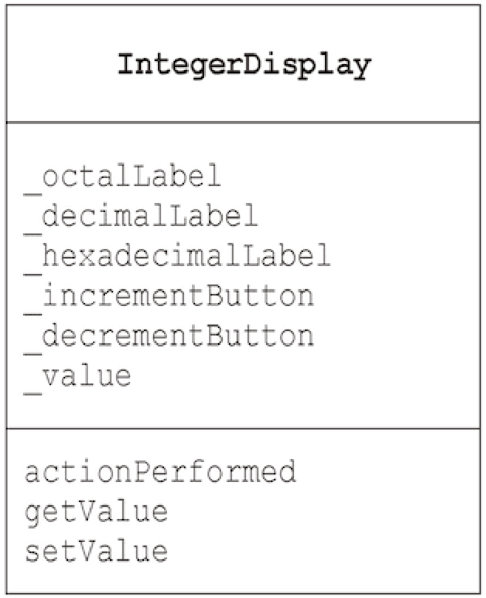
- When a ViewModel and view are in the same class, we should separate them.
- Then, we should synchronize the model and view using state manager, such as Provider.

- The Model owns the view.
- The view duplicates observed view (data).



Example: IntegerDisplay

- In this example, we build a GUI app that displays an integer value.



- This application has two responsibilities in one class.
    - The model (data): the information process in hex, oct, decimal format.
    - The view: the display of the information

```
def increment(self): self.set_value(self.value + 1)

def decrement(self): self.set_value(self.value - 1)

def get_value(self): return self.value

def set_value(self, value):
    self.value = value
    # Update labels with different bases
     # Remove '0o' prefix
    self.octal_label.setText(oct(self.value)[2:])
    # Remove '0x' prefix
    self.decimal_label.setText(str(self.value))
        self.hexadecimal_label.setText(hex(self.value)[2:])
```

- The data and view are tightly integrated so it is hard to fix bugs or extend features.
- We need to apply SRP rule to separate the class into model and view.

Refactoring: Separating Value

- The first step is to make the Value class that stores the value.

```
class Value:
    def __init__(self, value=0):
        self.value = value; self.listeners = []

    def set_value(self, value):
        self.value = value; self.notify_listeners()

    def get_value(self): return self.value
```

- The value should notify listeners when the value is updated.

```python
class Value:
    ...
    def add_value_listener(self, listener):
        self.listeners.append(listener)
    def remove_value_listener(self, listener):
        if listener in self.listeners:
            self.listeners.remove(listener)
            return True
        return False
    def notify_listeners(self):
        event = ValueChangeEvent(self)
        for listener in self.listeners:
            listener.value_changed(event)
```

- The ValueChangeEvent is used as an argument sent from the model (source) to the listeners including the View.

```python
class ValueChangeEvent:
    def __init__(self, source):
        self.source = source

    def get_source(self):
        return self.source
```

ValueListener

- We make the Listener interface.

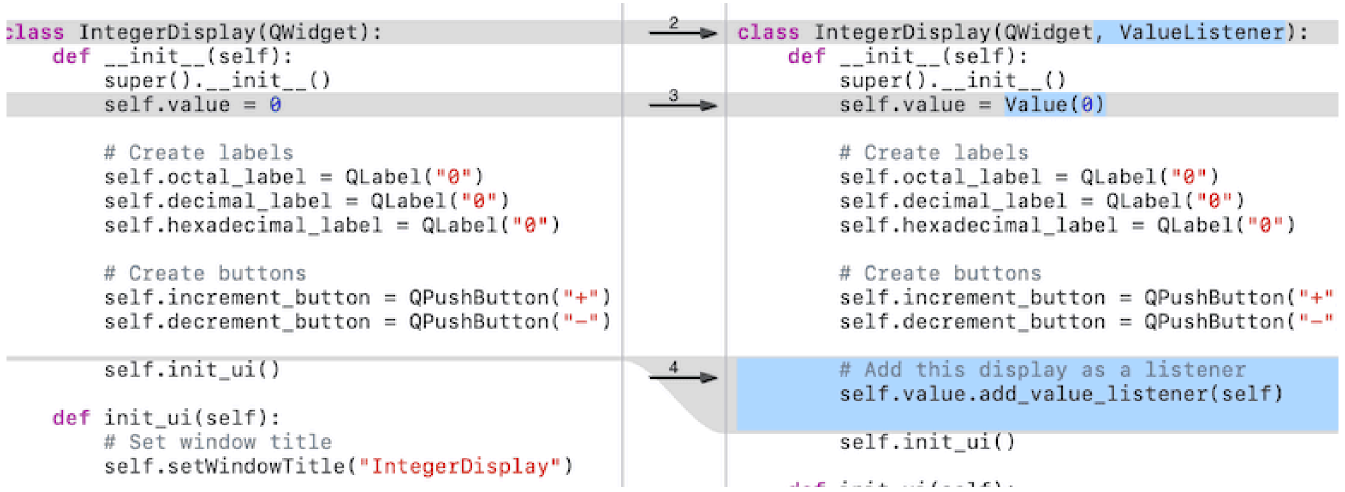
```
from abc import abstractmethod

class ValueListener:
    """Interface for value change listeners"""

    @abstractmethod
    def value_changed(self, event):
        pass
```

Link the value using the listener

- The View has no responsibility about the Model.
- Instead, it becomes the listener of the value to display the updated value.

```python
class IntegerDisplay(QWidget, ValueListener):
    def __init__(self):
        super().__init__()
        self.value = Value(0)
```

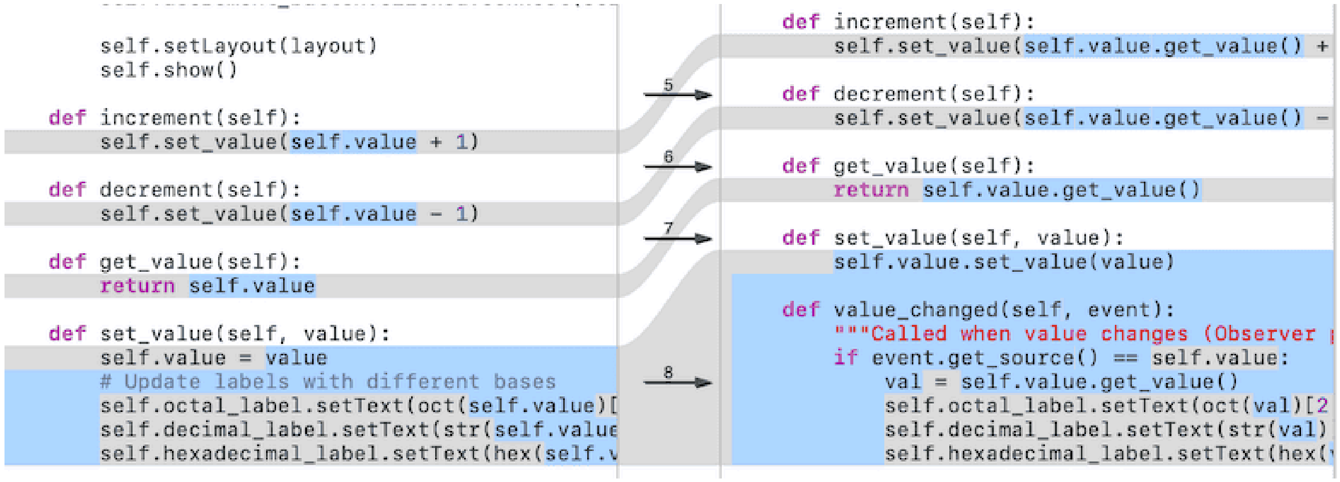
- Instead of direct access to the value, we use the value object.
- The View is a listener of the Model.



```
class IntegerDisplay(QWidget):
    def __init__(self):
        super().__init__()
        self.value = 0

        # Create labels
        self.octal_label = QLabel("0")
        self.decimal_label = QLabel("0")
        self.hexadecimal_label = QLabel("0")

        # Create buttons
        self.increment_button = QPushButton("+")
        self.decrement_button = QPushButton("-")

        self.init_ui()

    def init_ui(self):
        # Set window title
        self.setWindowTitle("IntegerDisplay")
```

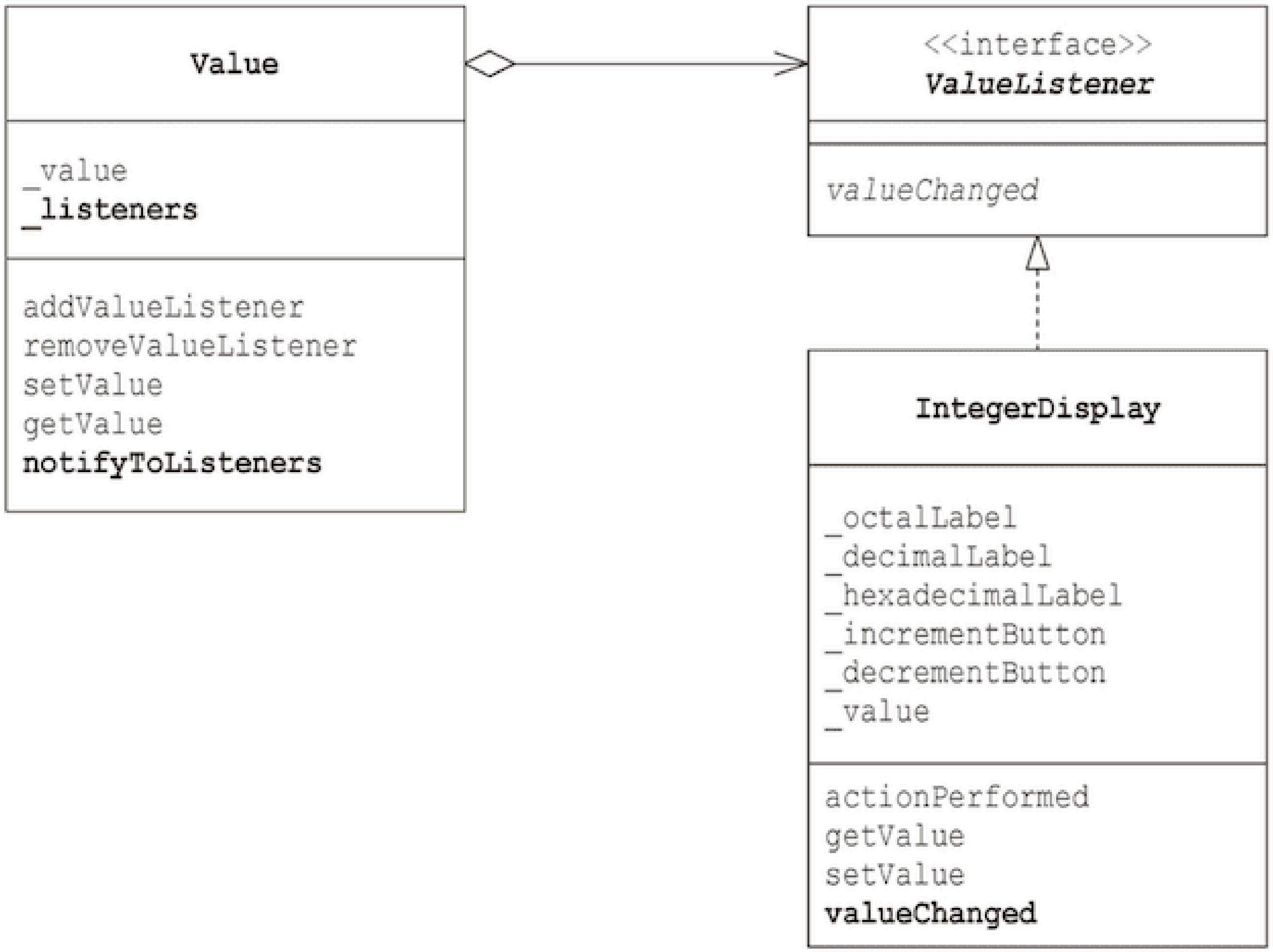
```
class IntegerDisplay(QWidget, ValueListener):
    def __init__(self):
        super().__init__()
        self.value = Value(0)

        # Create labels
        self.octal_label = QLabel("0")
        self.decimal_label = QLabel("0")
        self.hexadecimal_label = QLabel("0")

        # Create buttons
        self.increment_button = QPushButton("+"
        self.decrement_button = QPushButton("-"

        # Add this display as a listener
        self.value.add_value_listener(self)

        self.init_ui()
```

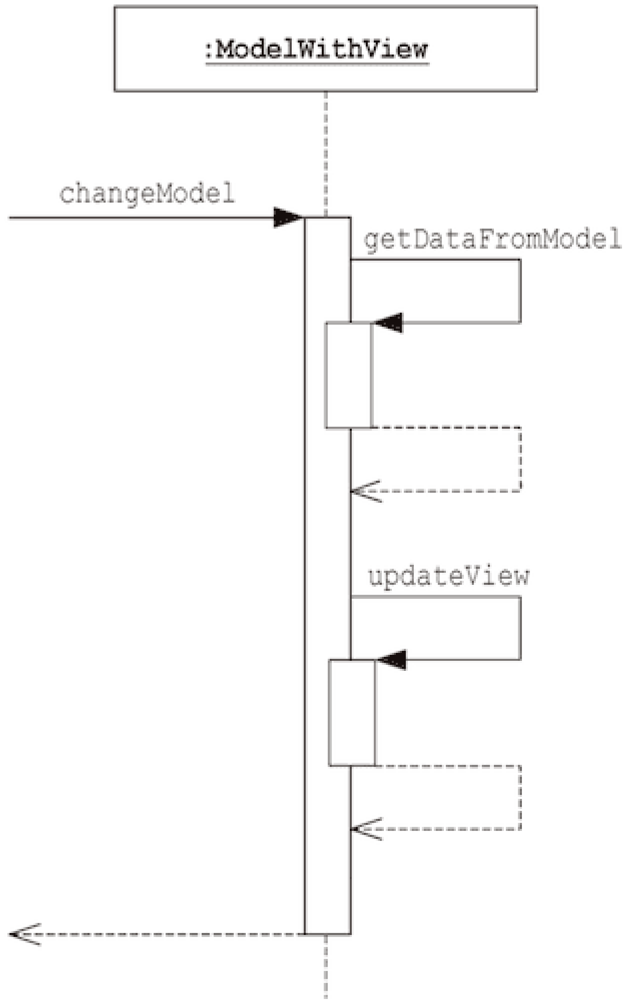
- When the Model is changed, the value_changed method is invoked using the Observer design pattern.



```
        self.setLayout(layout)
        self.show()

    def increment(self):
        self.set_value(self.value + 1)

    def decrement(self):
        self.set_value(self.value - 1)

    def get_value(self):
        return self.value

    def set_value(self, value):
        self.value = value
        # Update labels with different bases
        self.octal_label.setText(oct(self.value)[
        self.decimal_label.setText(str(self.value
        self.hexadecimal_label.setText(hex(self.v
```

```
    def increment(self):
        self.set_value(self.value.get_value() +

    def decrement(self):
        self.set_value(self.value.get_value() -

    def get_value(self):
        return self.value.get_value()

    def set_value(self, value):
        self.value.set_value(value)

    def value_changed(self, event):
        """Called when value changes (Observer
        if event.get_source() == self.value:
            val = self.value.get_value()
            self.octal_label.setText(oct(val)[2
            self.decimal_label.setText(str(val)
            self.hexadecimal_label.setText(hex(
```


Value
_value
_listeners
addValueListener
removeValueListener
setValue
getValue
notifyToListeners
<<interface>>
ValueListener
valueChanged
IntegerDisplay
_octalLabel
_decimalLabel
_hexadecimalLabel
_incrementButton
_decrementButton
_value
actionPerformed
getValue
setValue
valueChanged


:ModelWithView
changeModel
getDataFromModel
updateView

:Model
:View
changeModel

Example: Adding the Graph Listener

With the refactored design, it is easy to extend the feature.

- In this example, we make the Graph application that displays the value in a different form.

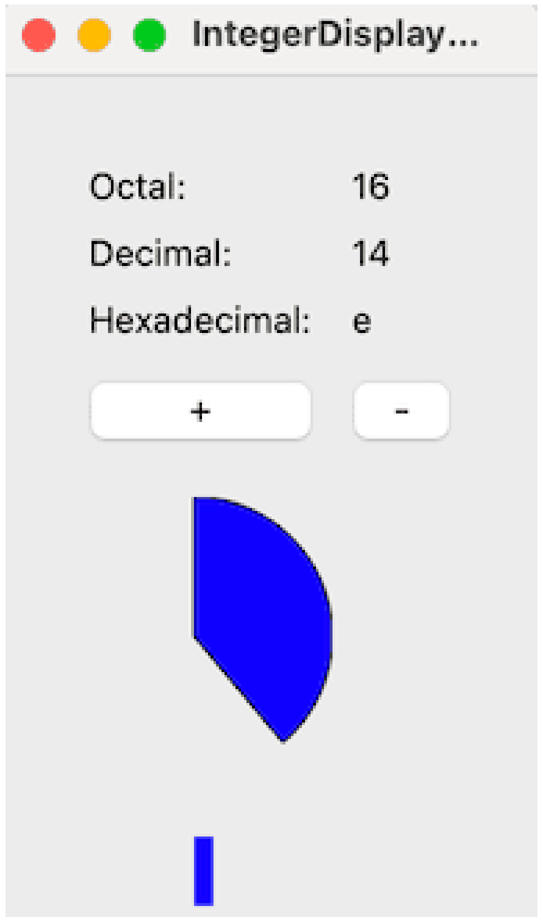
Graph Listener

- We need to add the new view by subclassing the ValueListener.

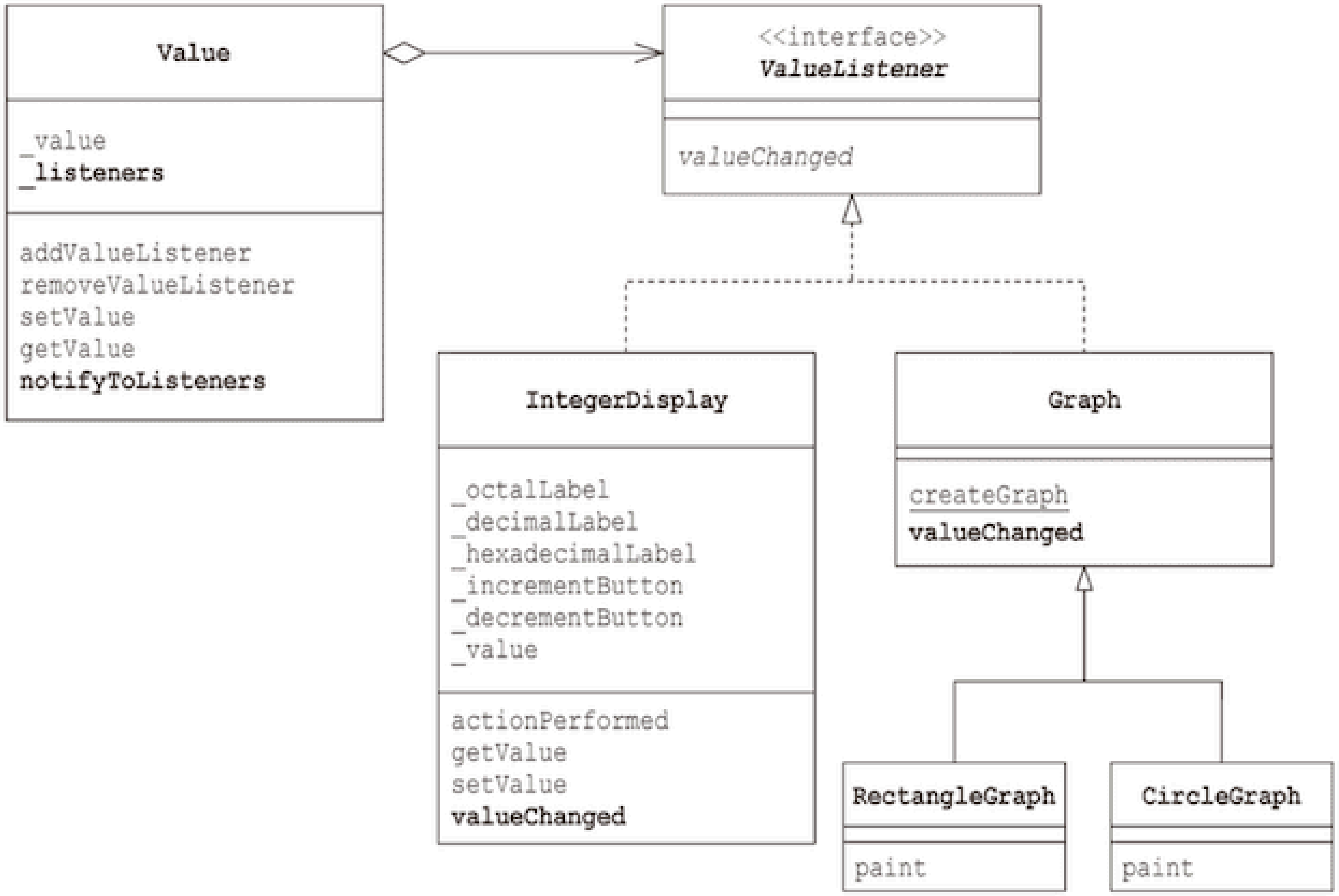
```python
class Graph(QWidget, ValueListener):
    """Base graph widget that visualizes values"""

    RECTANGLE = 0
    CIRCLE = 1

    def __init__(self):
        super().__init__()
        self.graph_value = 0
```

- We add graph objects as the listeners.

```
# Create graphs
self.graph_circle =
  Graph.create_graph(Graph.CIRCLE, 100, 100)
self.graph_rectangle =
  Graph.create_graph(Graph.RECTANGLE, 100, 50)

# Add listeners to value
self.value.add_value_listener(self)
self.value.add_value_listener(self.raph_circle)
self.value.add_value_listener(self.raph_rectangle)
```



Octal: 16
Decimal: 14
Hexadecimal: e



Discussion

Benefits of Duplicate Observed Data Refactoring