

A decorative graphic on the left side of the slide, consisting of a network of light green lines and small circles, resembling a circuit board or a neural network, extending from the top left towards the bottom left.

# CSE 219

# COMPUTER SCIENCE III

BEHAVIORAL DESIGN PATTERNS

SLIDES COURTESY:

RICHARD MCKENNA, STONY BROOK UNIVERSITY

# Behavioral Design Pattern

- These design patterns are specifically concerned with communication between objects.
- Increase flexibility in carrying out communication between objects.
- <https://www.youtube.com/watch?v=kiTDR0YoIqA>

# Common Design Patterns

## Creational

- Factory
- Singleton
- Builder
- Prototype

## Structural

- Decorator
- Adapter
- Facade
- Flyweight
- Component Architecture

## Behavioral

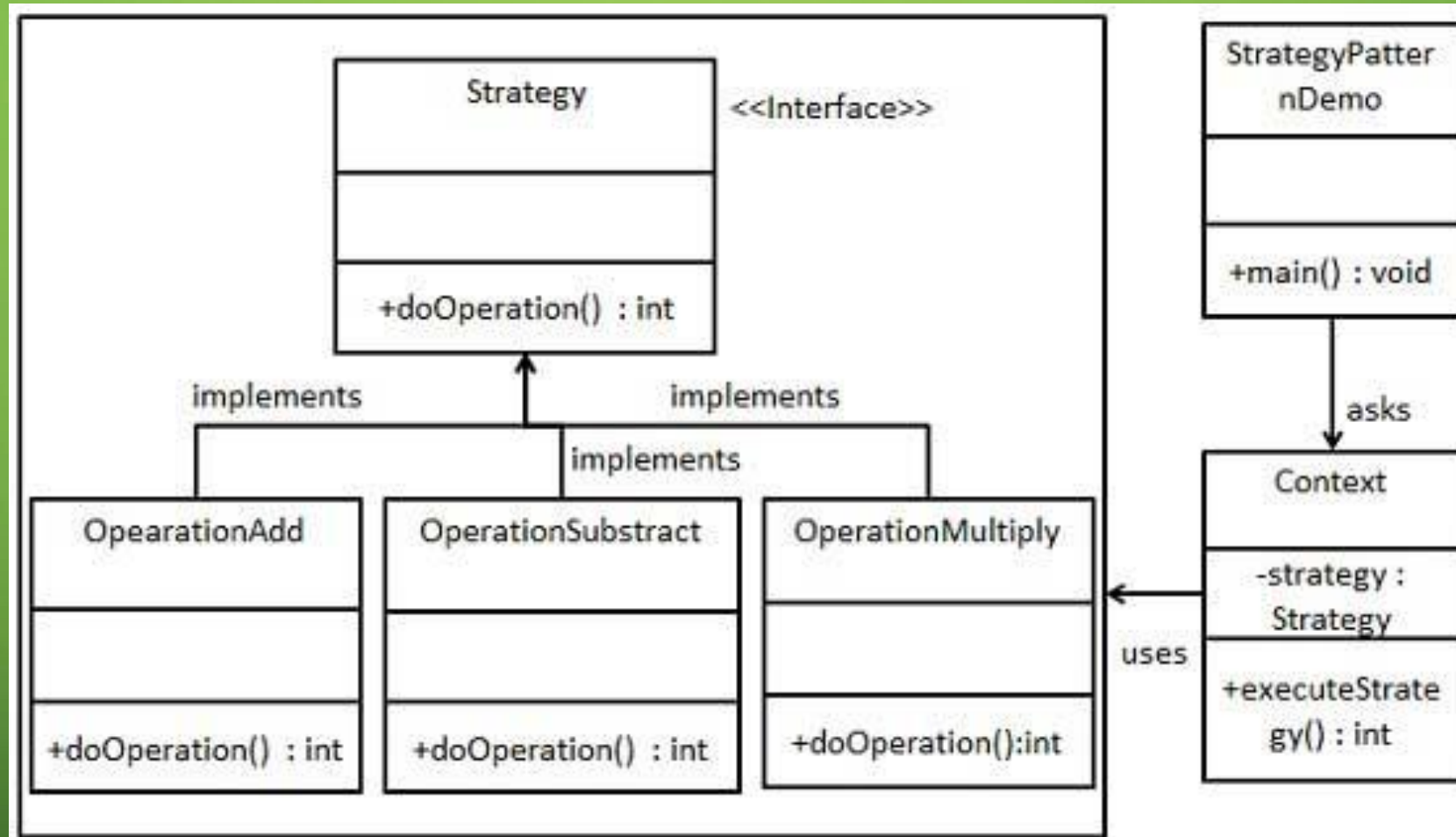
- **Strategy**
- Template
- Observer
- Command
- Iterator
- State

**Textbook: Head First Design Patterns**

# The Strategy Pattern

- How does it work?
  - defines a family of algorithms, encapsulates each one, and makes them interchangeable
  - lets the algorithm vary independently from the clients that use them
- An algorithm in a box
  - place essential steps for an algorithm in a strategy interface
  - different methods represent different parts of the algorithm
  - classes implementing this interface customize methods
- Classes can be composed (HAS-A) of the interface type
  - the interface type is the apparent type
  - the actual type can be determined at run-time

# Example



<https://www.youtube.com/watch?v=QKeL46JoDU4>

# Common Design Patterns

## Creational

- Factory
- Singleton
- Builder
- Prototype

## Structural

- Decorator
- Adapter
- Facade
- Flyweight
- Component Architecture

## Behavioral

- Strategy
- **Template**
- Observer
- Command
- Iterator
- State

**Textbook: Head First Design Patterns**

# Template Method Pattern

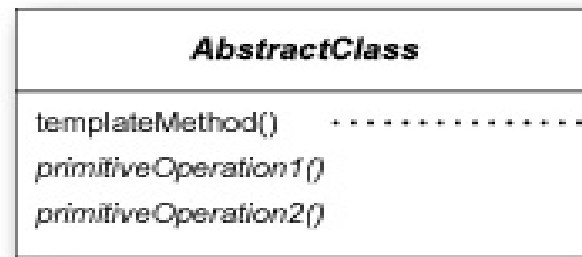
- Defines the skeleton of an algorithm in a method, deferring some steps to subclasses.
- Lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
- A template for an *algorithm*
- <https://www.youtube.com/watch?v=bPVDEk11z0o>

# Template Method Pattern

The template method makes use of the primitiveOperations to implement an algorithm. It is decoupled from the actual implementation of these operations.

The AbstractClass contains the template method.

...and abstract versions of the operations used in the template method.



There may be many ConcreteClasses, each implementing the full set of operations required by the template method.

The ConcreteClass implements the abstract operations, which are called when the `templateMethod()` needs them.



# Template Method Pattern

We've changed the `templateMethod()` to include a new method call.

```
abstract class AbstractClass {  
  
    final void templateMethod() {  
        primitiveOperation1();  
        primitiveOperation2();  
        concreteOperation();  
        hook();  
    }  
  
    abstract void primitiveOperation1();  
  
    abstract void primitiveOperation2();  
  
    final void concreteOperation() {  
        // implementation here  
    }  
  
    void hook() {}  
}
```

We still have our primitive methods; these are abstract and implemented by concrete subclasses.

A concrete operation is defined in the abstract class. This one is declared final so that subclasses can't override it. It may be used in the template method directly, or used by subclasses.

A concrete method, but it does nothing!

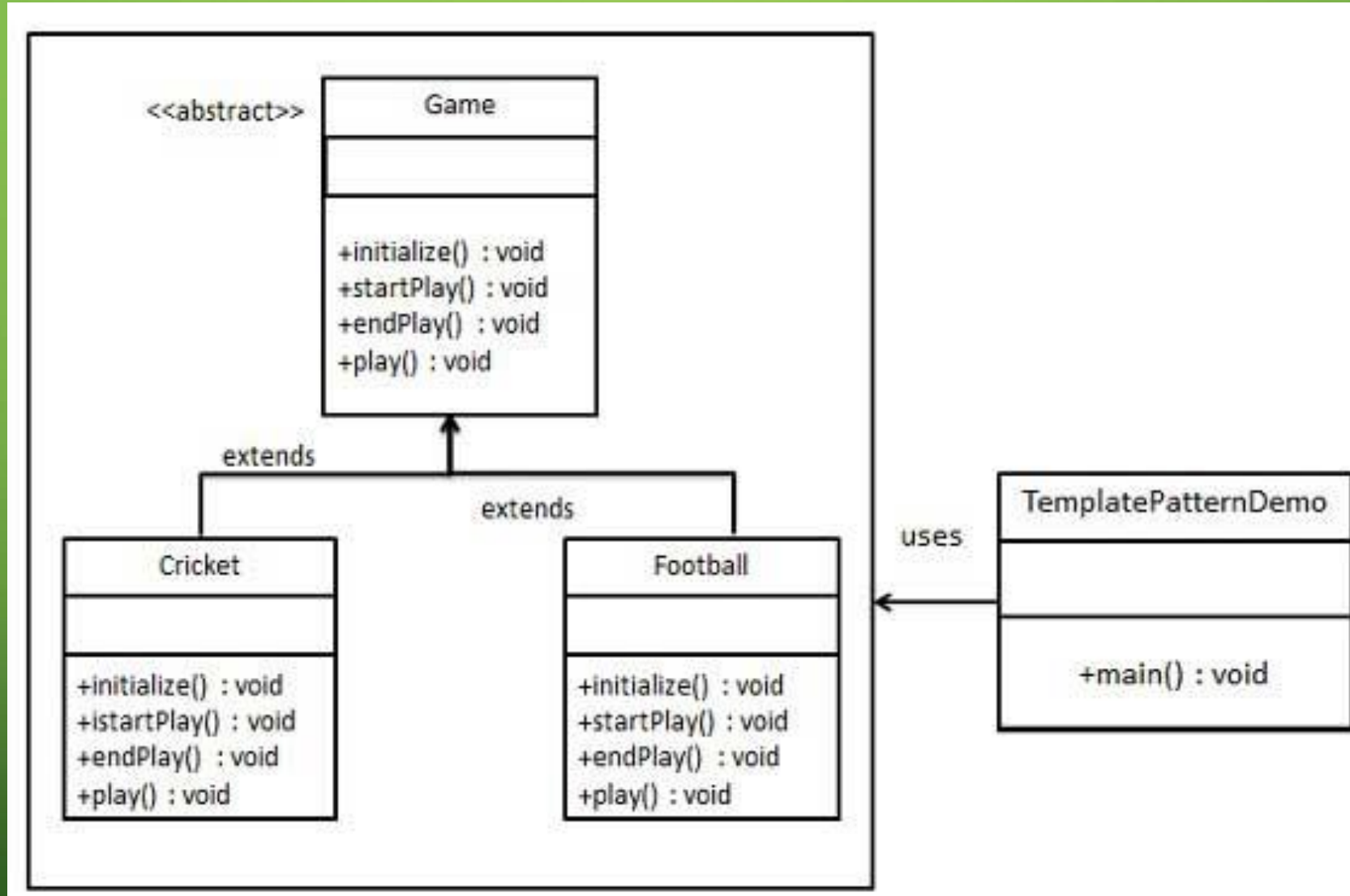
We can also have concrete methods that do nothing by default; we call these "hooks." Subclasses are free to override these but don't have to. We're going to see how these are useful on the next page.

# What's a hook?

- A type of method
- Declared in the abstract class
  - only given an empty or default implementation
- Gives subclasses the ability to “hook into” the algorithm at various points, if they wish
  - a subclass is also free to ignore the hook.



# Example



# Strategy vs. Template Method

- What's the difference?
- **Strategy**
  - subclasses decide how to implement steps in an algorithm
- **Template Method**
  - encapsulate interchangeable behaviors and use delegation to decide which behavior to use

# Common Design Patterns

## Creational

- Factory
- Singleton
- Builder
- Prototype

## Structural

- Decorator
- Adapter
- Facade
- Flyweight
- Component Architecture

## Behavioral

- Strategy
- Template
- **Observer**
- Command
- Iterator
- State

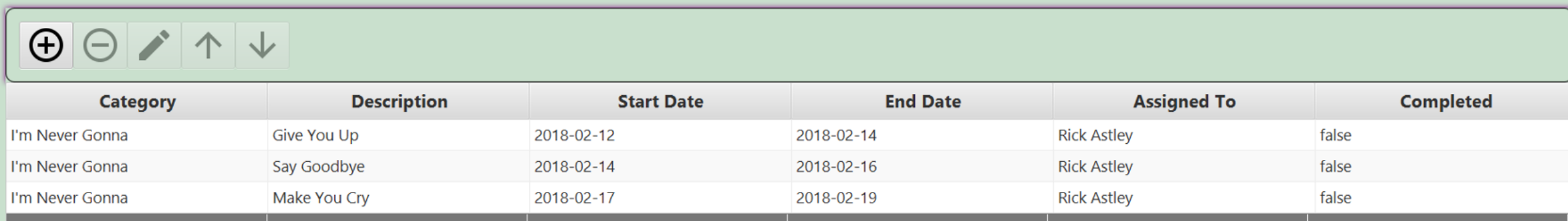
**Textbook: Head First Design Patterns**

# The Observer Pattern

- Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and updated automatically.
- Hmm, where have we seen this?
  - in our GUI
- State Manager class maintains application's state
  - call methods to change app's state
  - app's state change forces update of GUI
- <https://www.youtube.com/watch?v=hb2iv-UjfJs>

# TableView

- Used to display spreadsheets and tables



Category	Description	Start Date	End Date	Assigned To	Completed
I'm Never Gonna	Give You Up	2018-02-12	2018-02-14	Rick Astley	false
I'm Never Gonna	Say Goodbye	2018-02-14	2018-02-16	Rick Astley	false
I'm Never Gonna	Make You Cry	2018-02-17	2018-02-19	Rick Astley	false

- How is the table data stored?
  - in an **ObservableList**
  - we call this the table's data *model*



# Editing the table

- To edit the table, you must go through the model:

```
TableView table = new TableView(...  
ObservableList<DataPrototype> model = table.getItems();
```

```
// Add Data  
model.add(...
```

```
// Remove Data  
model.remove(...
```

```
// Change Data  
DataPrototype data = model.get(...  
data.set(...
```

```
// UPDATING THE MODEL (ObservableList) AND/OR THE DATA (DataPrototype)  
// WILL AUTOMATICALLY UPDATE THE VIEW (TableView) THANKS TO MVC!
```

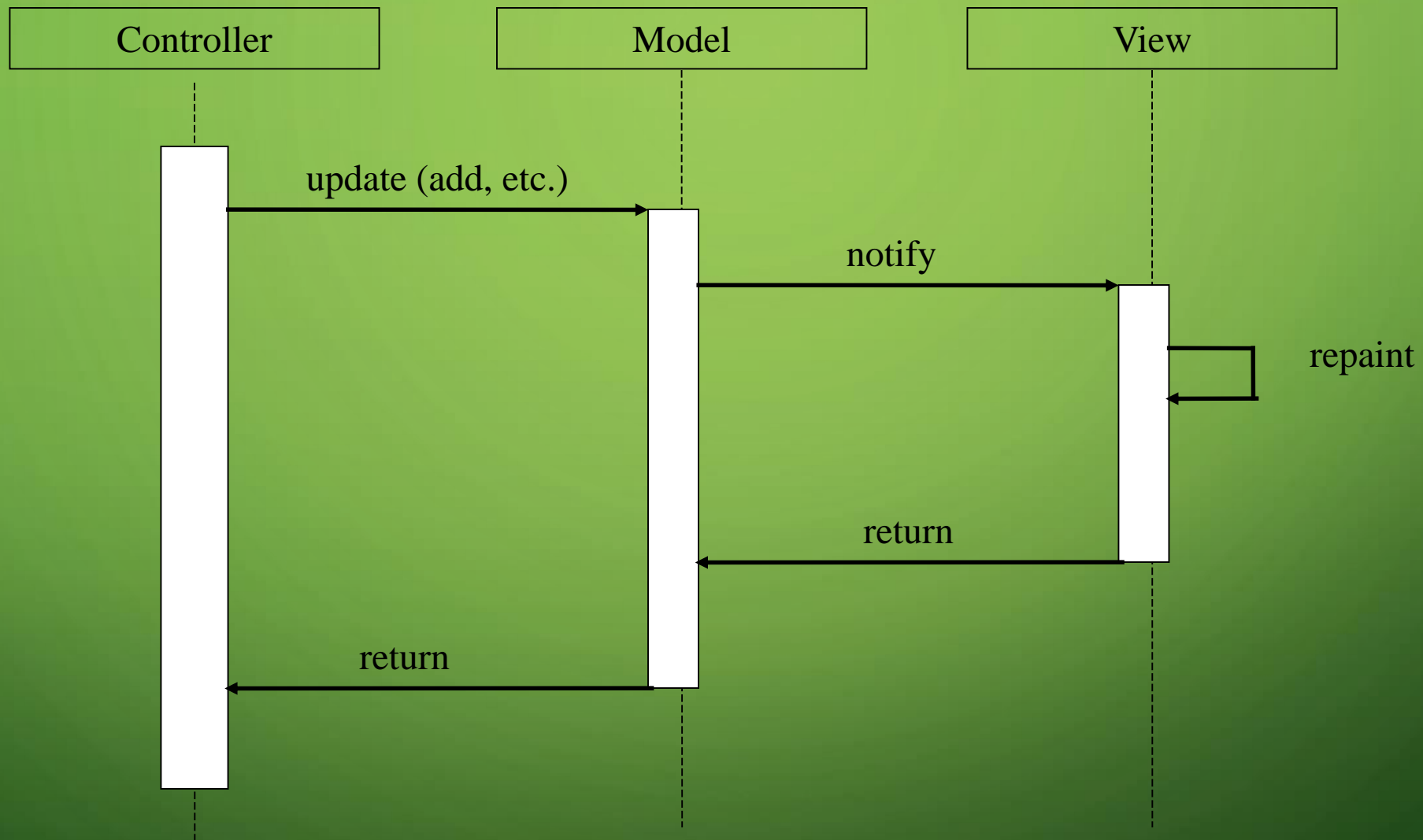


# Complex Controls have their own States

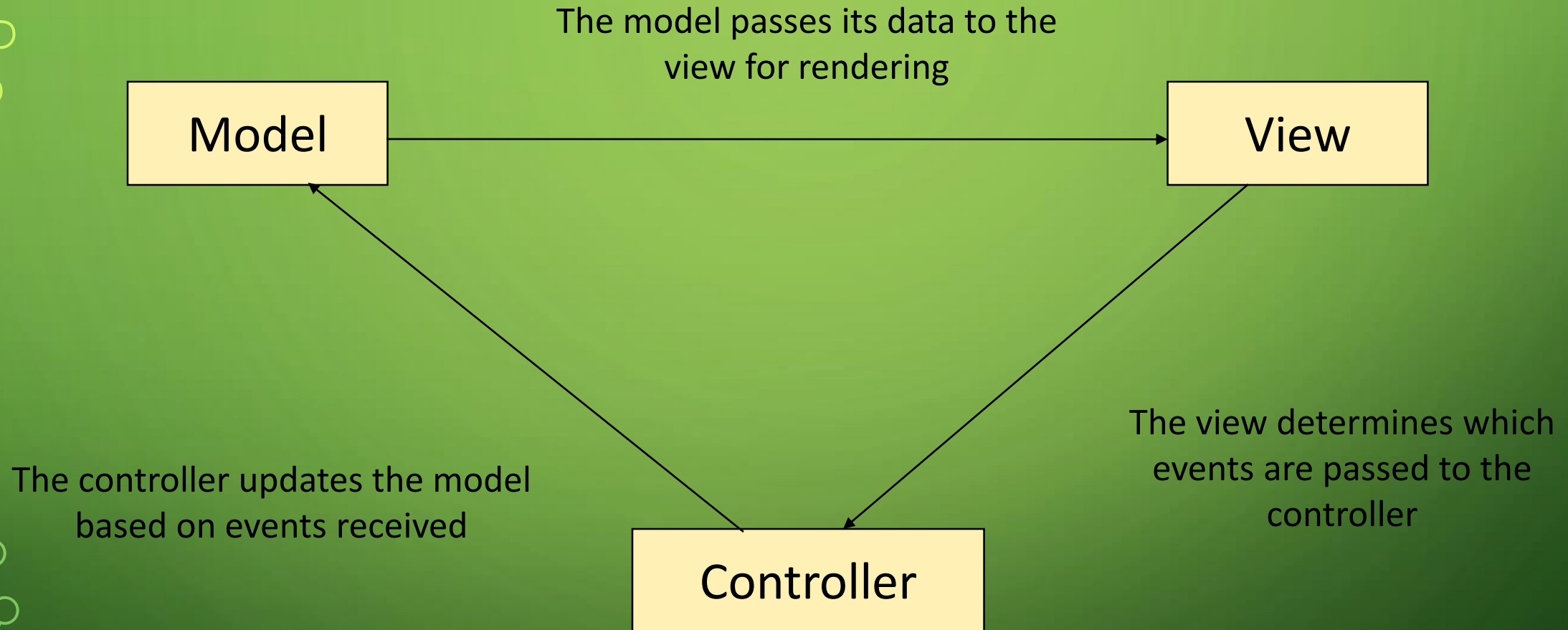
- Tables, trees, lists, combo boxes, etc.
  - data is managed separately from the view
  - when state changes, view is updated
- This is called MVC
  - Model
  - View
  - Controller
- MVC *employs* the Observer Pattern

# MVC *employs* the Observer Pattern

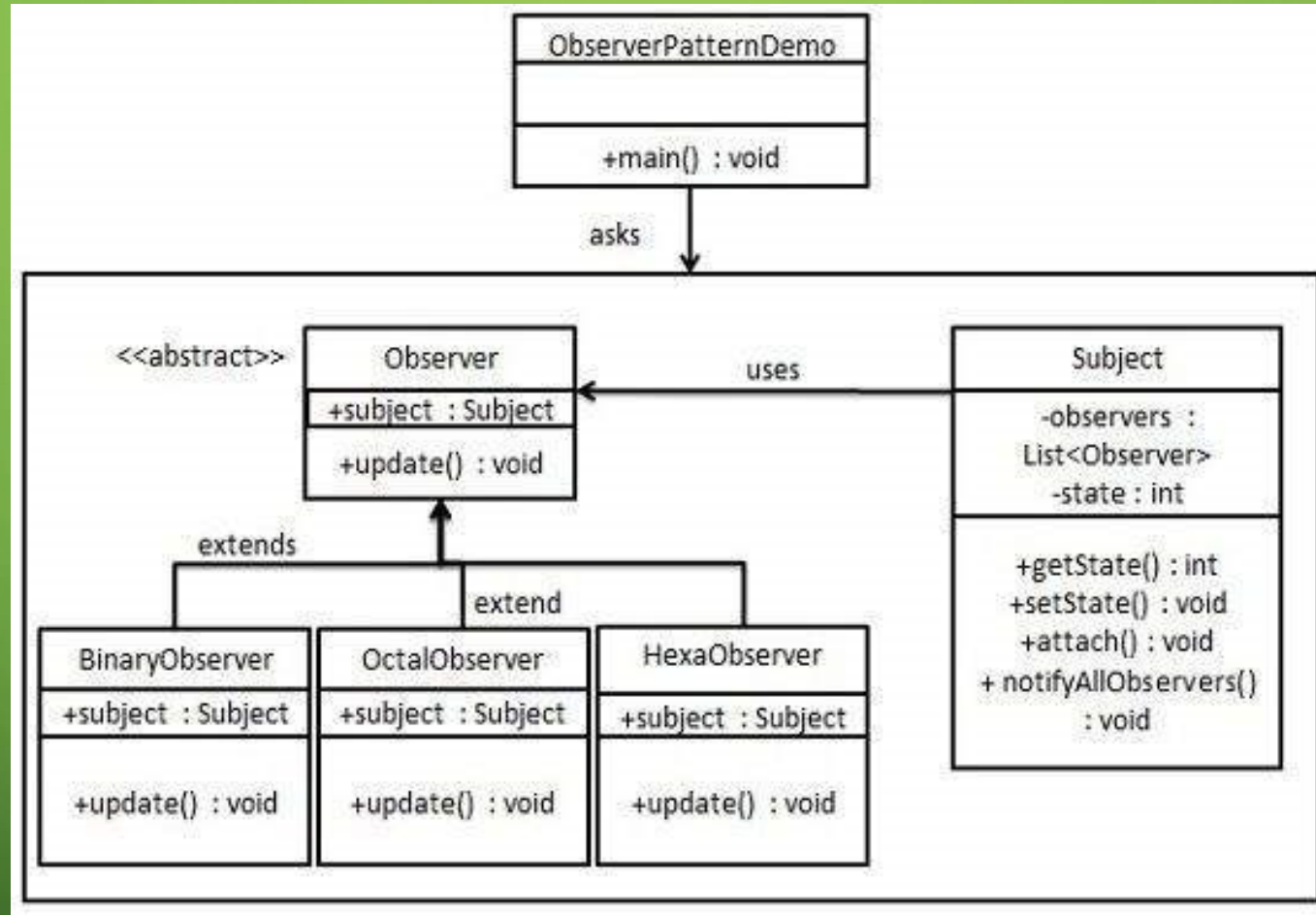
- **Model**
  - data structure, no visual representation
  - notifies views when something interesting happens
- **View**
  - visual representation
  - views attach themselves to model in order to be notified
- **Controller**
  - event handler
  - listeners that are attached to view in order to be notified of user interaction (or otherwise)
- **MVC Interaction**
  - controller updates model
  - model tells view that data has changed
  - view redrawn



# MVC Architecture



# Example



[https://www.tutorialspoint.com/design\\_pattern/observer\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/observer_pattern.htm)

# Common Design Patterns

## Creational

- Factory
- Singleton
- Builder
- Prototype

## Structural

- Decorator
- Adapter
- Facade
- Flyweight
- Component Architecture

## Behavioral

- Strategy
- Template
- Observer
- **Command**
- Iterator
- State

**Textbook: Head First Design Patterns**

# Command Abstraction

- For many GUIs, a single function may be triggered by many means (e.g., keystroke, menu, button, etc...)
  - we want to link all similar events to the same listener
- The information concerning the command can be abstracted to a **separate command object**
- Common Approach:
  - specify a String for each command
    - have listener respond to each command differently
  - ensure commands are handled in a uniform way
  - commands can be specified inside a text file
  - The Command Pattern

<https://www.youtube.com/watch?v=iNKvqMiPtmY>



# Example

- Suppose I wanted to create a simple GUI:
  - 1 colored panel
  - 2 buttons, yellow & red
  - 2 menu items, yellow & red
  - clicking on the buttons or menu items changes the color of the panel
- Since the buttons & the menu items both perform the same function, they should be tied to the same commands
  - I could even add popup menu items





# Using Command Strings

```
public class ColorCommandFrame1 extends JFrame
    implements ActionListener {


    private Toolkit tk = Toolkit.getDefaultToolkit();
    private ImageIcon yellowIcon
        = new ImageIcon(tk.getImage("yellow_bullet.bmp"));
    private ImageIcon redIcon
        = new ImageIcon(tk.getImage("red_bullet.bmp"));

    private JPanel coloredPanel = new JPanel();
    private JButton yellowButton = new JButton(yellowIcon);
    private JButton redButton = new JButton(redIcon);

    private JMenuBar menuBar = new JMenuBar();
    private JMenu colorMenu = new JMenu("Color");
    private JMenuItem yellowMenuItem = new JMenuItem(yellowIcon);
    private JMenuItem redMenuItem = new JMenuItem(redIcon);


    private JPopupMenu popupMenu = new JPopupMenu();
    private JMenuItem yellowPopupItem = new JMenuItem(yellowIcon);
    private JMenuItem redPopupItem = new JMenuItem(redIcon);

    private static final String YELLOW_COMMAND = "YELLOW_COMMAND";
    private static final String RED_COMMAND = "RED_COMMAND";
```



```
public ColorCommandFrame1() {  
    super("ColorCommandFrame1");  
    setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);  
    setExtendedState(JFrame.MAXIMIZED_BOTH);  
    initButtons();  
    initPopupMenu();  
    initMenu();  
}
```

```
public void initButtons() {  
    yellowButton.setActionCommand(YELLOW_COMMAND);  
    redButton.setActionCommand(RED_COMMAND);  
    yellowButton.addActionListener(this);  
    redButton.addActionListener(this);  
    coloredPanel.add(yellowButton);  
    coloredPanel.add(redButton);  
    Container contentPane = getContentPane();  
    contentPane.add(coloredPanel);  
}
```



```
public void initPopupMenu() {
    yellowPopupItem.setActionCommand(YELLOW_COMMAND);
    redPopupItem.setActionCommand(RED_COMMAND);
    yellowPopupItem.addActionListener(this);
    redPopupItem.addActionListener(this);
    popupMenu.add(yellowPopupItem);
    popupMenu.add(redPopupItem);

    coloredPanel.addMouseListener(new MouseAdapter() {
        public void mousePressed(MouseEvent e) {
            maybeShowPopup(e);
        }

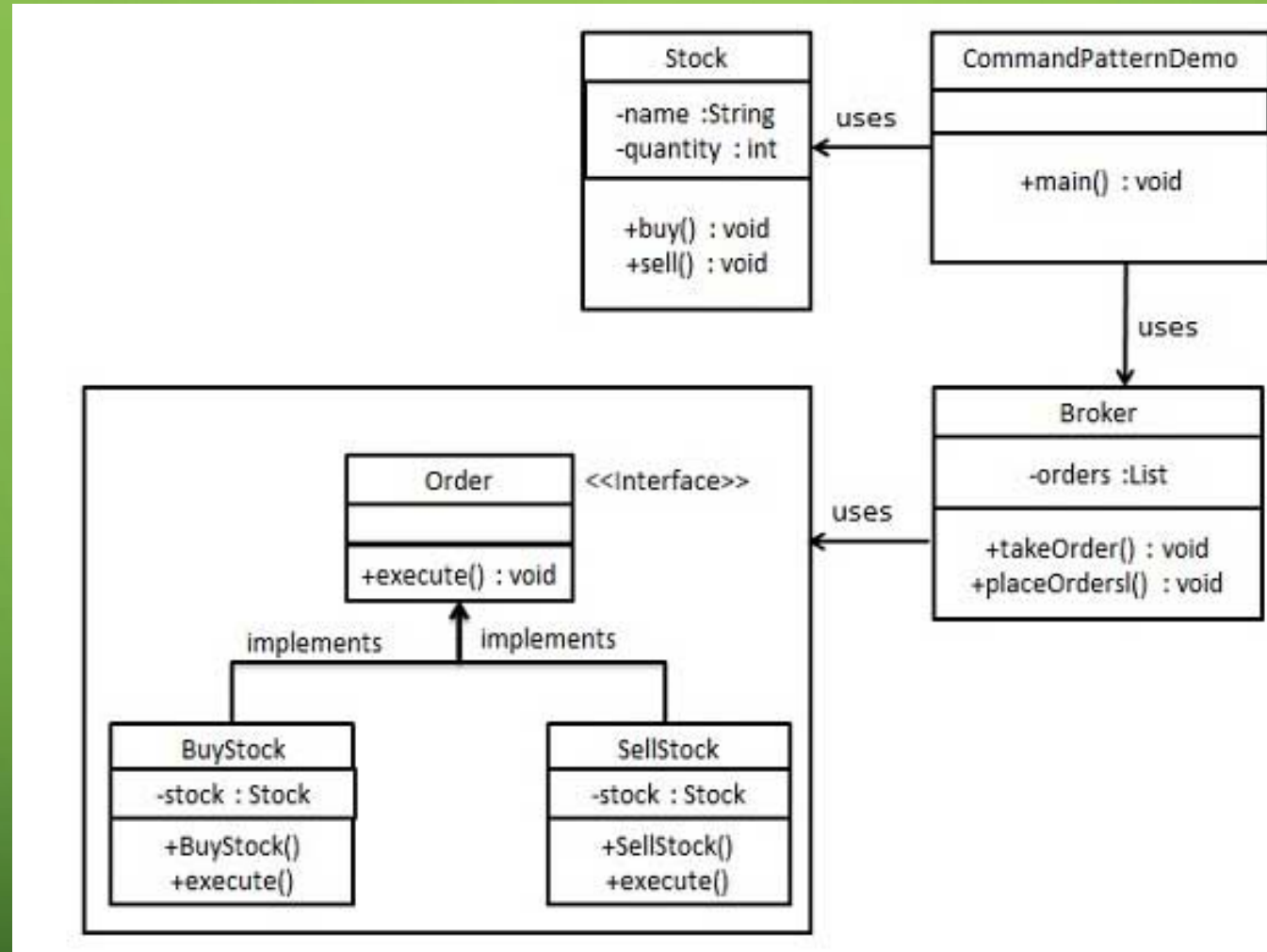
        public void mouseReleased(MouseEvent e) {
            maybeShowPopup(e);
        }

        private void maybeShowPopup(MouseEvent e) {
            if (e.isPopupTrigger()) {
                popupMenu.show(e.getComponent(), e.getX(), e.getY());
            }
        }
    });
}
```

```
public void initMenu() {  
    yellowMenuItem.setActionCommand(YELLOW_COMMAND) ;  
    redMenuItem.setActionCommand(RED_COMMAND) ;  
    yellowMenuItem.addActionListener(this) ;  
    redMenuItem.addActionListener(this) ;  
    colorMenu.add(yellowMenuItem) ;  
    colorMenu.add(redMenuItem) ;  
    menuBar.add(colorMenu) ;  
    setJMenuBar(menuBar) ;  
}
```

```
public void actionPerformed(ActionEvent ae) {  
    String command = ae.getActionCommand() ;  
    if (command.equals(YELLOW_COMMAND))  
        coloredPanel.setBackground(Color.YELLOW) ;  
    else if (command.equals(RED_COMMAND))  
        coloredPanel.setBackground(Color.RED) ;  
}
```

# Example



[https://www.tutorialspoint.com/design\\_pattern/command\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/command_pattern.htm)

# Common Design Patterns

## Creational

- Factory
- Singleton
- Builder
- Prototype

## Structural

- Decorator
- Adapter
- Facade
- Flyweight
- Component Architecture

## Behavioral

- Strategy
- Template
- Observer
- Command
- **Iterator**
- State

**Textbook: Head First Design Patterns**



# Iteration

- What's the problem?
  - you have to perform some operation on a sequence of elements in a given data structure
- Solution:
  - Iterator Pattern a.k.a. Iteration Abstraction
    - iterate over a group of objects without revealing details of how the items are obtained

<https://www.youtube.com/watch?v=Pganyj1dVVU>

# Iterator

- An **Iterator** produces proper elements for processing
- Defining an Iterator may be complex
- Using an Iterator must be simple
  - they're all used in the same way
- E.g. **update()** all elements of **List list**:

```
Iterator it;  
for (it=list.listIterator(); it.hasNext(); )  
    it.next().update();
```

- Makes iteration through elements of a set “higher level”
- Separates the *production* of elements for iteration from the *operation* at each step in the iteration.



# Iterator (cont'd)

- Iterator is a design pattern that is encountered very often.
  - Problem: Mechanism to operate on every element of a set.
  - Context: The set is represented in some data structure (list, array, hashtable, etc.)
  - Solution: Provide a way to iterate through every element.
- Common Classes using Iterators in Java API
  - StringTokenizer
  - Vector, ArrayList, etc ...
  - Even I/O streams work like Iterators

# Iterator (in Java)

```
public interface Iterator {  
    // Returns true if there are more  
    // elements to iterate over; false  
    // otherwise  
    public boolean hasNext();  
  
    // If there are more elements to  
    // iterate over, returns the next one.  
    // Modifies the state "this" to record  
    // that it has returned the element.  
    // If no elements remain, throw  
    // NoSuchElementException.  
    public Object next()  
        throws NoSuchElementException;  
  
    public void remove();  
}
```

# Iterator vs. Enumeration

- Java provides another interface **Enumeration** for iterating over a collection.
- **Iterator** is
  - newer (since JDK 1.2)
  - has shorter method names
  - has a **remove()** method to remove elements from a collection during iteration
- **Iterator**                      **Enumeration**  
hasNext()    hasMoreElements()  
next()        nextElement()  
remove()     -
- **Iterator** is recommended for new implementations.

# Example Loop controlled by next ()

```
private Payroll payroll = new Payroll();
```

```
...
```

```
public void decreasePayroll() {
```

```
    Iterator it = payroll.getIterator();
```

```
    while (it.hasNext()) {
```

```
        Employee e = (Employee)it.next();
```

```
        double salary = e.getSalary();
```

```
        e.setSalary(salary*.9);
```

```
    }
```

```
for (Employee emp : payroll) {
```

```
}
```

```
}
```

# Implementing an Iterator

```
public class Payroll {  
    private Employee[] employees;  
    private int num_employees;  
    ...  
    // An iterator to loop through all Employees  
    public Iterator getIterator() {  
        return new EmplGen();  
    }  
    ...  
    private class EmplGen implements Iterator {  
        // see next slide  
        ...  
    }  
}
```

# Implementing an Iterator

```
private class EmplGen implements Iterator {
```

```
    private int n = 0;
```

← state of iteration  
captured by index n

```
    public boolean hasNext() {  
        return n < num_employees;  
    }
```

← returns true if there  
is an element left  
to iterate over

```
    public Object next() throws NoSuchElementException {
```

```
        Object obj;
```

```
        if (n < num_employees) {
```

```
            obj = employees[n];
```

```
            n++;
```

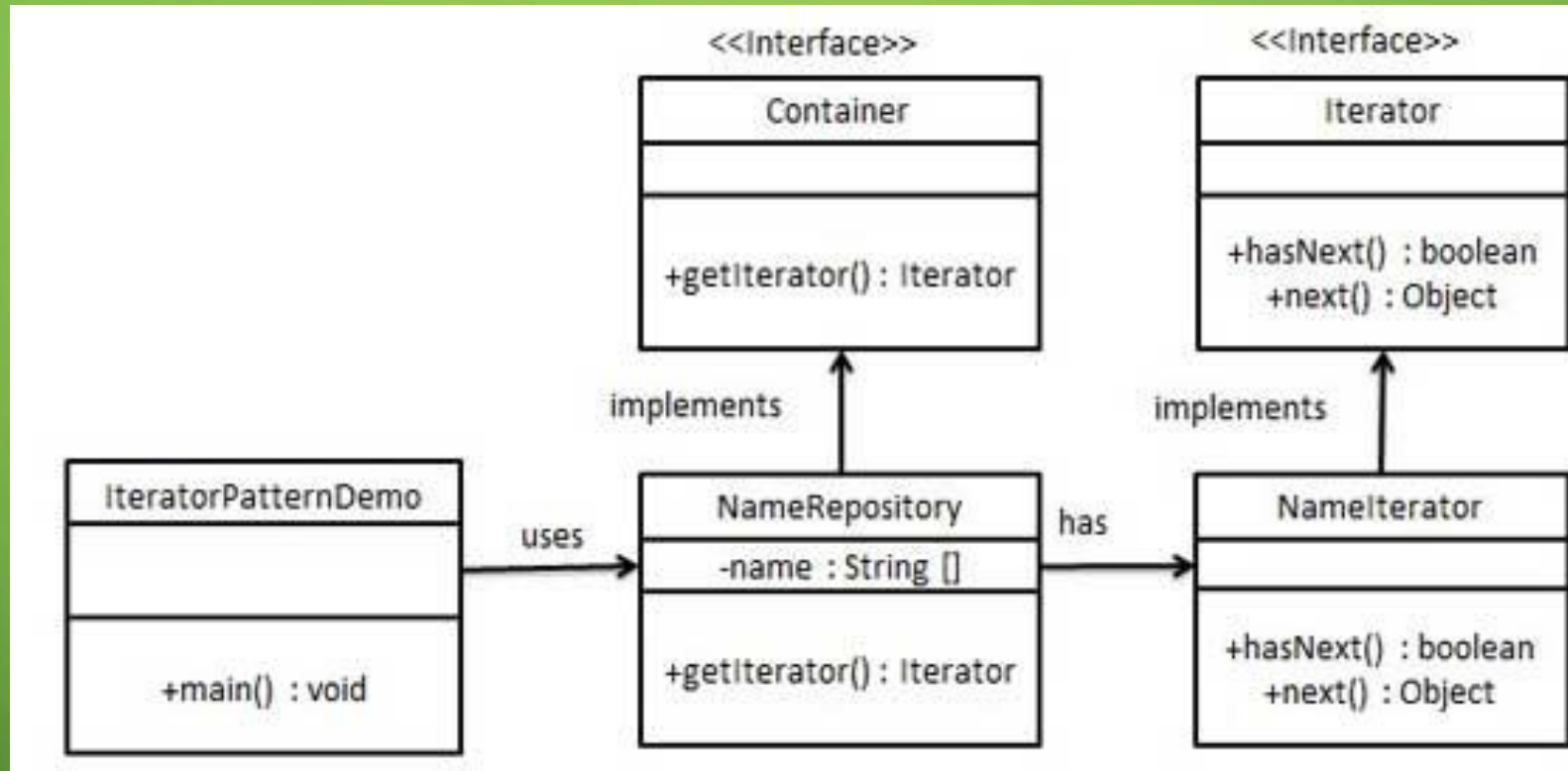
```
            return obj;
```

← returns the next  
element in the  
iteration sequence

```
        }
```

```
        else throw new NoSuchElementException  
            ("No More Employees");  
    }
```

# Example



[https://www.tutorialspoint.com/design\\_pattern/iterator\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/iterator_pattern.htm)



# Common Design Patterns

## Creational

- Factory
- Singleton
- Builder
- Prototype

## Structural

- Decorator
- Adapter
- Facade
- Flyweight
- Component Architecture

## Behavioral

- Strategy
- Template
- Observer
- Command
- Iterator
- State

**Textbook: Head First Design Patterns**



# State Pattern

- Dynamically change the representation of an object.
  - also called Data Abstraction
- Users of the object are unaware of the change.
- Example:
  - Implement a set as a **Vector** if the number of elements is small
  - Implement a set as a **Hashtable** if the number of elements is large
- State pattern is used only on mutable objects.

[https://www.youtube.com/watch?v=8jy\\_fXl1OzA](https://www.youtube.com/watch?v=8jy_fXl1OzA)

# Example: Set

```
public class Set {
    private Object elements;

    public boolean isIn(Object member){
        if (elements instanceof Vector)
            // search using Vector methods
        else
            // search using Hashtable methods
        }

    public void add(Object member){
        if (elements instanceof Vector)
            // add using Vector methods
        else
            // add using Hashtable methods
        }
    }
}
```

# Using the state pattern: SetRep

```
public interface SetRep {  
    public void add(object member);  
    public boolean isIn(Object member);  
    public int size();  
    public void remove();  
}
```

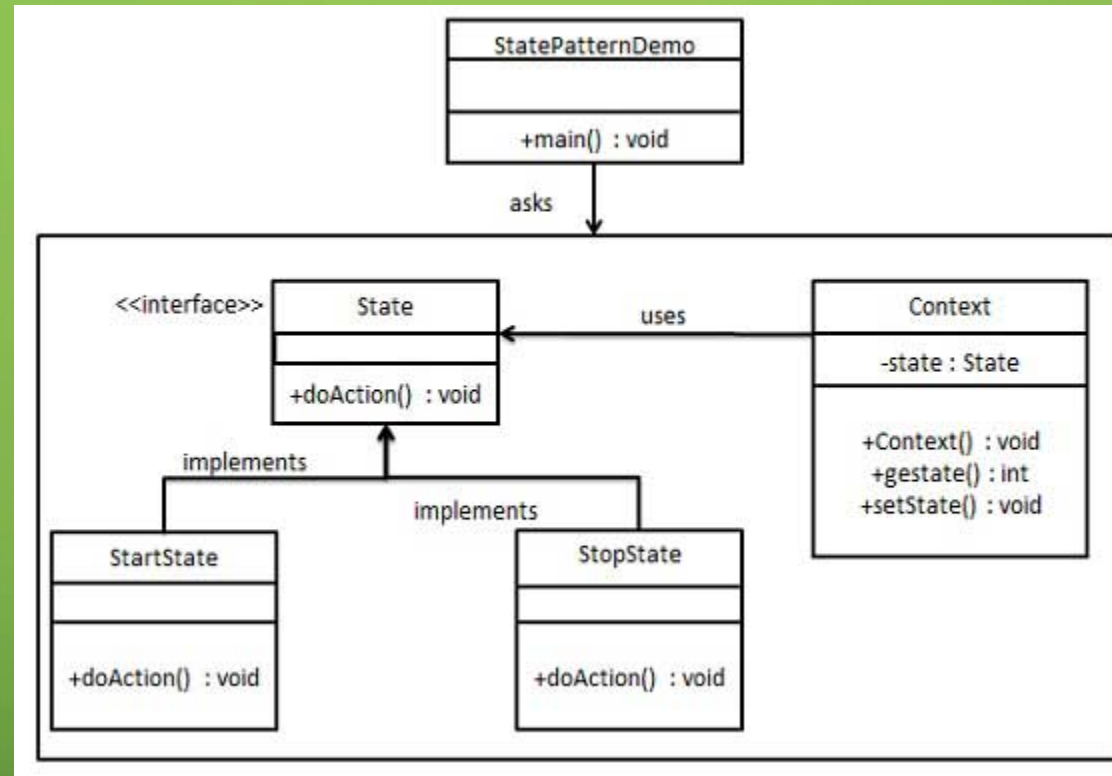
```
public class SmallSet implements SetRep {  
    private Vector set;  
    public void add(Object element) { ... }  
    public void remove() { ... }  
    ...  
}
```

```
public class LargeSet implements SetRep {  
    private Hashtable set;  
    public void add(Object element) { ... }  
    public void remove() { ... }  
    ...  
}
```

# Using the state pattern: a new Set

```
public class Set {  
    private SetRep rep;  
    private int threshold;  
  
    public void add(Object element) {  
        if (rep.size() == threshold)  
            rep = new LargeSet(rep.elements());  
        rep.add(element);  
    }  
  
    public void remove(Object element) {  
        rep.remove(elem);  
        if (rep.size == threshold)  
            rep = new SmallSet(rep.elements());  
    }  
}
```

# Example



[https://www.tutorialspoint.com/design\\_pattern/state\\_pattern.htm](https://www.tutorialspoint.com/design_pattern/state_pattern.htm)

# There are others too

- Chain of Responsibility
- Composite
- Interpreter
- Mediator
- Memento
- Proxy
- Visitor