Software Engineering I (02161)

Week 9

Assoc. Prof. Hubert Baumeister

DTU Compute
Technical University of Denmark

Spring 2018

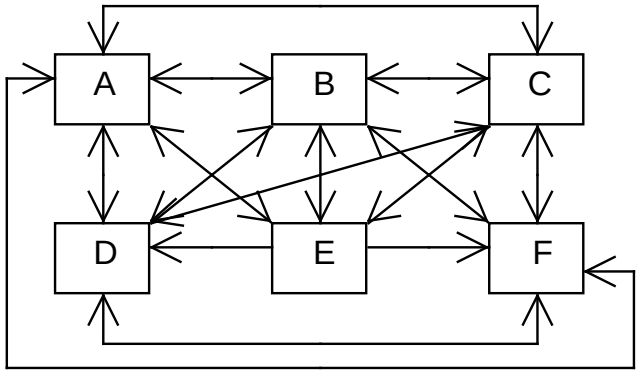
Recap

- Last week
  - Principles of good design: DRY, KISS, YAGNI, high cohesion / low coupling, Layered Architecture
  - Design Patterns: Composite Pattern, Template Method, Facade, Strategy/Policy, Wrapper/Adapter
    - → More design patterns: Investigate yourself
- This week
  - Layered Architecture: Presentation Layer
  - Model View Controller
  - Observer pattern
  - SOLID Principles

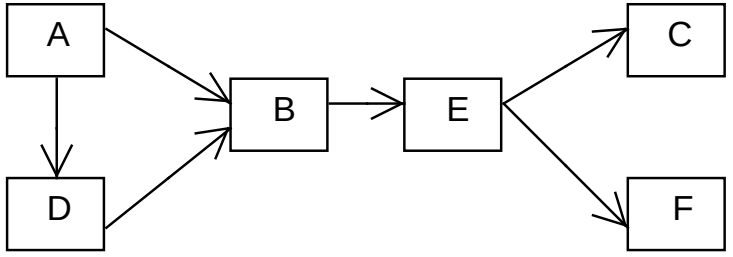
Contents

High Cohesion / Low Coupling

High coupling

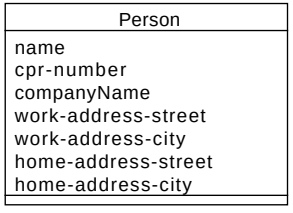


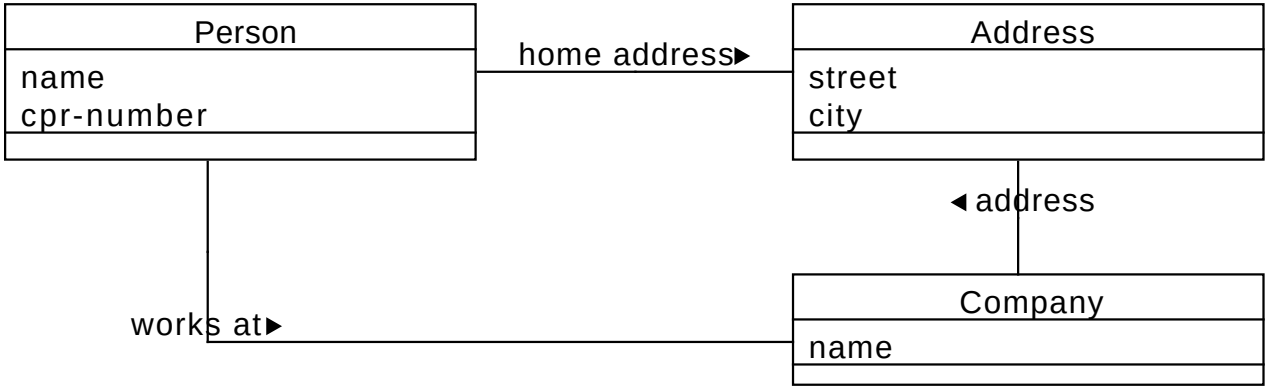
Low coupling



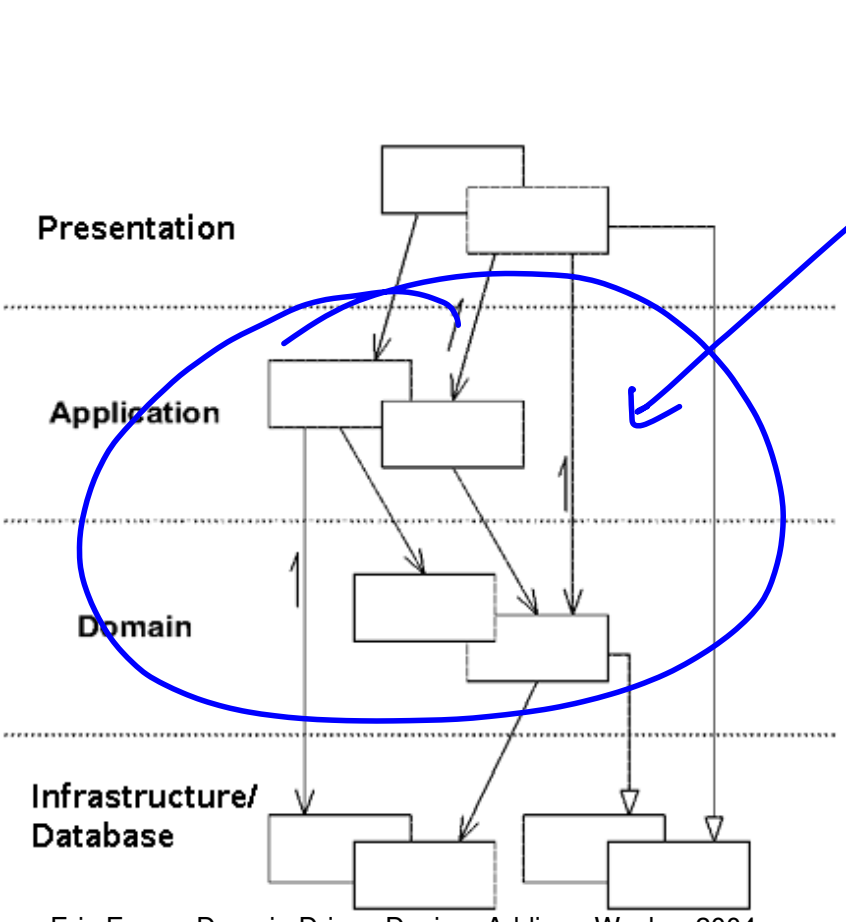
High Cohesion

Low Cohesion





Layered Architecture



Eric Evans, Domain Driven Design, Addison-Wesley, 2004

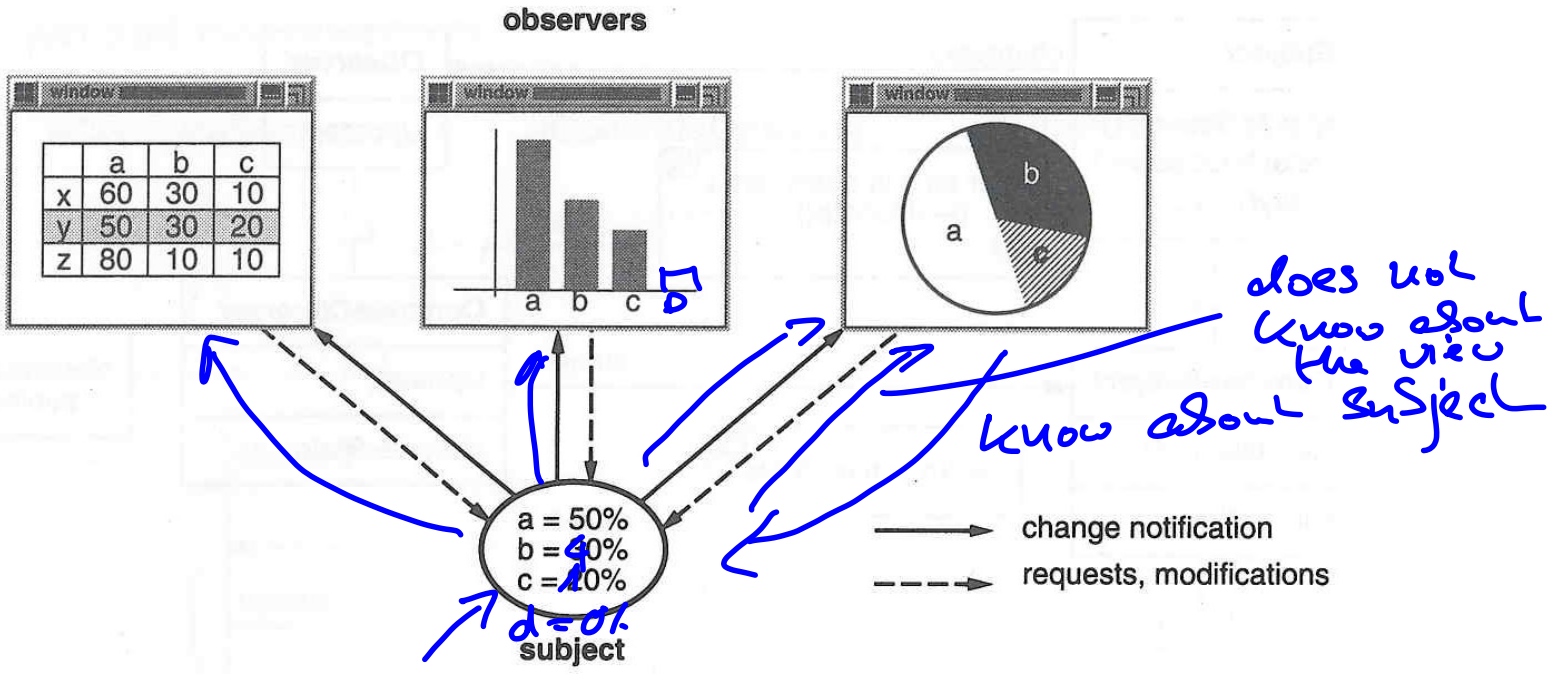
Important:

- All business logic in application and domain layer
- **No** business logic in presentation layer or infrastructure/database layer

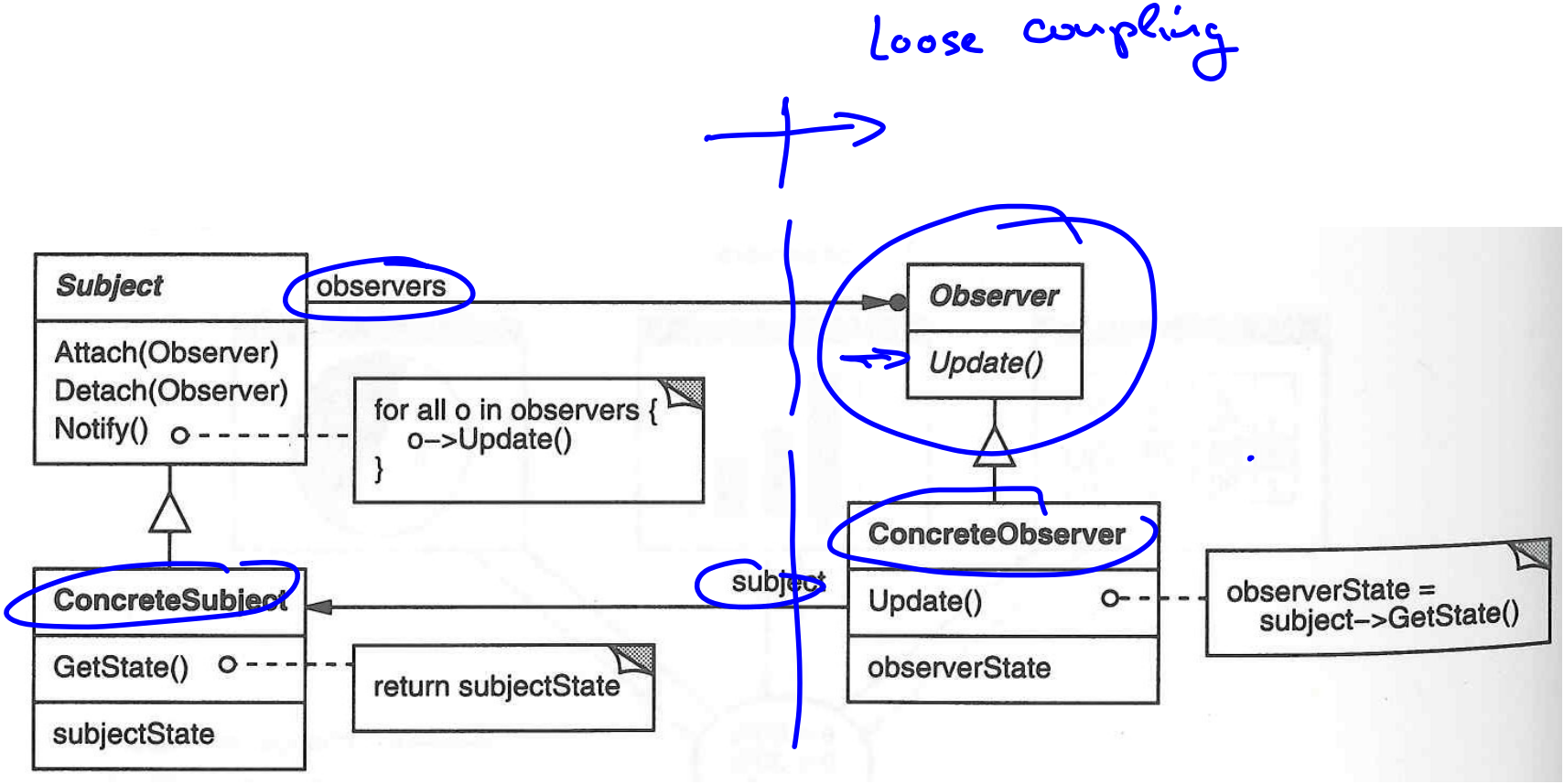
Observer Pattern

Observer Pattern

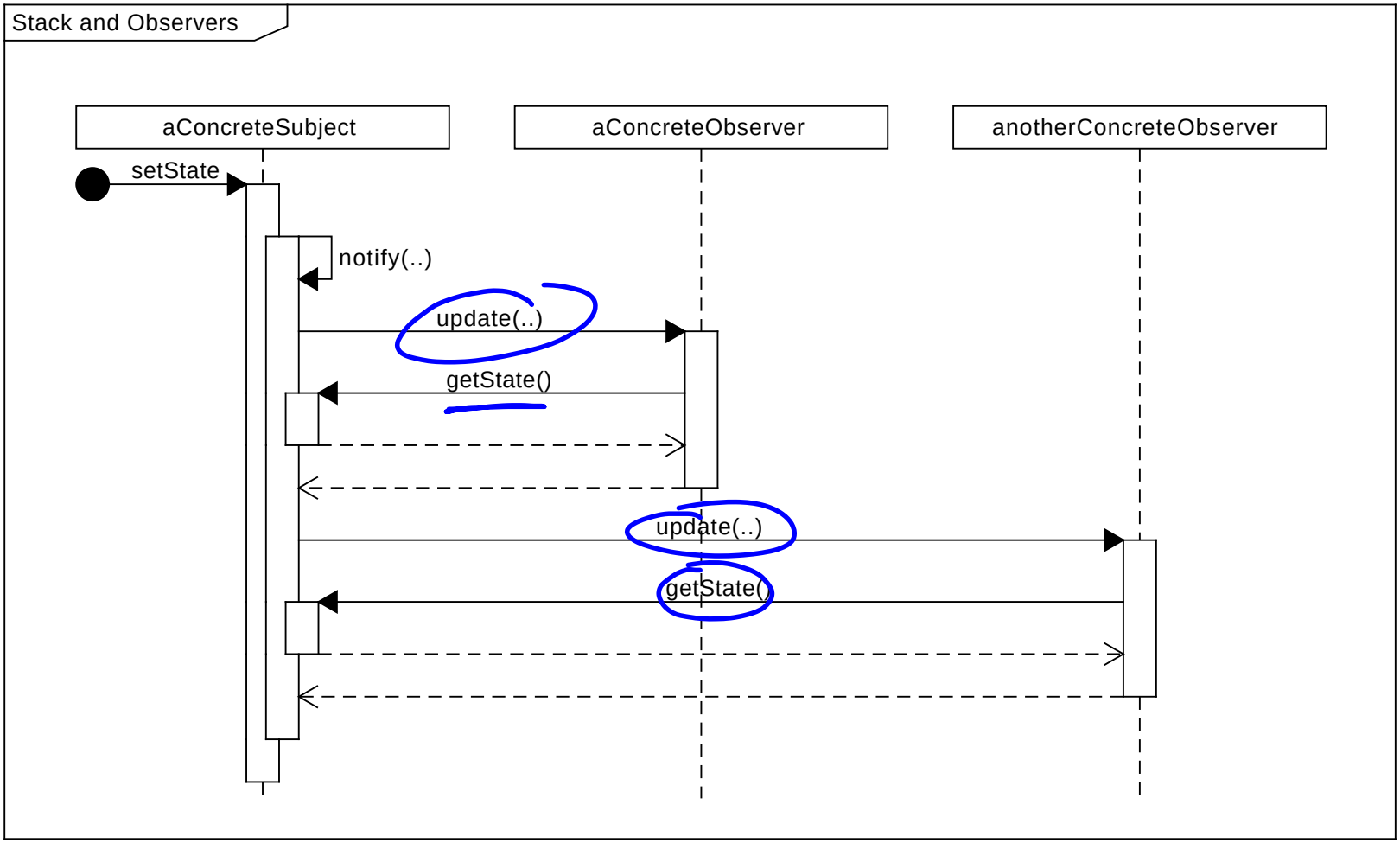
Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.



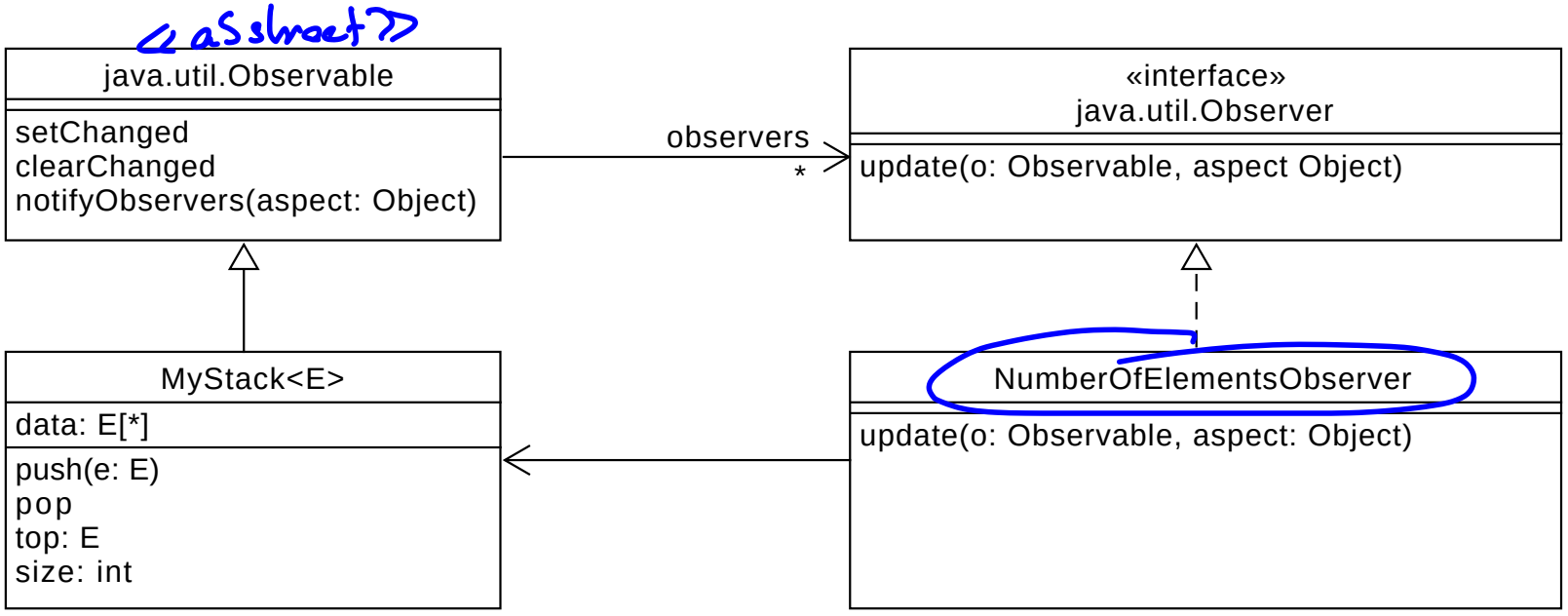
Observer Pattern



Observer Pattern



Implementation in Java



Example: Stack with observers

```
public class MyStack<E> extends Observable {
   List<E> data = new ArrayList<E>();

   void push(Type o) {
      data.add(o);
      setChanged();
      notifyObserver("data elements");
   }

   E pop() {
      E top = data.remove(data.size())'
      setChanged();
      notifyObserver("data elements");
   }

   E top() {
      return data.get(data.size());
   }

   int size() {
      return data.size();
   }
   String toString() {
      System.out.print("[");
      for (E d : data) { System.out.print(" "+d);
      System.out.print(" ]");
   }
   ...
```

Example: Stack observer

- Observe the number of elements that are on the stack.
- Each time the stack changes its size, a message is printed on the console.

```
class NumberOfElementsObserver() implements Observer {

   public void update(Observable o, Object aspect) {
      System.out.println(((MyStack)o).size()+
            " elements on the stack");
   }
}
```

- Observe the elements on the stack.
- Each time the stack changes, print the elements of the stack on the console.

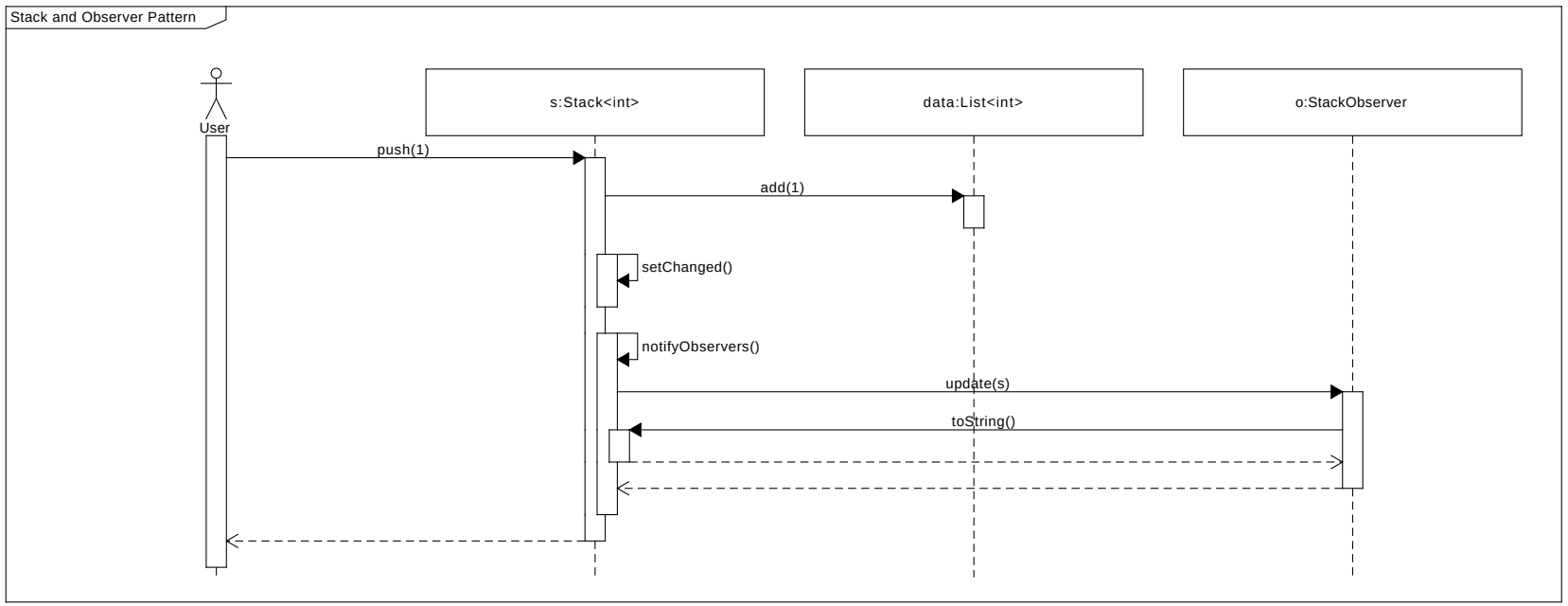
```
class StackObserver() implements Observer {

   public void update(Observable o, Object aspect) {
      System.out.println(o);
   }
}
```

Example: Stack observer

Adding an observer

```
....
MyStack<Integer> stack = new MyStack<Integer>;
NumberOfElementsObserver obs1 =
            new NumberOfElementsObserver();
NumberOfElementsObserver obs2 =
            new StackObserver();
stack.addObserver(obs1);
stack.push(10);
stack.addObserver(obs2);
stack.pop();
...
stack.deleteObserver(obs1)
...
```

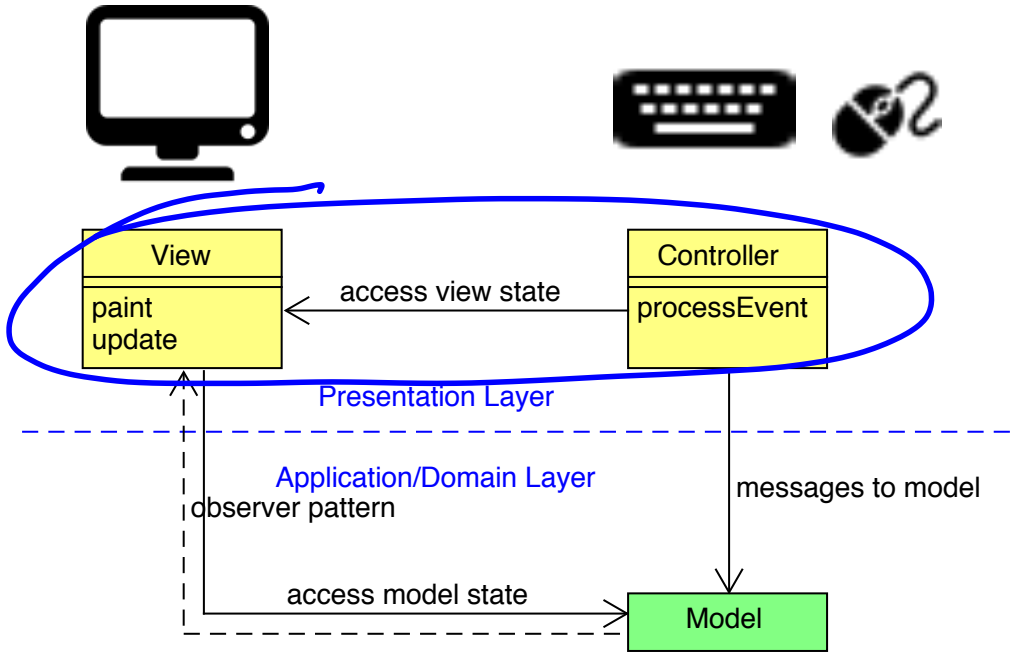
Sequence diagram for the stack



Contents

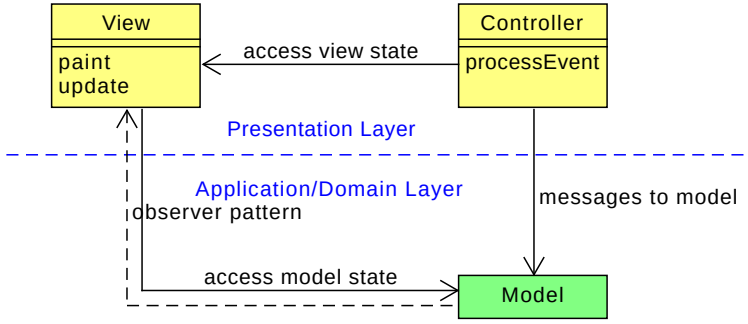
Model View Controller (MVC)

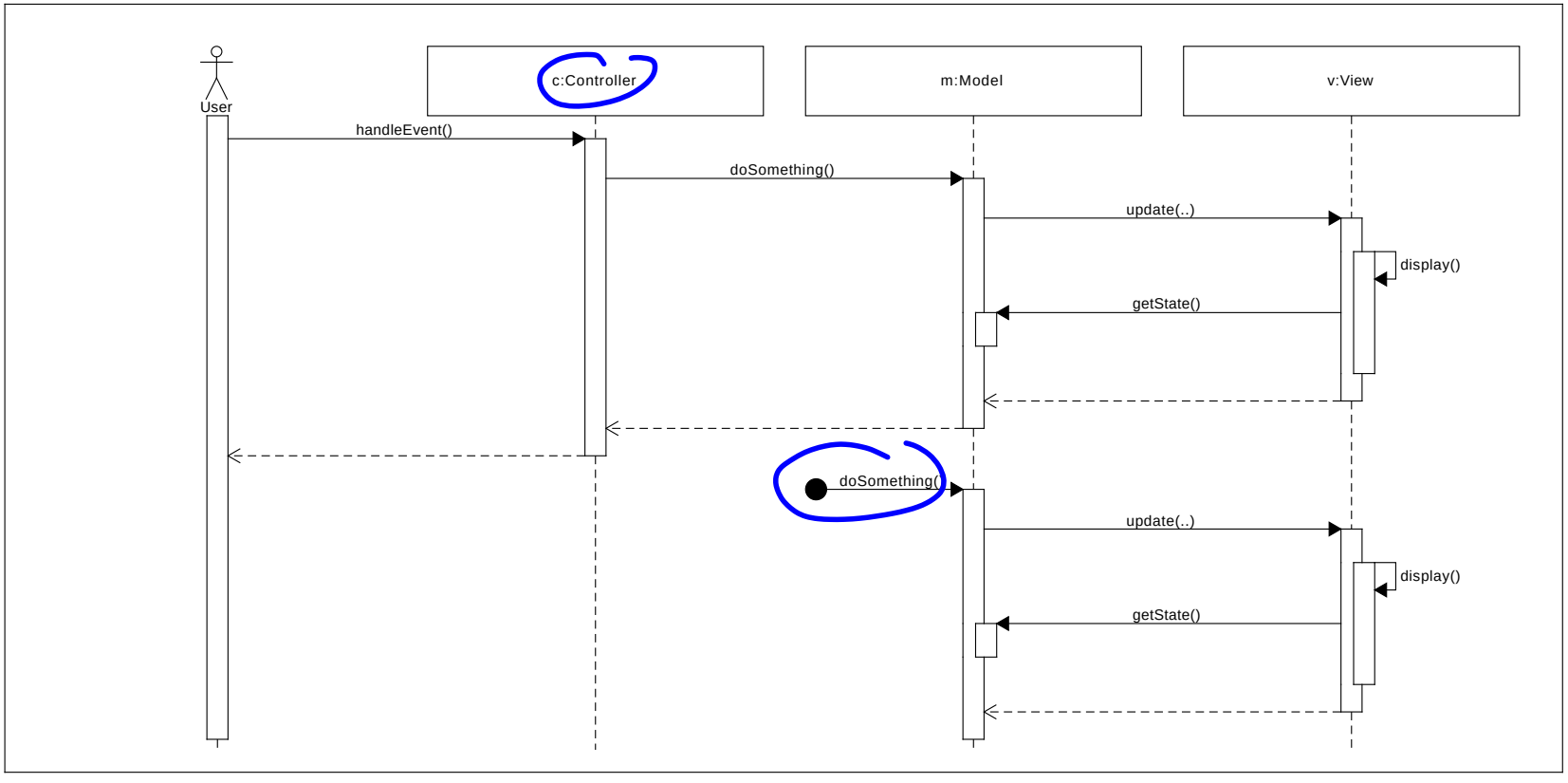
- Invented by Trygve Reenskaug for Smalltalk-76/80



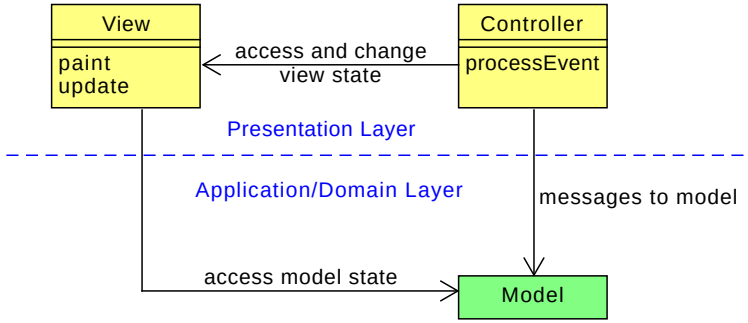
- View: translate model state to display graphics
- Controller: translates mouse and keyboard events to application/domain specific messages to model

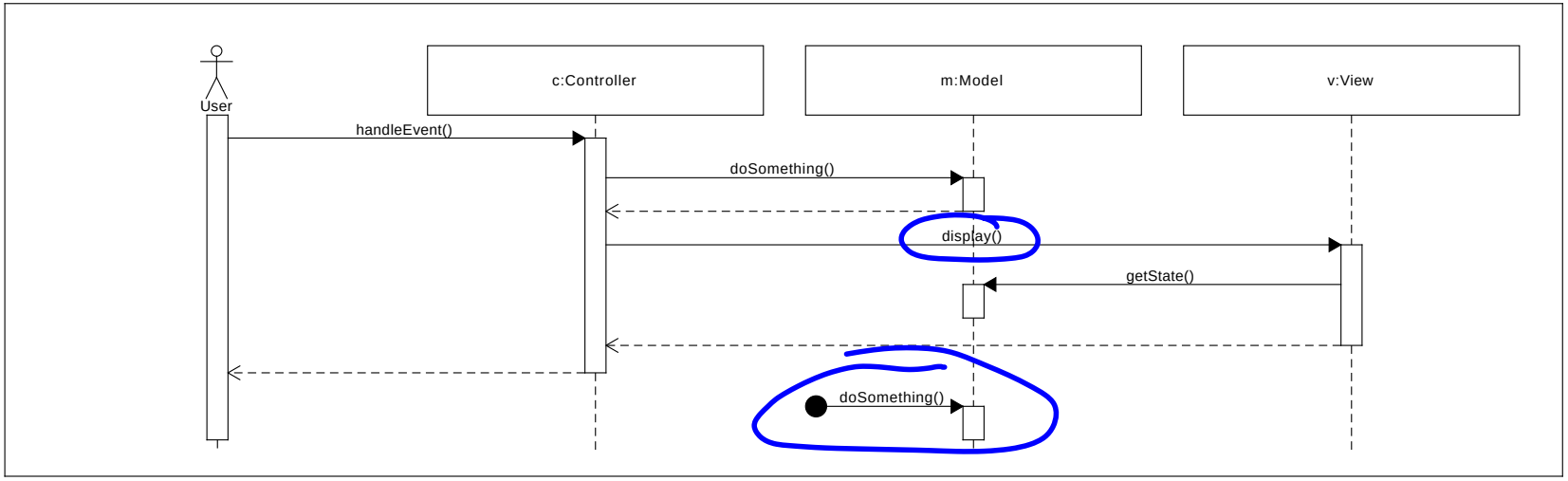
MVC with Observer Pattern





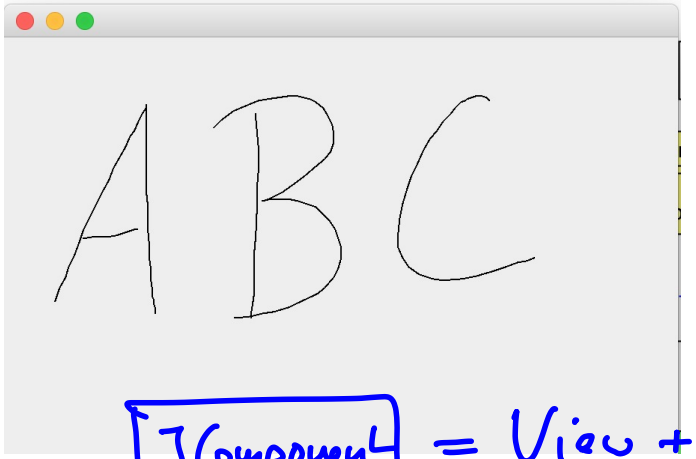
MVC without Observer Pattern



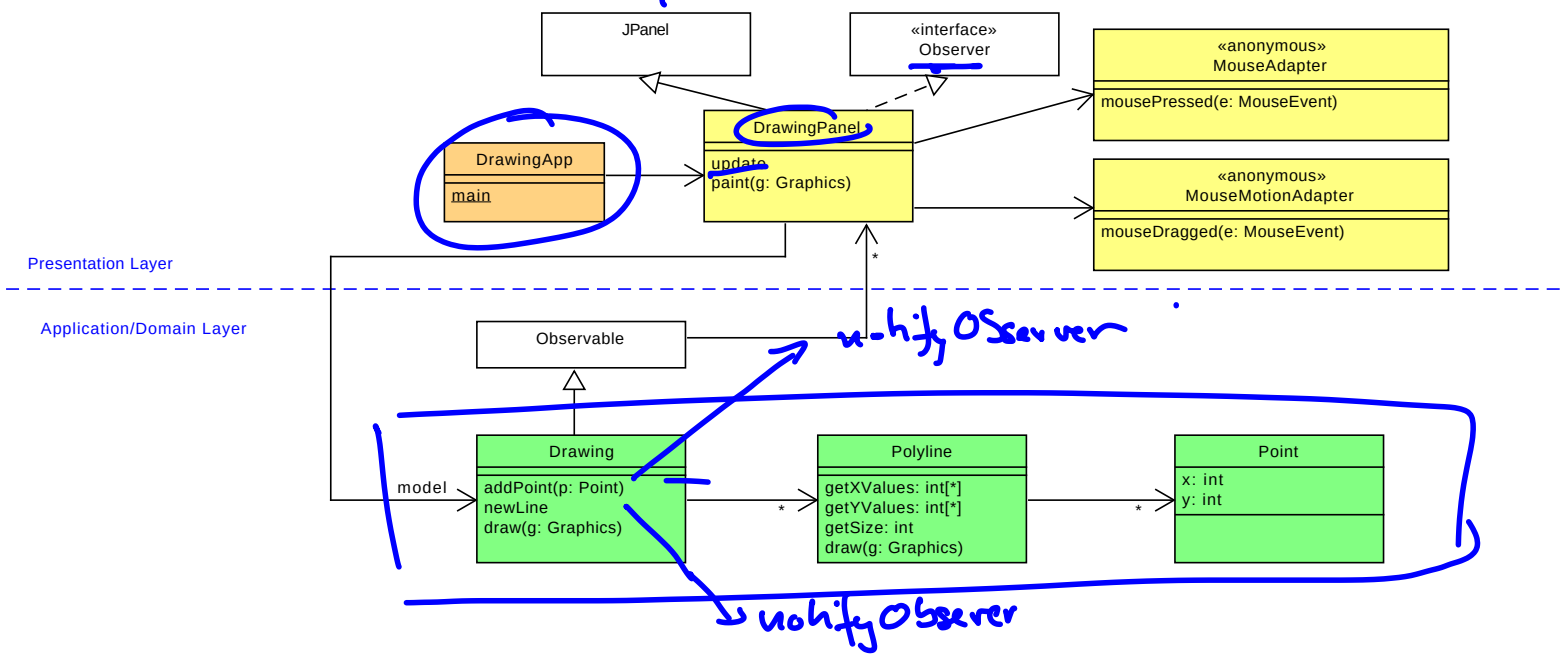


MVC Example: Drawing Program

Screenshot



Class diagram



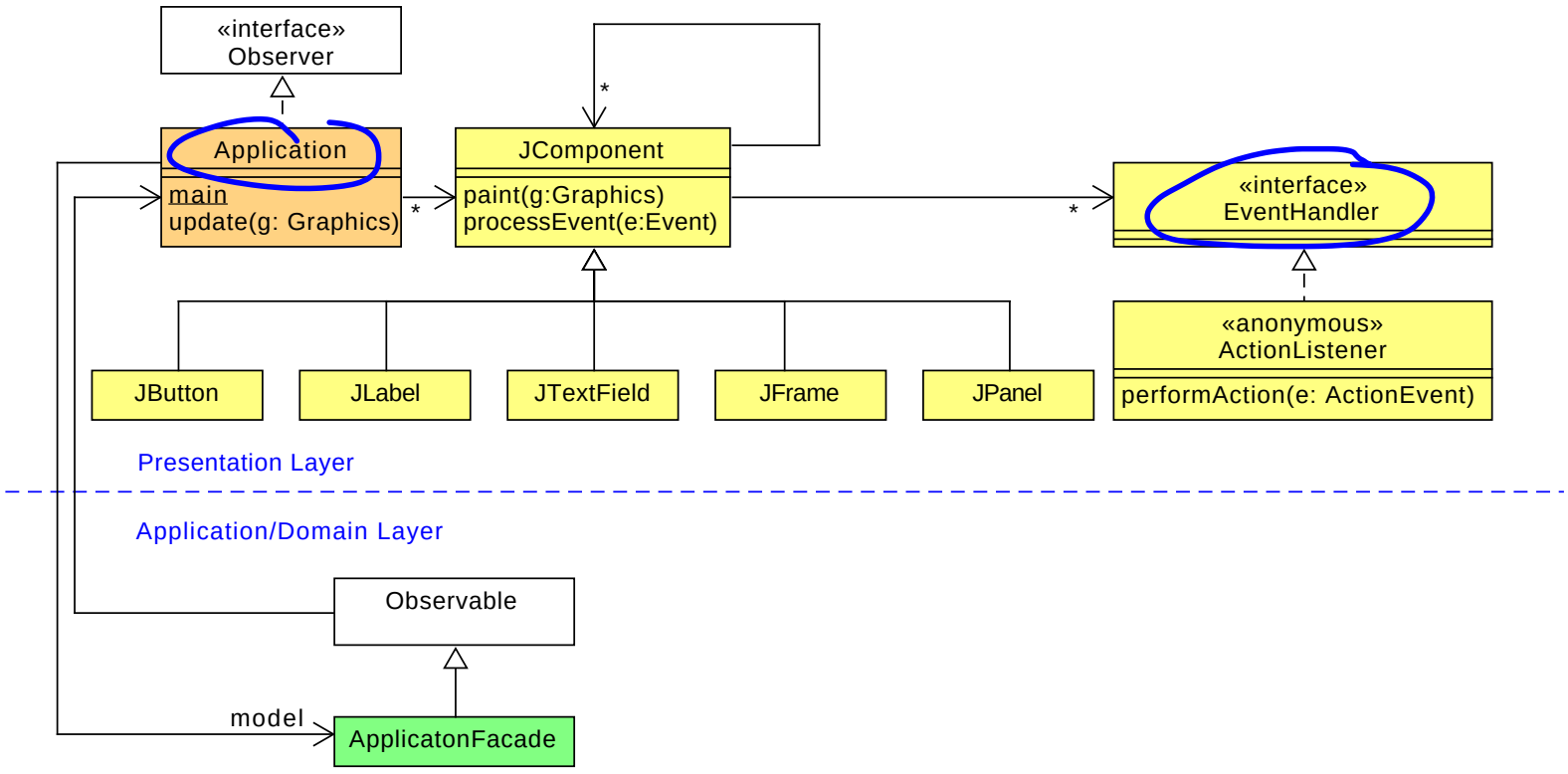
MVC Example: Drawing Program

```
public class DrawingPanel extends JPanel implements Observer {
  Drawing model;

  public DrawingPanel(Drawing model) {
    this.model = model;
    model.addObserver(this);

    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        drawing.newLine();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        drawing.addPoint(new Point(e.getX(),e.getY()));
      }
    });
  }

  public void paint(Graphics g) {
    drawing.draw(g);
  }

  public void update(Observable o, Object arg) {
    repaint();
  }
}
```

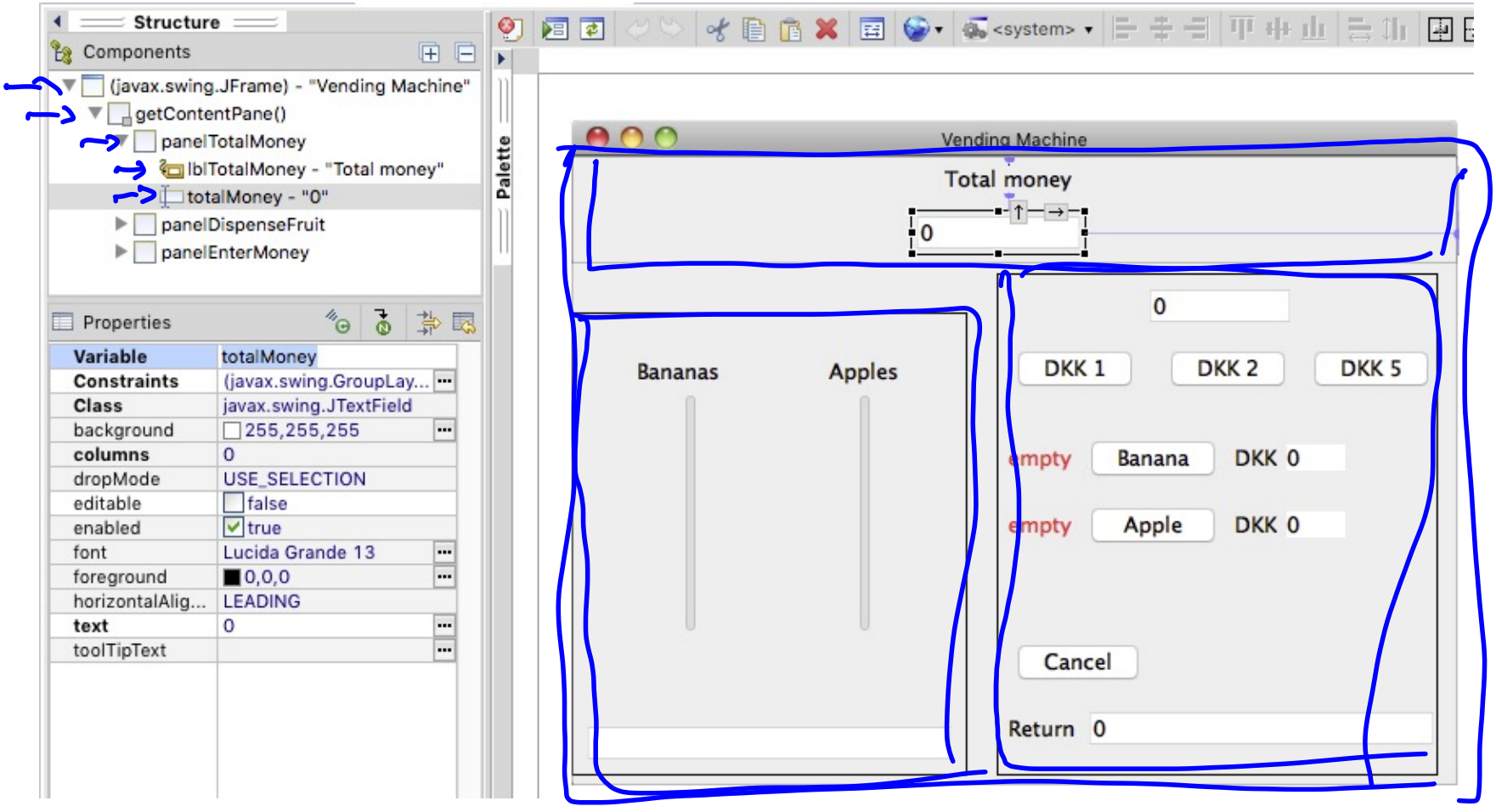
MVC Example: Drawing Program without Observer

```
public class DrawingPanel extends JPanel {
  Drawing model;

  public DrawingPanel(Drawing model) {
    this.model = model;
    model.addObserver(this);

    addMouseListener(new MouseAdapter() {
      public void mousePressed(MouseEvent e) {
        drawing.newLine();
        repaint();
      }
    });

    addMouseMotionListener(new MouseMotionAdapter() {
      public void mouseDragged(MouseEvent e) {
        drawing.addPoint(new Point(e.getX(),e.getY()));
        repaint();
      }
    });
  }

  public void paint(Graphics g) {
    drawing.draw(g);
  }
}
```

Java and Swing



Java and Swing



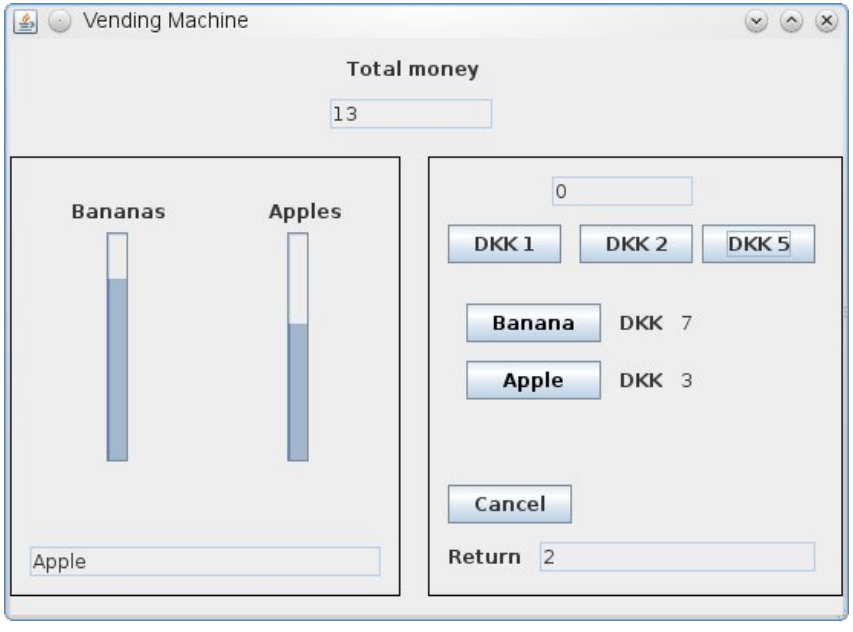
- Window Builder for Eclipse:
  `https://www.eclipse.org/windowbuilder/`

Contents

Example Vending Machine

Two different presentation layers; same application layer

- Swing GUI

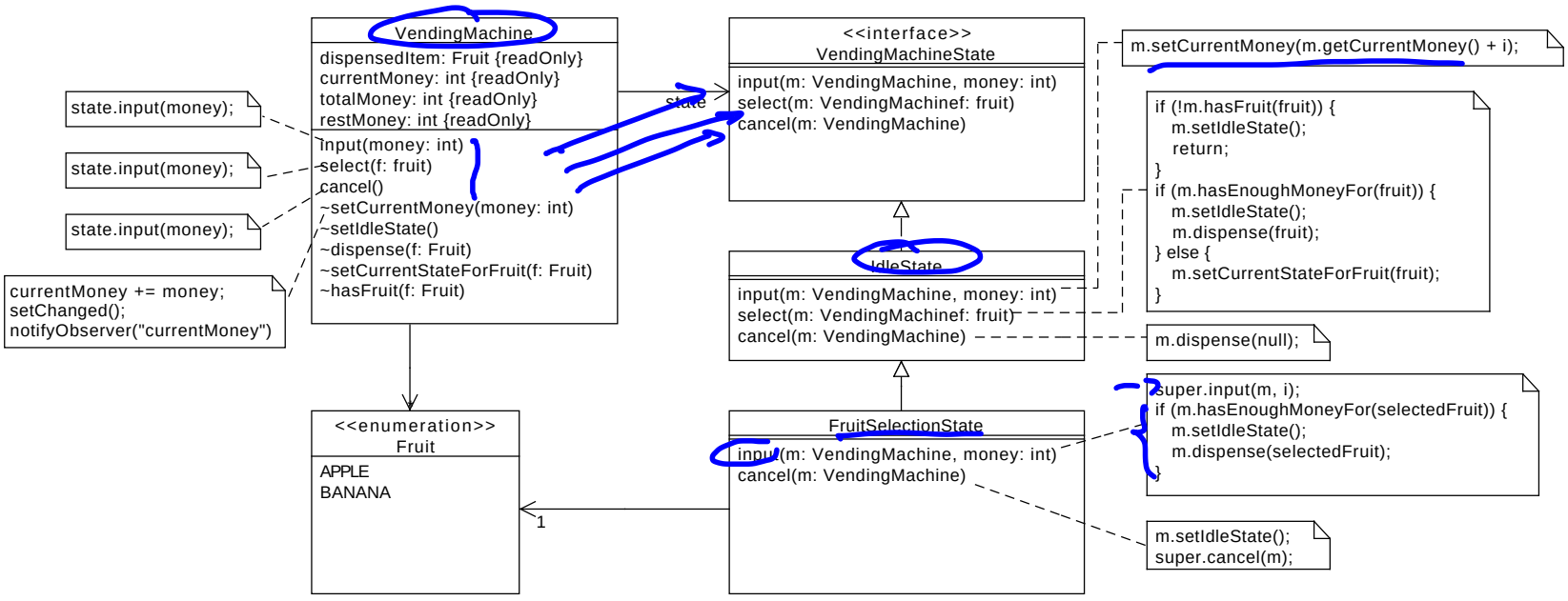


- Command line interface

```
Current Money: DKK 5
    0) Exit
    1) Input 1 DKK
    2) Input 2 DKK
    3) Input 5 DKK
    4) Select banana
    5) Select apple
    6) Cancel
Select a number (0-6):
Rest: DKK 2
Current Money: DKK 0
Dispensing: Apple
```

Vending machine Implementation

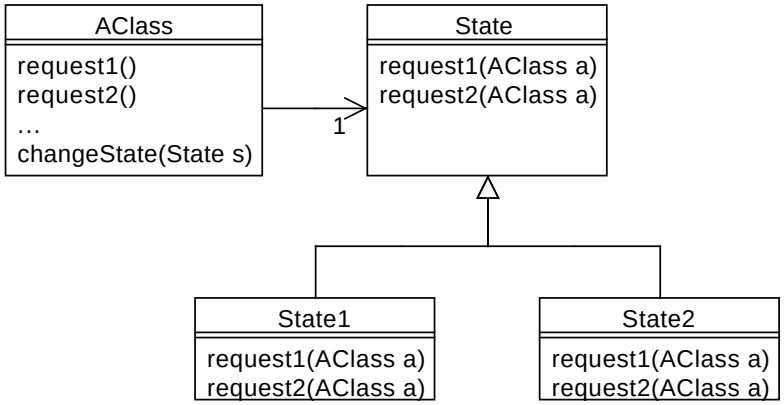
Uses the state pattern

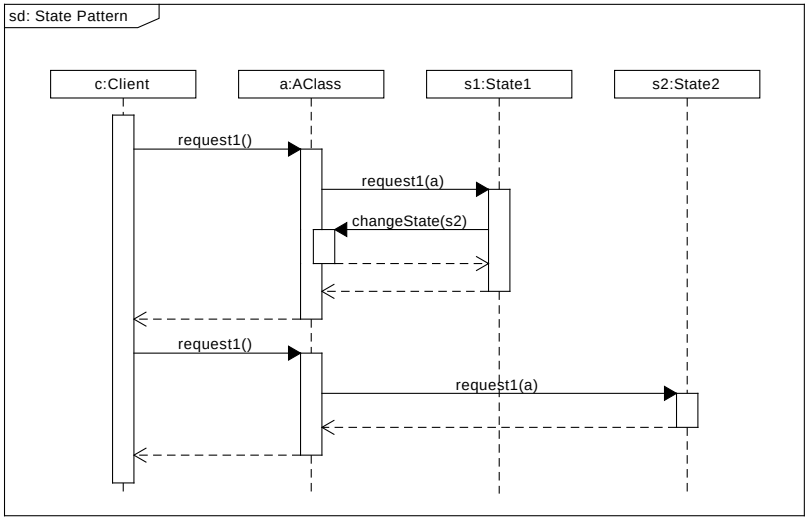


State Pattern

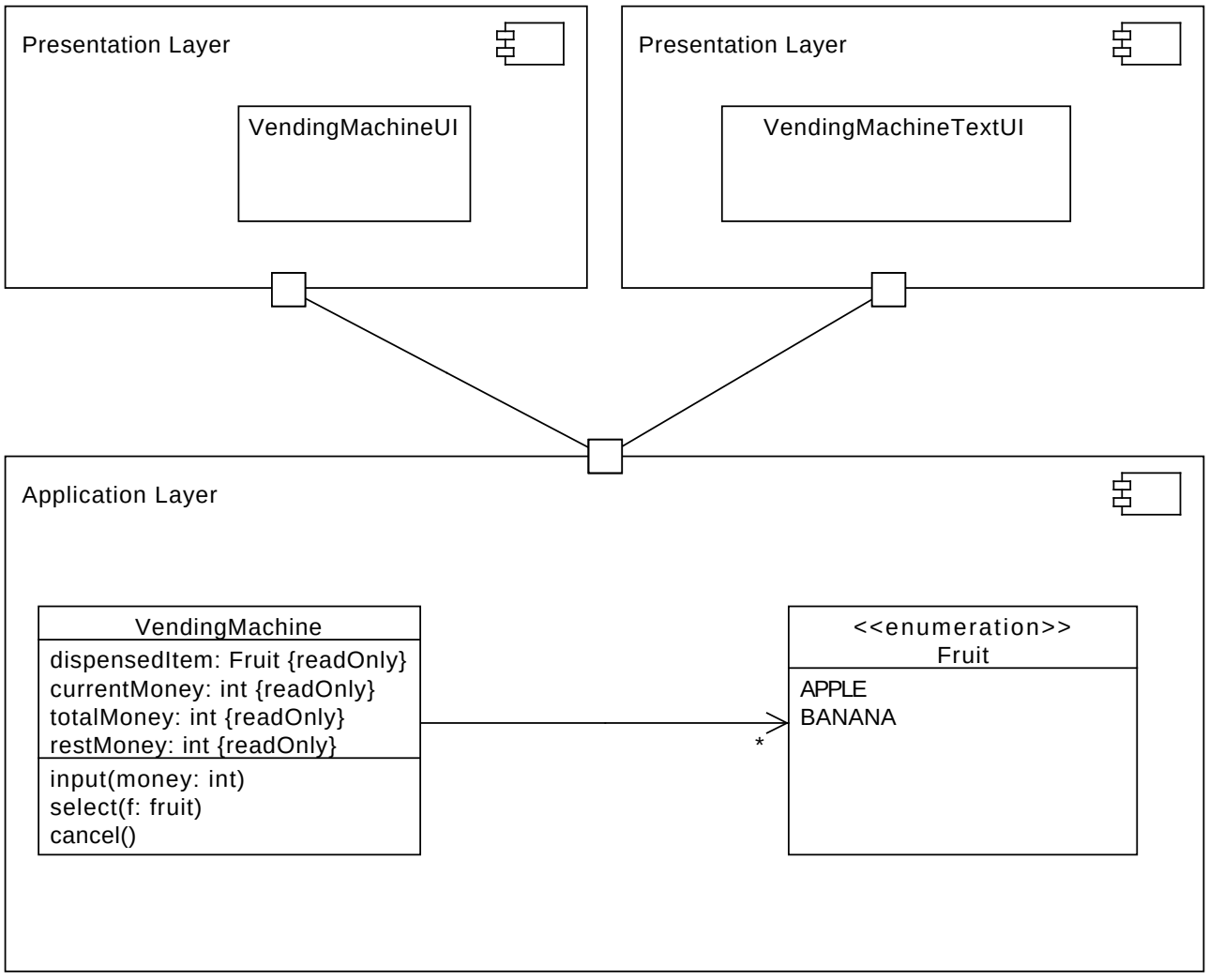
State Pattern

”Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.” Design Pattern book





Architecture

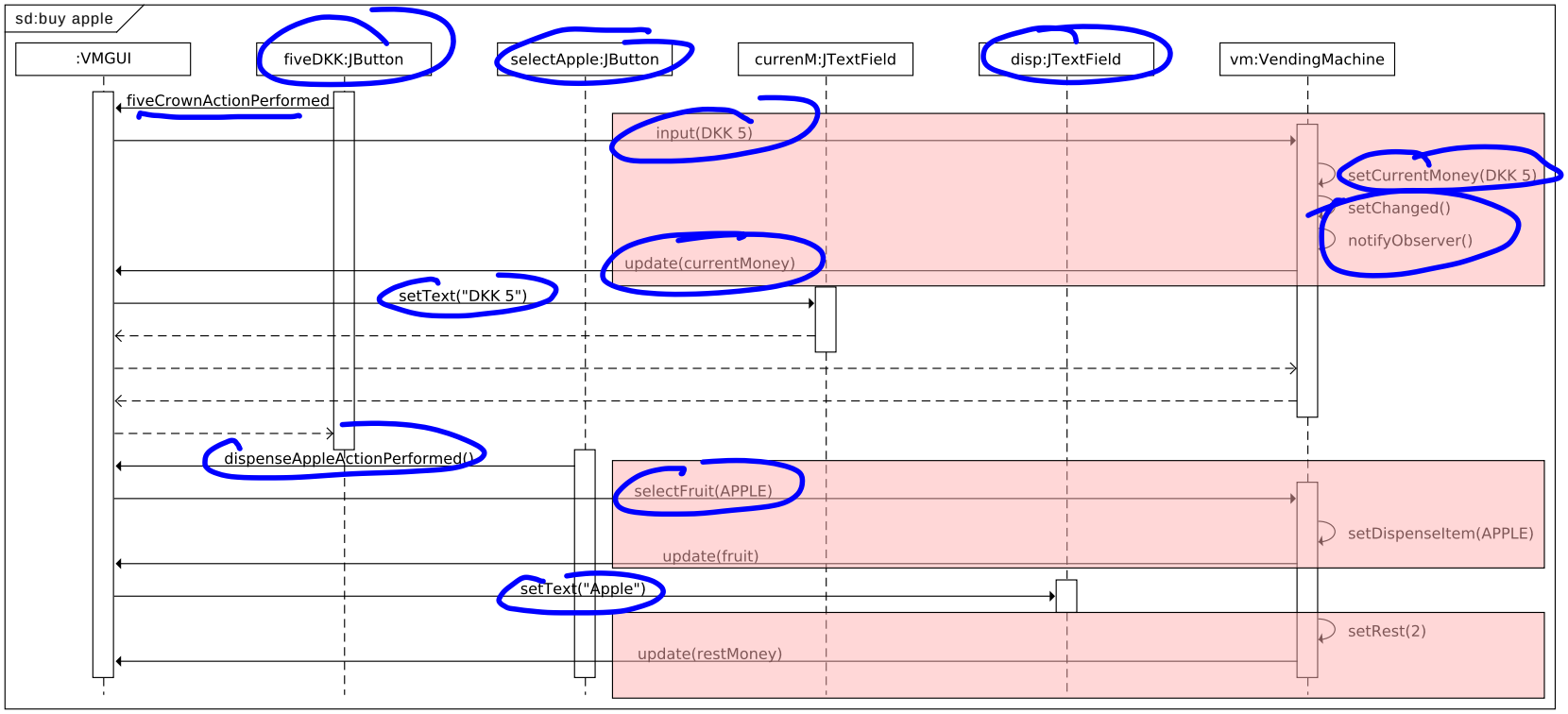


VendingMachine acts as an *facade* (Facade Design Pattern)

Presentation Layer: Swing GUI

```
public class VendingMachineUI extends implements java.util.Observer {
   private VendingMachine vendingMachine = new VendingMachine(10, 10);
   ...
   private JFrame topWindow = new JFrame();
   private JButton fiveCrowns = new JButton();
   private JTextField currentM = new JTextField();
   ...

   private void initComponents() {
      fiveCrowns.setText("DKK 5");
      fiveCrowns.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent evt) {
               vendingMachine.input(5);
            }
         }
      )
      ...
   };
   ...
   public void update(Observable o, Object arg) {
      currentM.setText("" + vendingMachine.getCurrentMoney());
      ...
   }
}
```

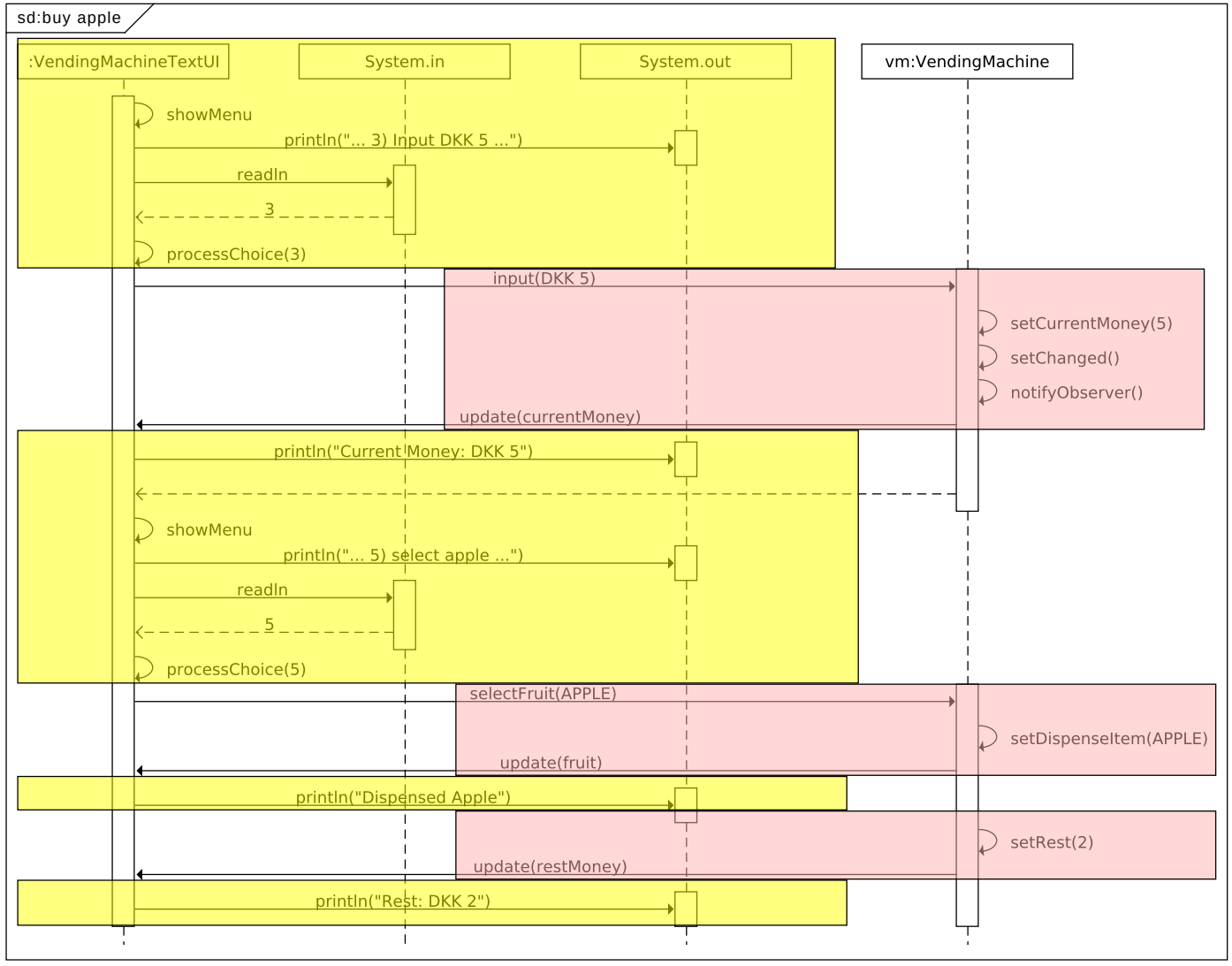
Presentation Layer: Swing GUI



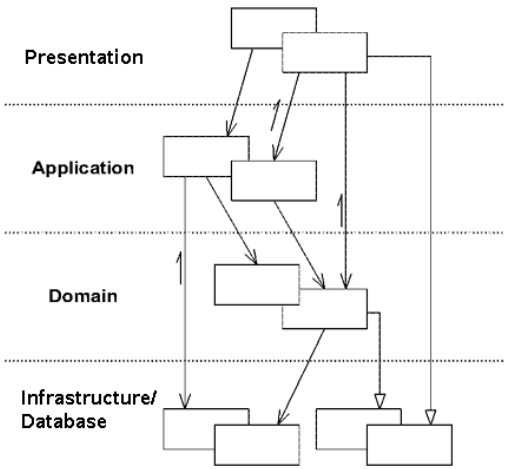
Presentation Layer: Text UI

```
public class VendingMachineTextUI implements Observer {
  VendingMachine vendingMachine;
  public static void main(String[] args) throws Exception {
    new VendingMachineTextUI(5,5).mainLoop(System.in, System.out);
  }
  public void mainLoop(InputStream in, PrintStream out) throws IOExcep
    BufferedReader rs = new BufferedReader(new InputStreamReader(in));
    do {
      showMenu(out);
      int number = Integer.valueOf(rs.readLine());
      processChoice(number, out);
    } while (number != 0);
  }
  private void processChoice(int number, PrintStream out) {
    switch (number) {
      case 3: vendingMachine.input(5); break;
      ...
    }
  }
  public void update(Observable o, Object aspect) {
    if (NotificationType.CURRENT_MONEY.equals(aspect)) {
      System.out.println("Current Money: DKK " +
        vendingMachine.getCurrentMoney());
      return;
    }
  }
}
```

Presentation Layer: Text UI



Advantages of the separation



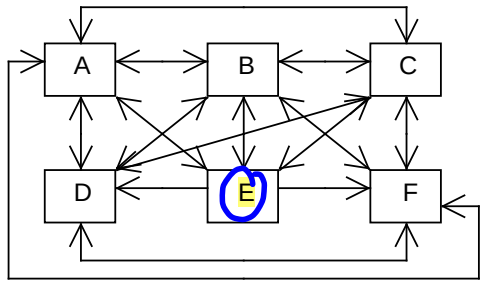
1. Presentation layer easily changed
2. Additional presentation layers can be added easily without having to reimplement the business logic
   - mobile app in addition to desktop and Web application
3. Automatic tests: test the application and domain layer: test ”under the GUI”

Contents

S.O.L.I.D. principles

Mentioned in Robert C. Martin's book "Agile Principles, Patterns, and Practices in C#"

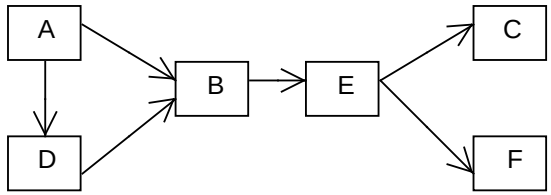
- `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`



Bad code

- brittle
- fragile
- code can't be reused

→ high coupling



Principles help

→ Manage complexity and dependencies

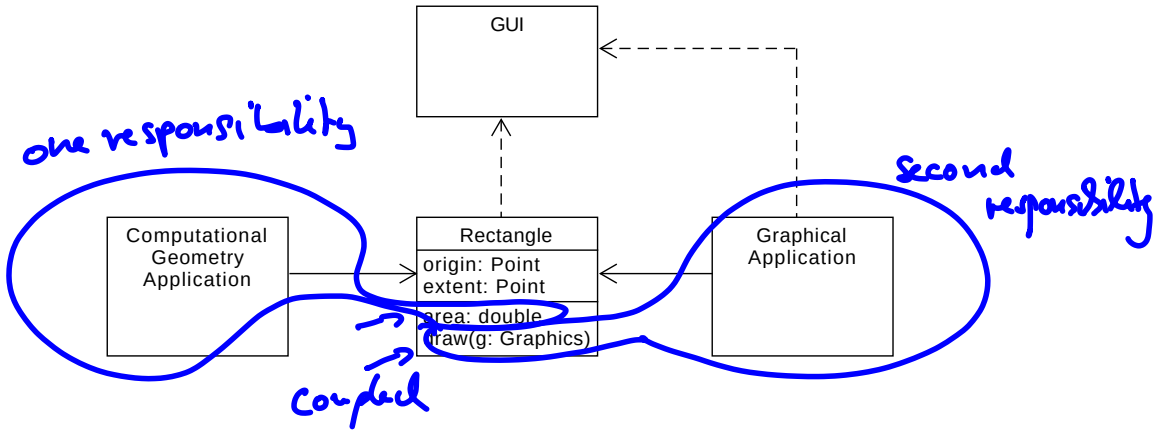
→ Write maintainable, easy to read code

S.O.L.I.D. Principles

- Single responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

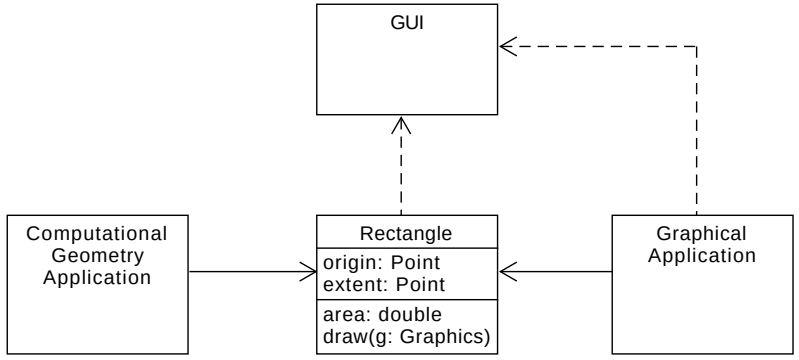
Single responsibility principle

- ”A class should have only a single responsibility”

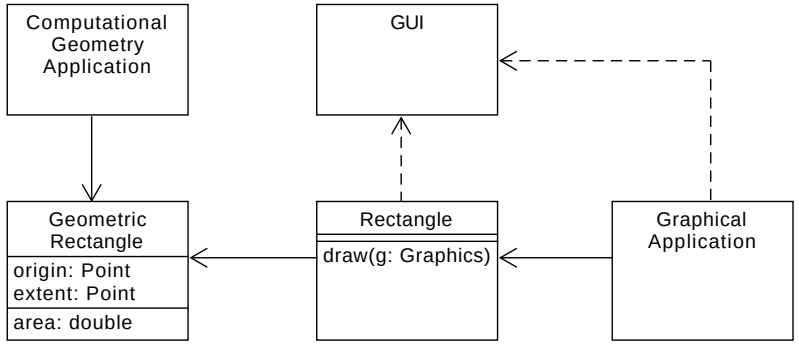


Single responsibility principle

- ”A class should have only a single responsibility”

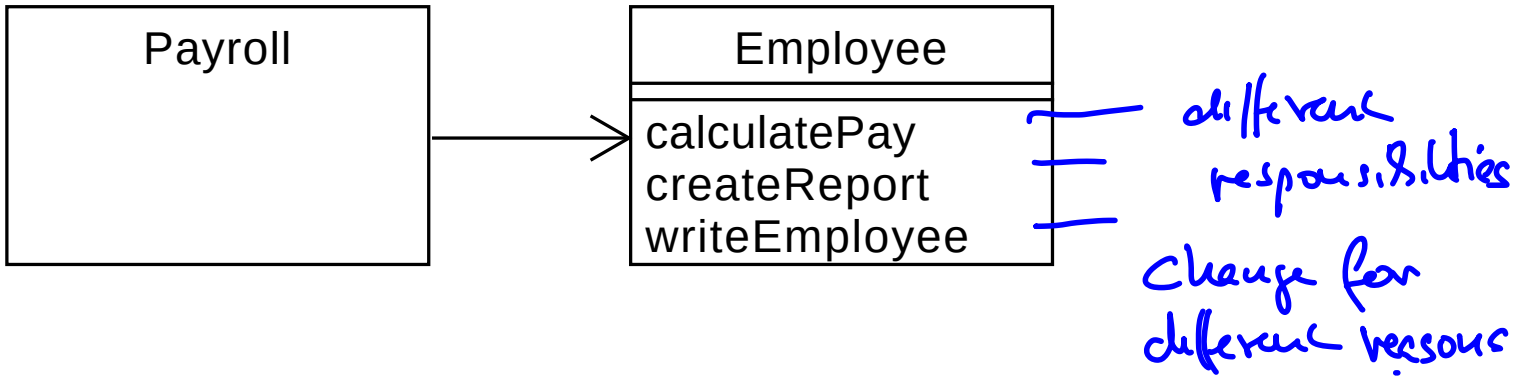


- Possible solution

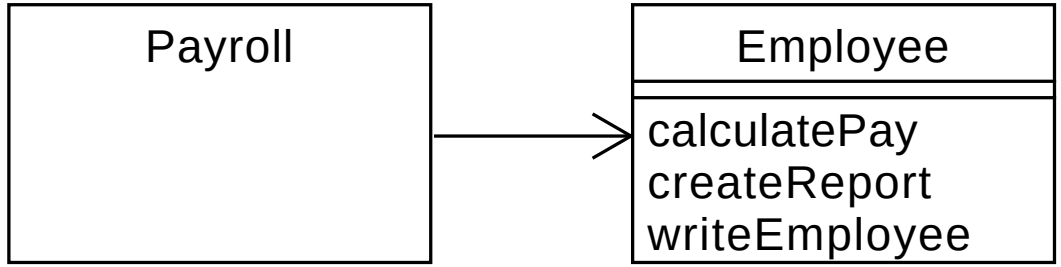


- Responsibility $\rightarrow$ axis of change (cohesion!)

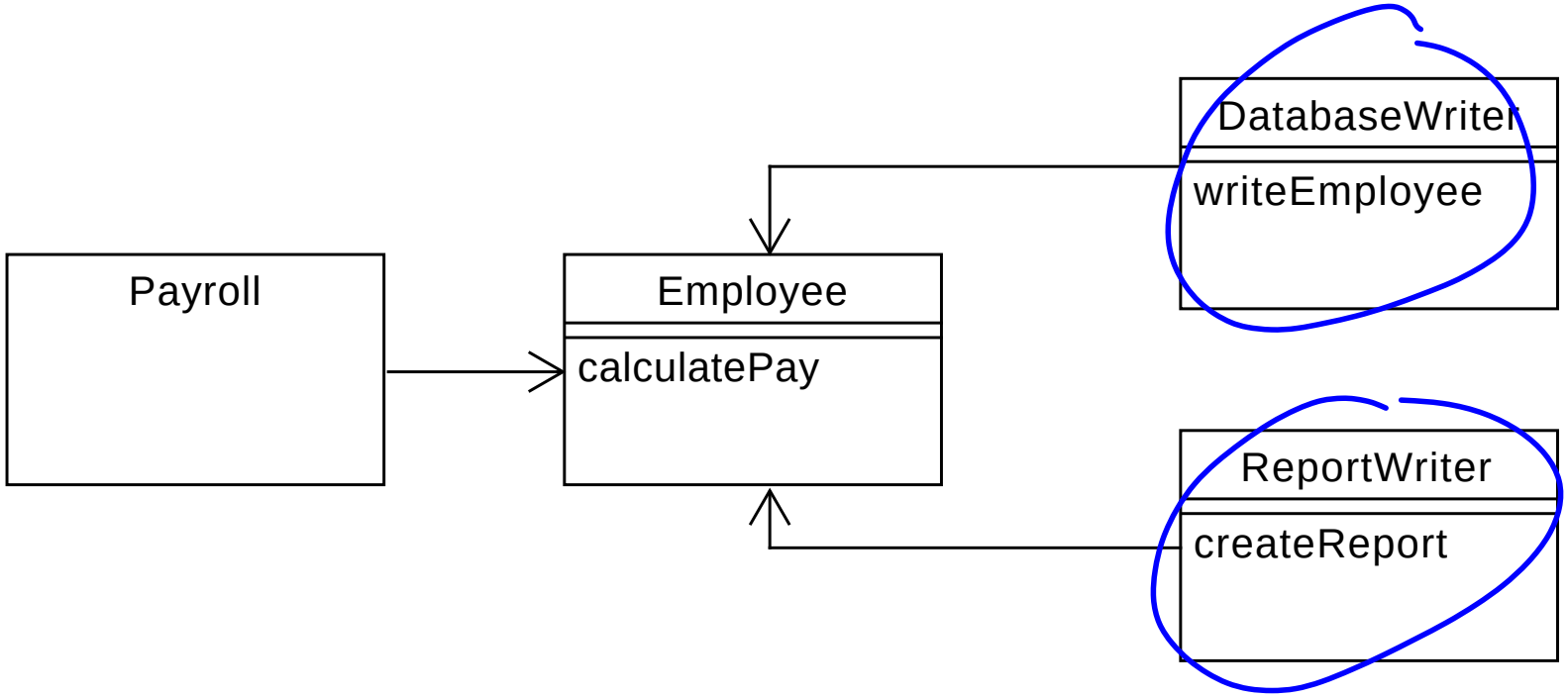
Single responsibility principle



Single responsibility principle



- Possible solutions

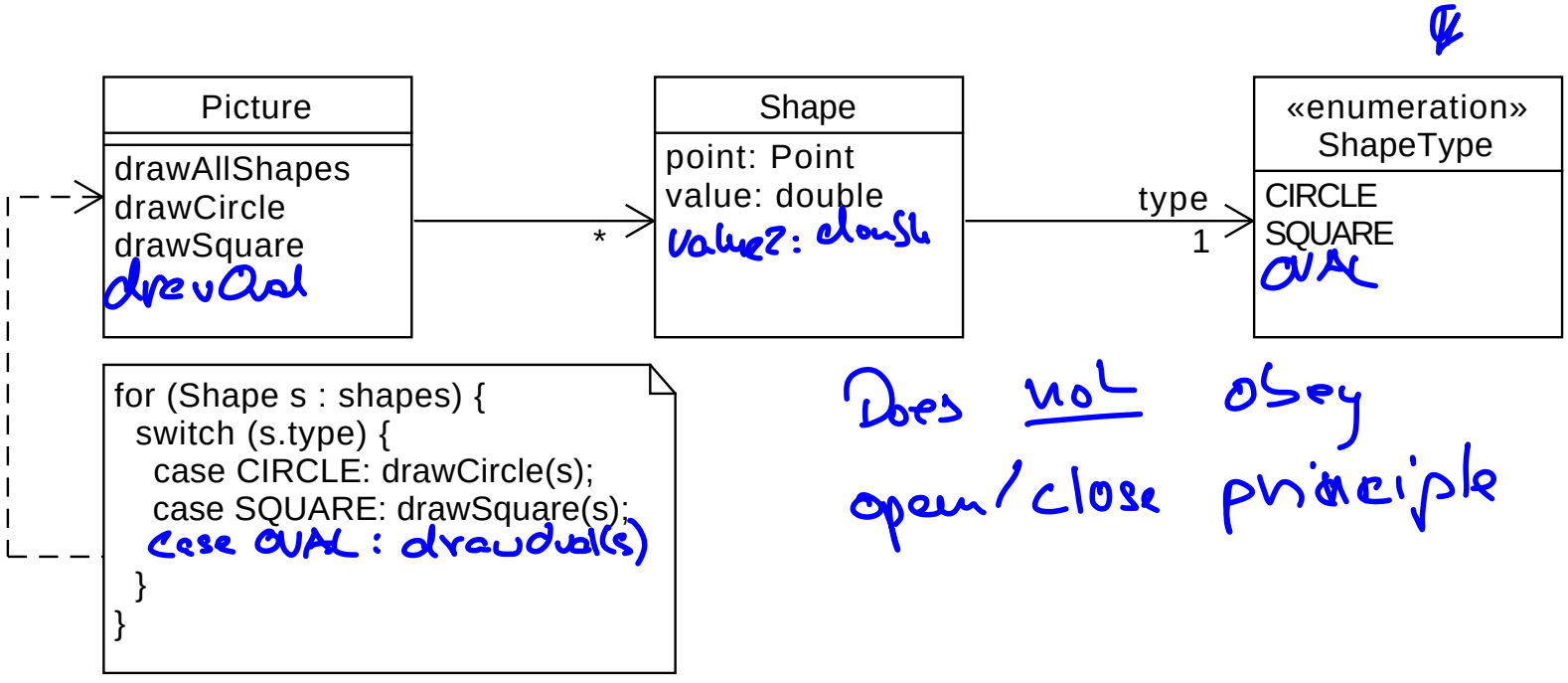


Open/Closed Principle

- Bertrand Meyer (88): ”Modules should be open for extension, but closed for modification.”

Open/Closed Principle

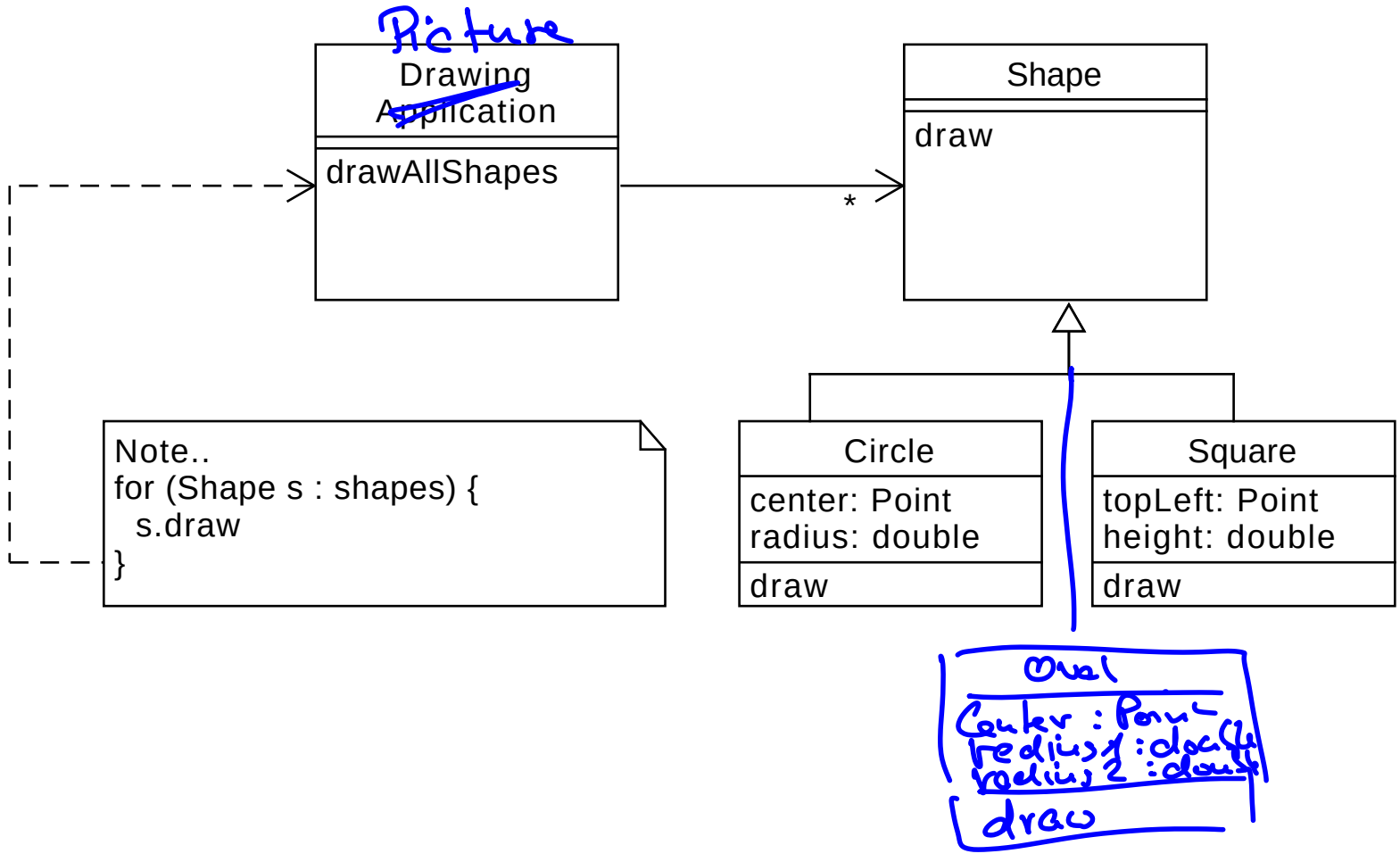
- Bertrand Meyer (88): ”Modules should be open for extension, but closed for modification.”



Open/Closed Principle

satisfies open/close principle

- Possible Solution



Next Week

- Remaining 3 S.O.L.I.D. principles
- Hexagonal architecture
- Layered architecture: Persistence Layer