

A photograph of a red squirrel climbing a tree trunk. The squirrel is facing left, its reddish-brown fur contrasting with the dark, textured bark of the tree. It is gripping the trunk with its front paws and a small branch. The background is filled with green foliage and other tree trunks, creating a natural, woodland setting.

## M8(b)- Inversion of Control

---

Jin L.C. Guo

Image source: <https://www.goodfreephotos.com/albums/animals/mammals/red-squirrel-climbing-up-a-tree.jpg>

# Objective

- Be able to Use Callback to achieve decoupling
- Be able to use the Observer design pattern effectively;
- Event Handling in GUI applications
- Understand the concept of an application framework;
- Understand the Model-View-Controller Decomposition;
- Be able to use the Visitor Design Pattern effectively;
- Be able to determine when to used different design patterns effectively.

# Objective

- Be able to Use Callback to achieve decoupling
- Be able to use the Observer design pattern effectively;
- Event Handling in GUI applications
- Understand the concept of an application framework;
- Understand the Model-View-Controller Decomposition;
- Be able to use the Visitor Design Pattern effectively;
- Be able to determine when to used different design patterns effectively.

# Event

- A notification that something interesting has happened.
- Examples in Graphic Interface?

*Move a mouse*

*User click a button*

*Press a key*

*Mouse press and drag*

*Menu item is selected*

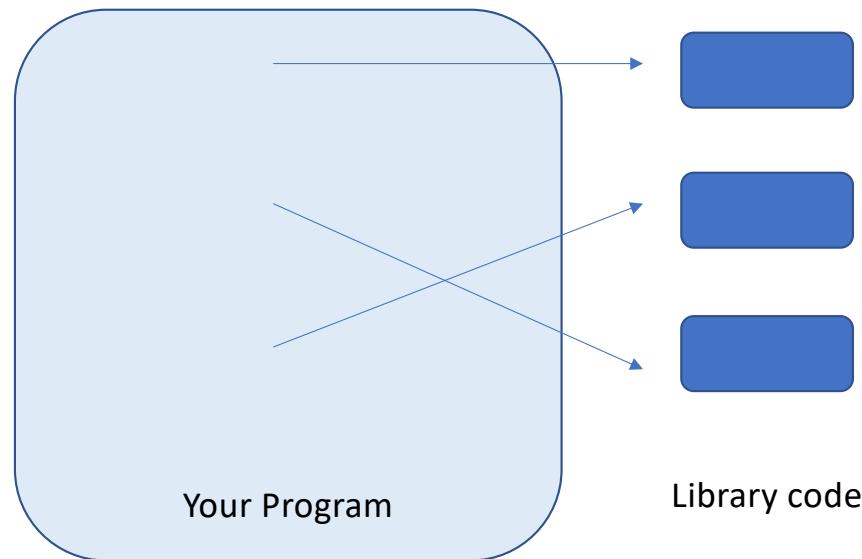
*Window is closed*

*Popup window is hidden*



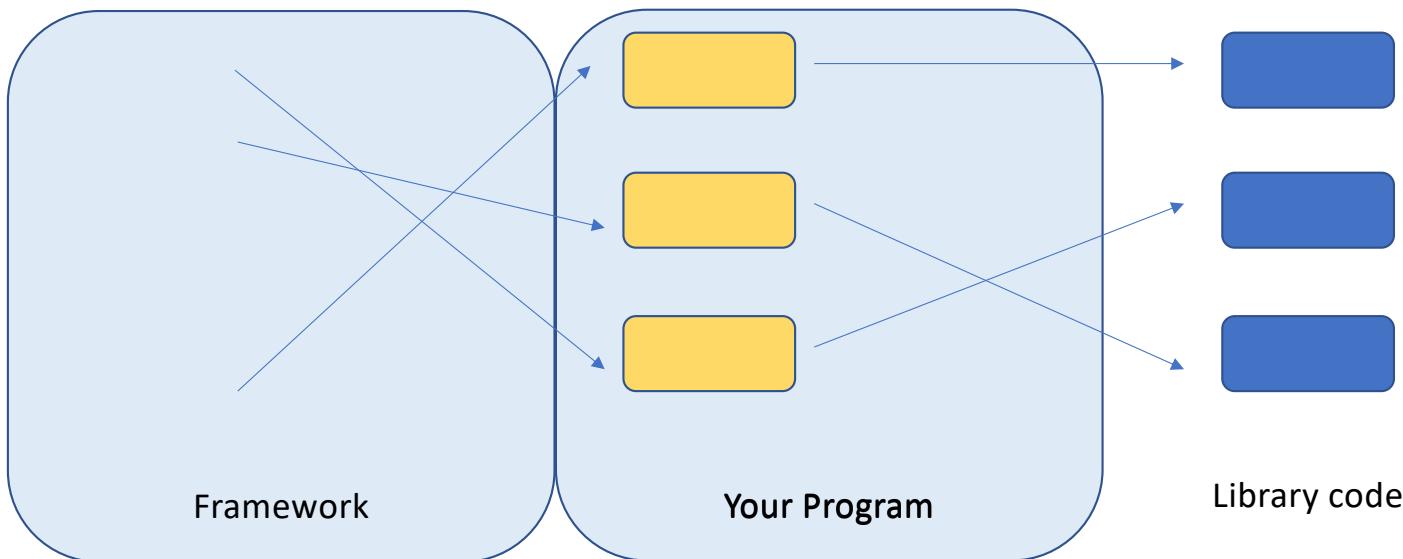
# Library vs Framework

injecting  
library  
code  
into your  
program



**public static void main(String[] args)**

# Library vs Framework



You provide code  
for me framework  
to call

# Launch JavaFX framework

```
public class MyApplication extends Application
{
    /**
     * Launches the application.
     * @param pArgs This program takes no argument.
     */
    public static void main(String[] pArgs)
    {
        launch(pArgs);
    }

    @Override
    public void start(Stage pPrimaryStage)
    {
        //Setup the stage
        pPrimaryStage.show();
    }
}
```

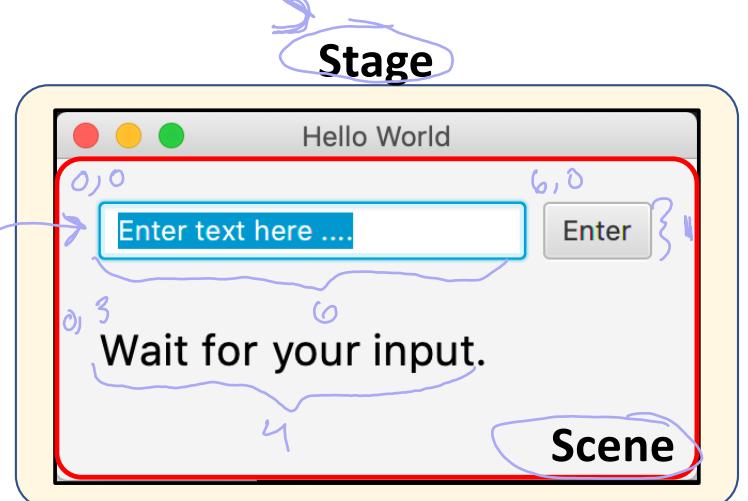
Annotations and handwritten notes:

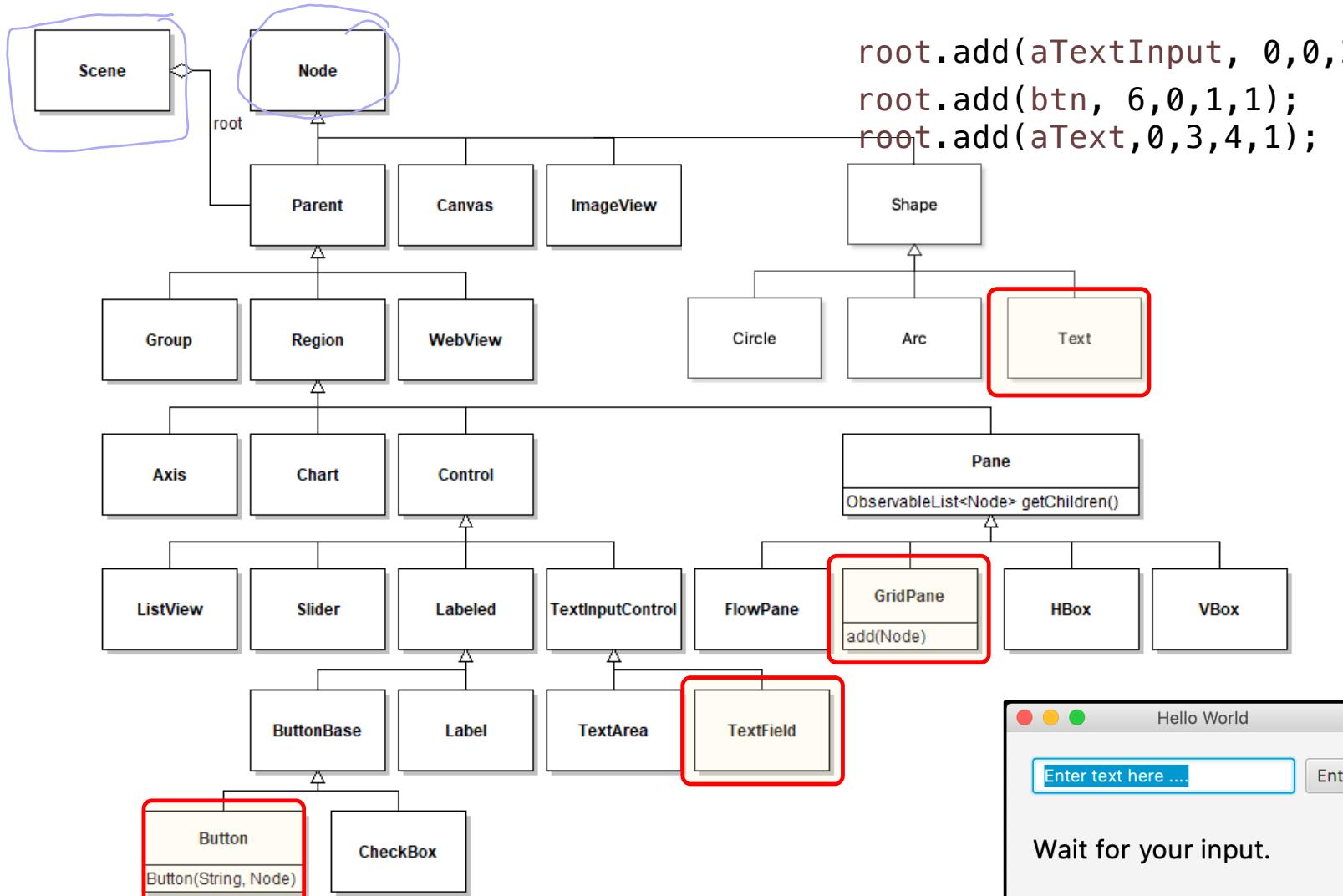
- A blue wavy line highlights the entire class definition.
- A blue bracket underlines the word `extends`.
- A blue bracket underlines the parameter `pArgs` in the `main` method, with the handwritten note "size o array" next to it.
- A blue bracket underlines the `start` method.
- A blue bracket underlines the parameter `pPrimaryStage` in the `start` method.
- A blue bracket underlines the call to `pPrimaryStage.show()`.

customize Stage

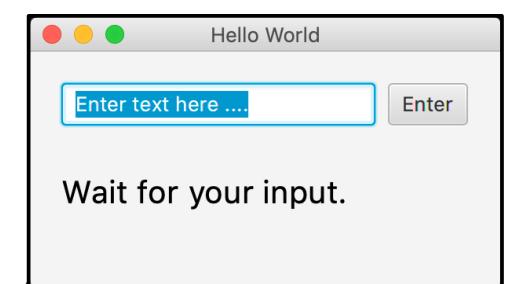
```
@Override  
public void start(Stage pPrimaryStage)  
{  
    ... //Other setup steps  
  
    GridPane root = new GridPane();  
    root.add(aTextInput, 0,0,6,1);  
    root.add(btn, 6,0,1,1); spanx=spany  
    root.add(aText,0,3,4,1);  
  
    Scene scene = new Scene(root, Width, Height);  
    ... //Other setup steps  
  
    primaryStage.setScene(scene);  
  
    pPrimaryStage.show(); - Show stage  
}
```

constructed in launch method



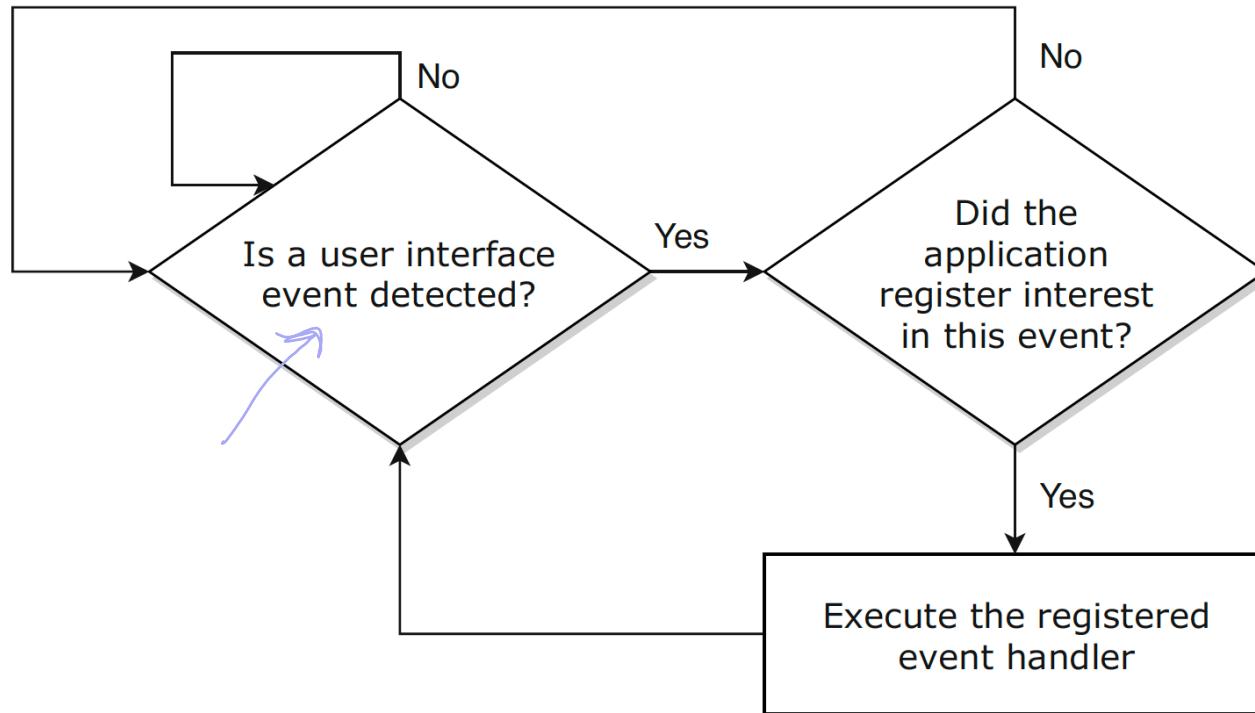


Scene scene = new Scene(root, **Width**, **Height**);



*after start method*

## When does event handling happen?



# Text Display Example

register event  
handling code

use JavaFX  
functionality

take event + handle  
as input and use  
lambda expression



```
Text aText = new Text();
```

```
TextField aTextInput = new TextField();
```

```
aTextInput.setOnAction(actionEvent) -> aText.setText(aTextInput.getText());
```

```
Button btn = new Button();
```

```
btn.setOnAction(actionEvent) -> aText.setText(aTextInput.getText());
```

↑  
simple example of event handling

more one more complex problems

## Recap: Objective

- Be able to Use Callback to achieve decoupling
- Be able to use the Observer design pattern effectively;
- Event Handling in GUI applications
- Understand the concept of an application framework;
- Understand the **Model-View-Controller Decomposition;**
- Be able to use the Visitor Design Pattern effectively;
- Be able to determine when to used different design patterns effectively.

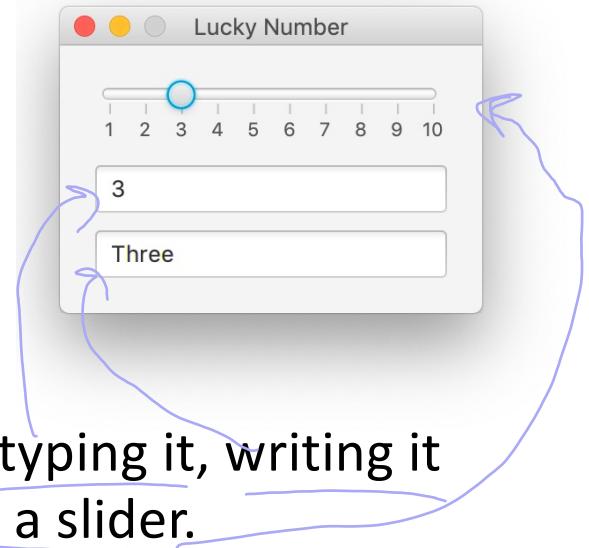
break down,  
decouple

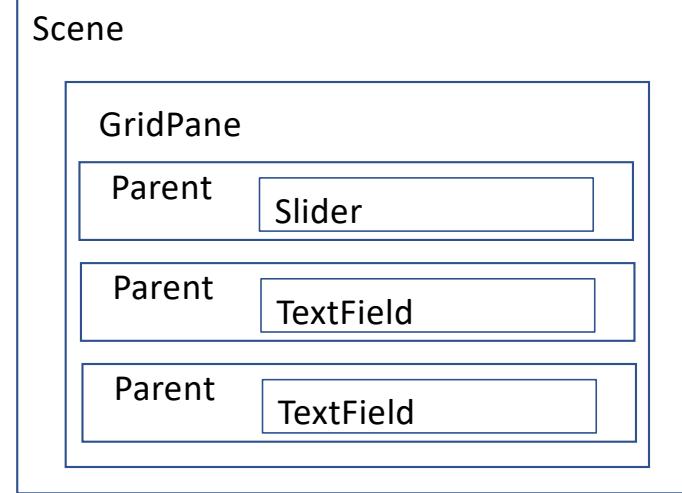
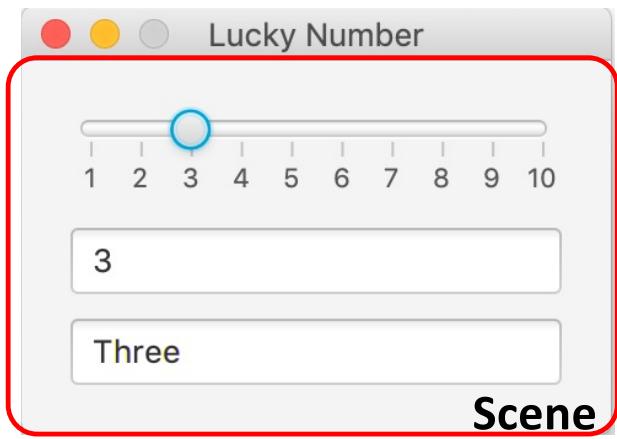
# Lucky Number Example

The user should be able to select a number between 1 and 10 inclusively.

The selection should be performed through either typing it, writing it out in the corresponding fields, or selecting it from a slider.

The current selection should also be viewed in the integer and text fields and the slider.





# Problem Decomposition

**IntegerPanel**

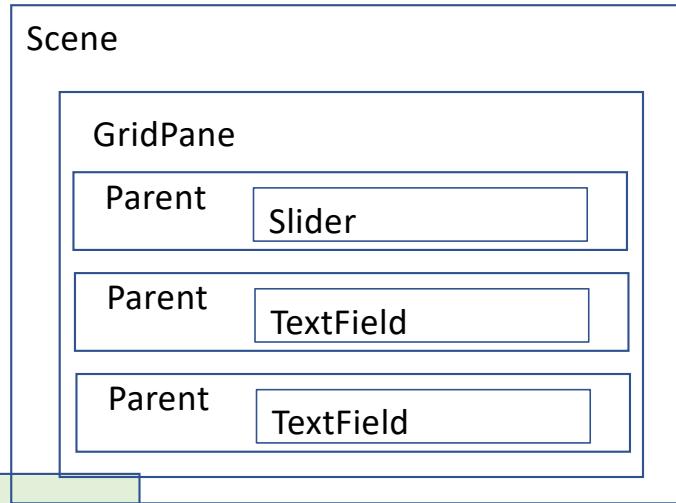
```
int aSelection  
TextField aText  
void setSelection(int)  
int getSelection()
```

**SliderPanel**

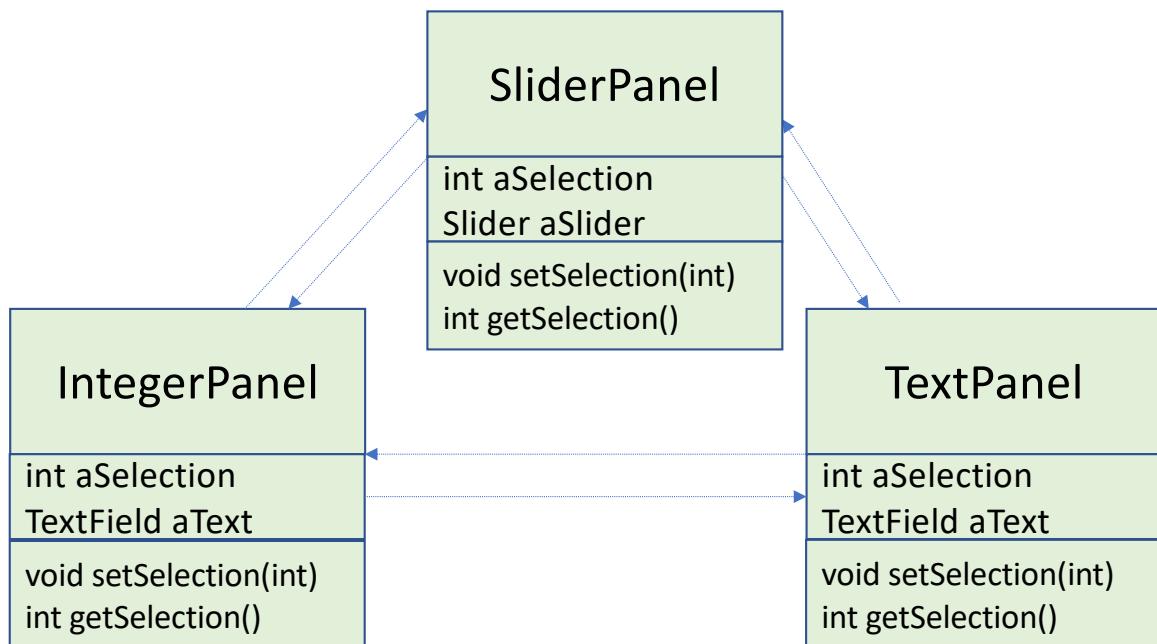
```
int aSelection  
Slider aSlider  
void setSelection(int)  
int getSelection()
```

**TextPanel**

```
int aSelection  
TextField aText  
void setSelection(int)  
int getSelection()
```



# Problem Decomposition



**High Coupling**  
*Components are inter-dependent*

**Low Extensibility**  
*hard to add/remove selection mechanism*

# MVC Decomposition

data  
↓  
rendering  
↓

## Model – View – Controller

- Model + view  
-.

- any {
  - Design pattern
  - Architectural pattern
  - Guideline to separate concerns

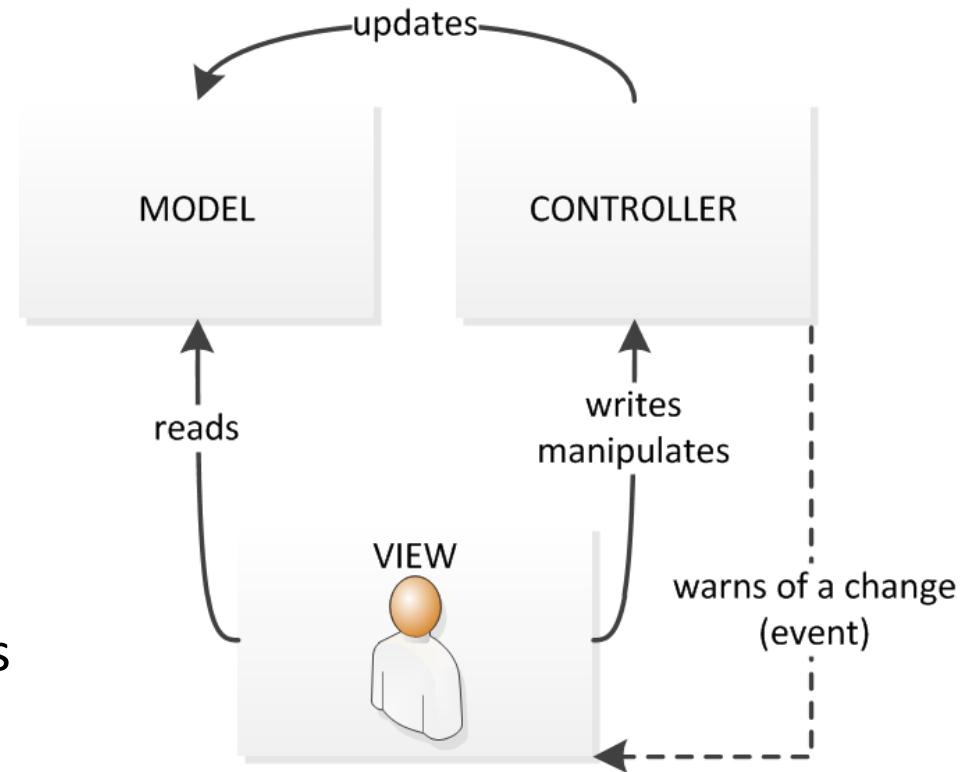
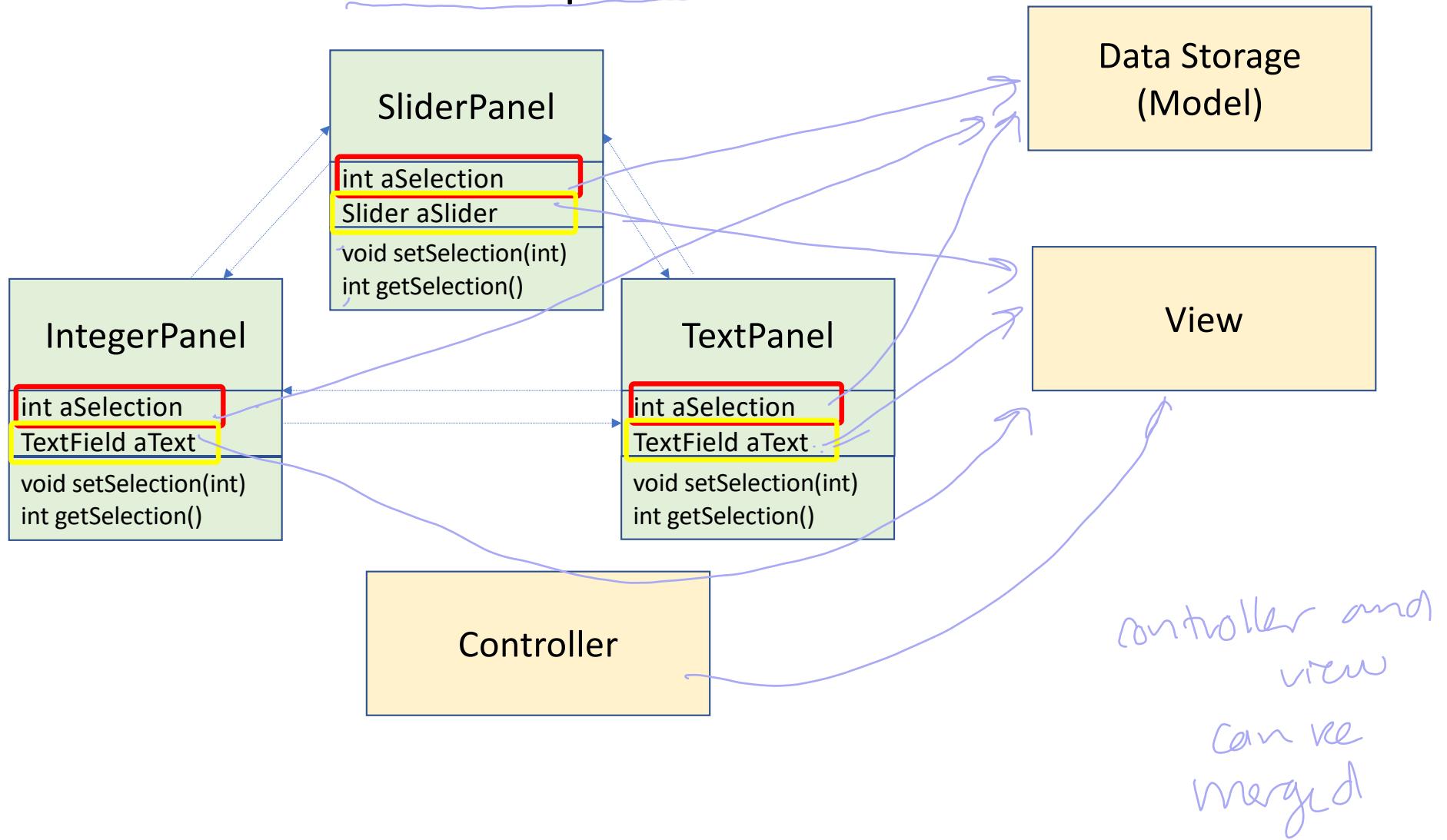
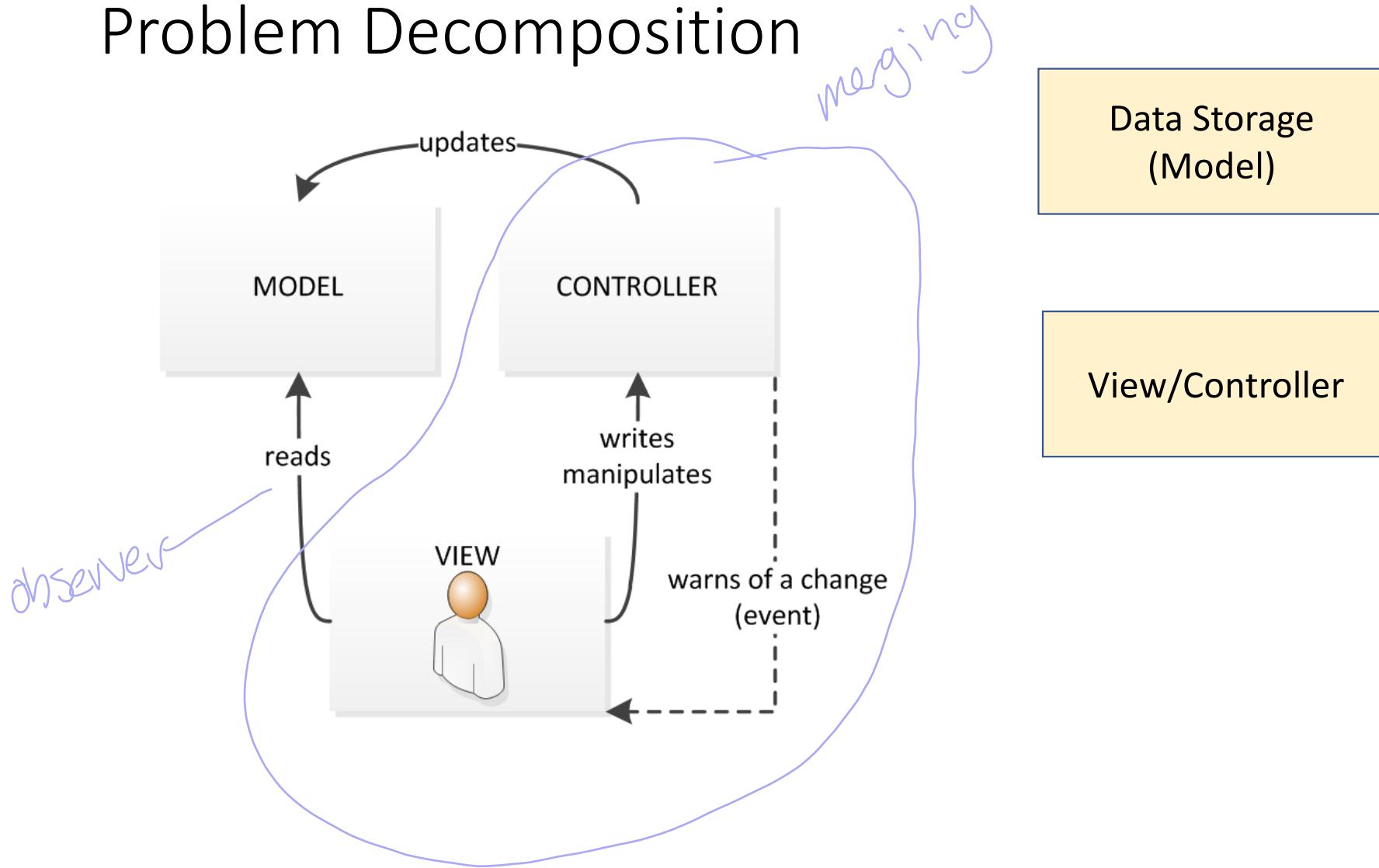


Image Source: <https://upload.wikimedia.org/wikipedia/commons/6/63/ModeleMVC.png>

# Problem Decomposition



# Problem Decomposition



Data Storage  
(Model)

View/Controller

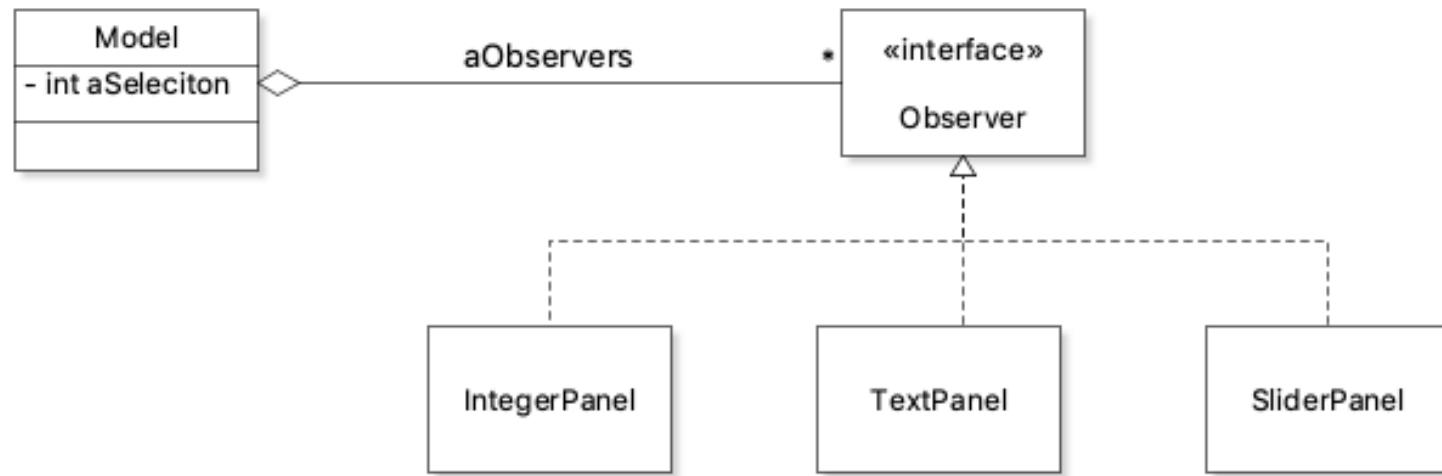
# Activity

- Improve the design using Observer Pattern and MVC decomposition.

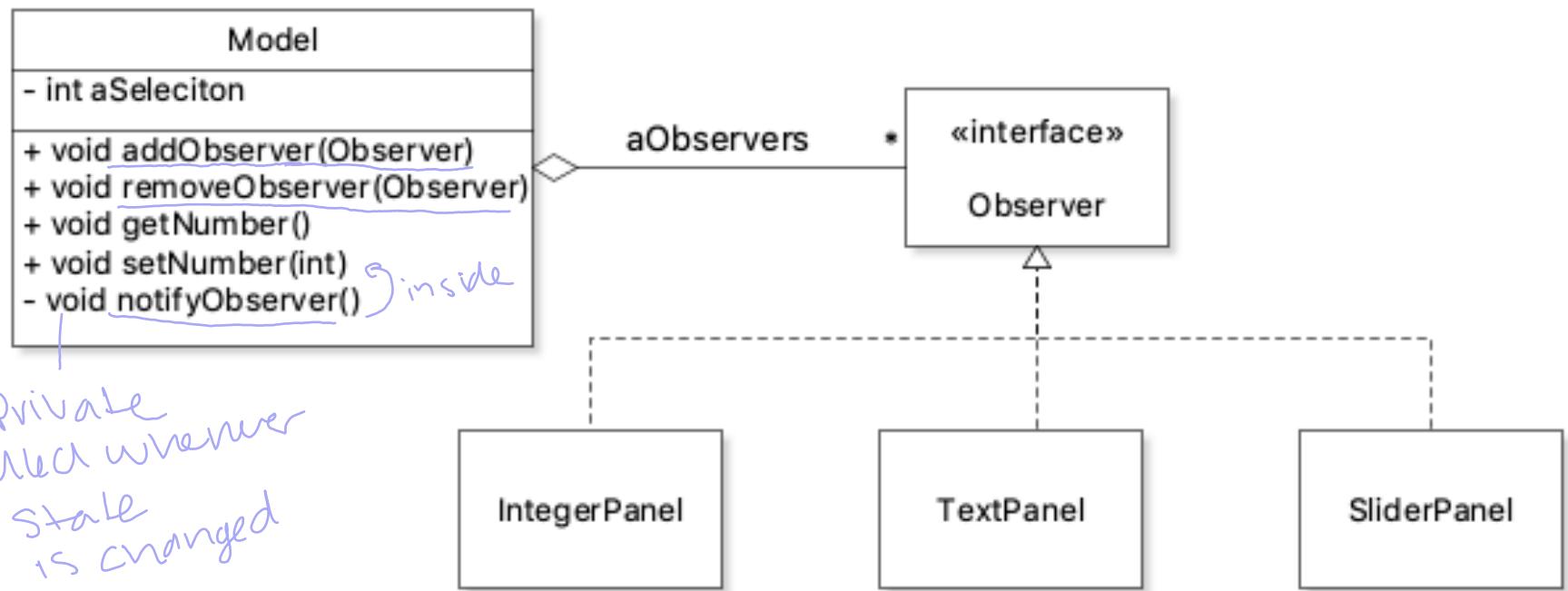
observe the model

# Activity: Applying Observer in MVC

- What methods should be included in Model?



Panels are observers of the model



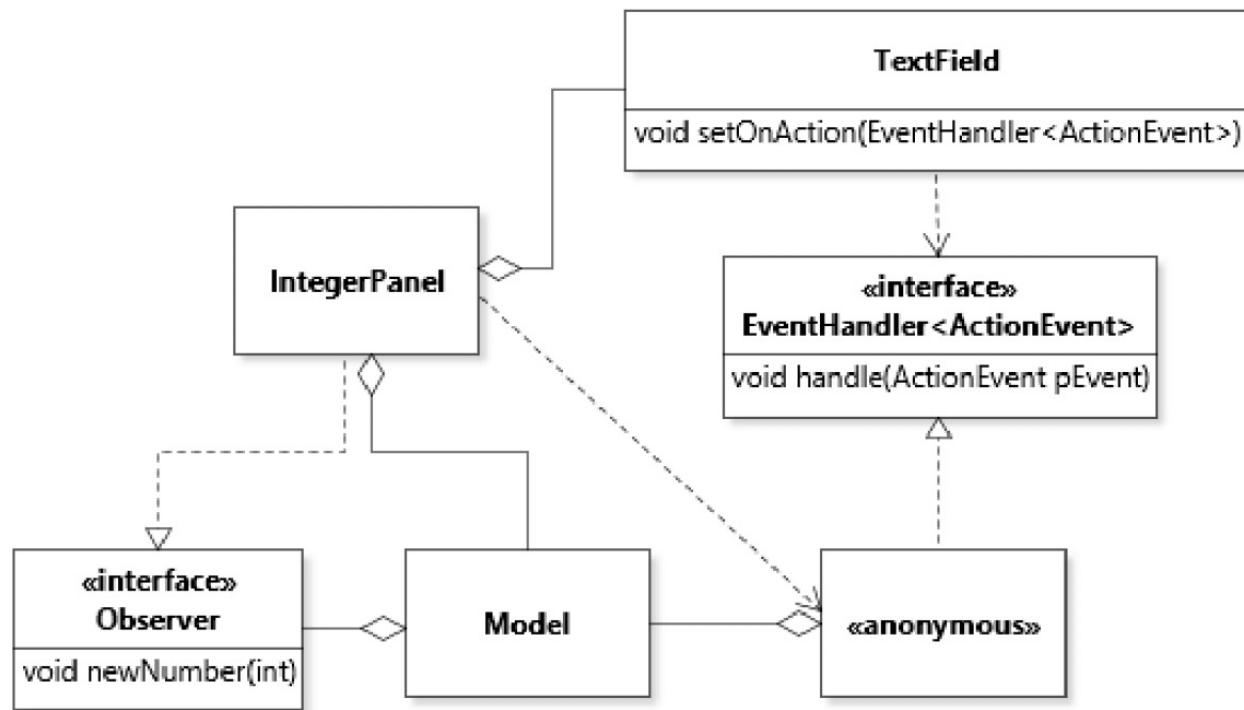
```
/**  
 * Abstract observer role for the model.  
 */  
interface Observer  
{  
    void newNumber(int pNumber);  
}
```

```
class IntegerPanel extends Parent implements Observer
{
    private TextField aText = new TextField();
    private Model aModel;
    ...
    ...
    @Override
    public void newNumber(int pNumber)
    {
        aText.setText(new Integer(pNumber).toString());
    }
}
```

Call aModel.setNumber(lInteger);

```
/**  
 * Constructor.  
 */  
IntegerPanel(Model pModel)  
{  
    aModel = pModel;  
    aModel.addObserver(this);  
    aText.setWidth(LuckyNumber.WIDTH);  
    aText.setText(new Integer(aModel.getNumber()).toString());  
    getChildren().add(aText);  
  
    aText.setOnAction(new EventHandler<ActionEvent>(){  
        @Override  
        public void handle(ActionEvent pEvent){  
            int lInteger = 1;  
            try{  
                lInteger = Integer.parseInt(aText.getText());  
            } catch(NumberFormatException pException ){  
                //Code to handle exception  
            }  
            aModel.setNumber(lInteger); -Change model state  
        }  
    });  
}
```

anonymous  
class



# Objective

- Be able to Use Callback to achieve decoupling
- Be able to use the Observer design pattern effectively;
- Event Handling in GUI applications
- Understand the concept of an application framework;
- Understand the Model-View-Controller Decomposition;
- Be able to use the Visitor Design Pattern effectively;
- Be able to determine when to used different design patterns effectively.



## University

### Faculty of Art

English

Philosophy

Sociology

...

### Faculty of Science

CS

Biology

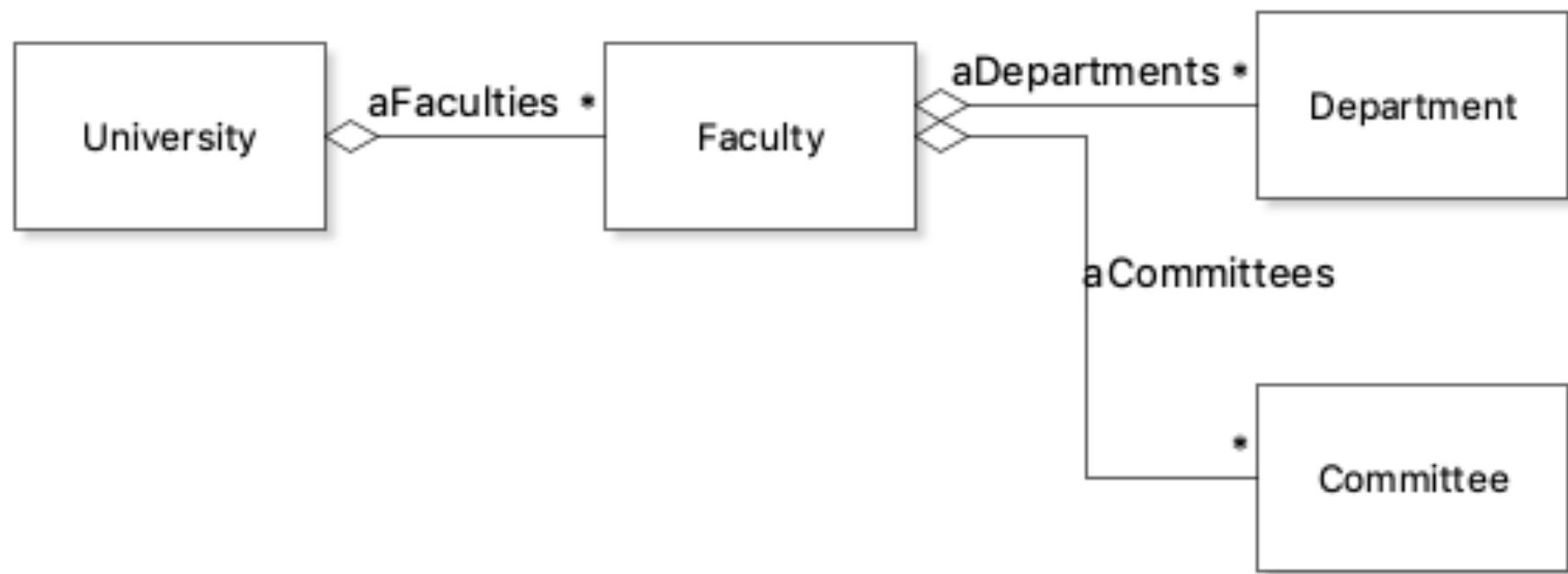
Physics

...

Academic Committee

Scholarship Committee

Students Committee



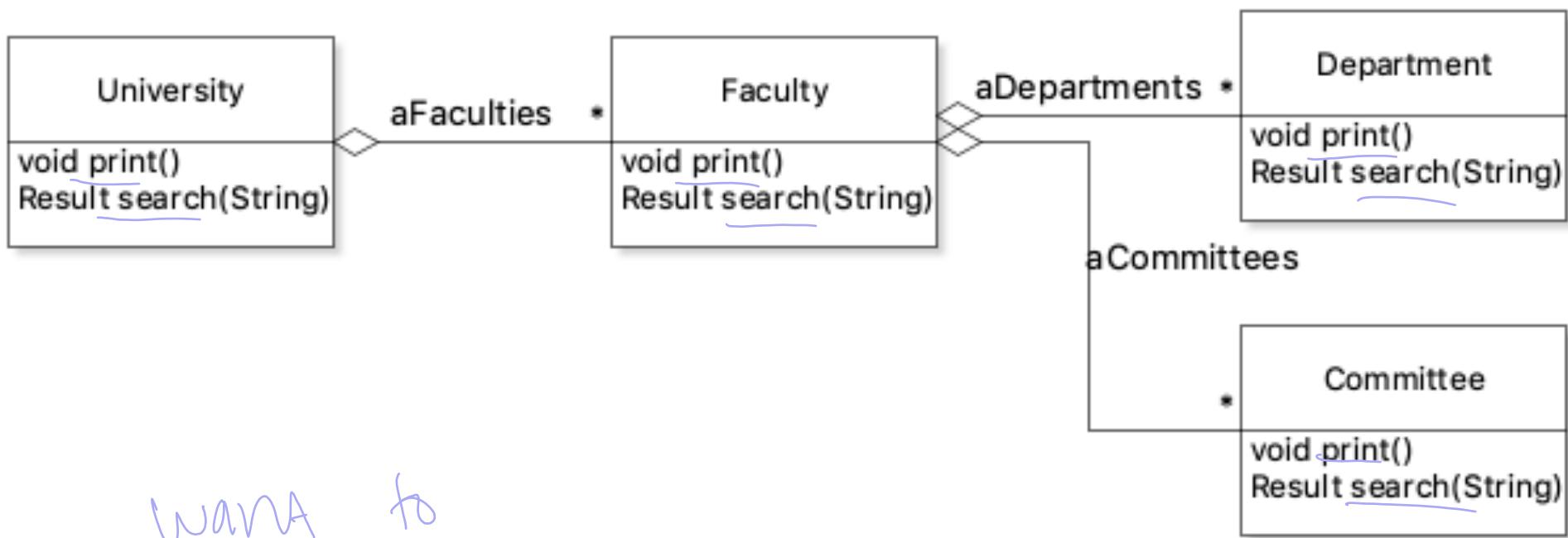
# Add functionalities to aggregate

Print annual report for every organizations in university

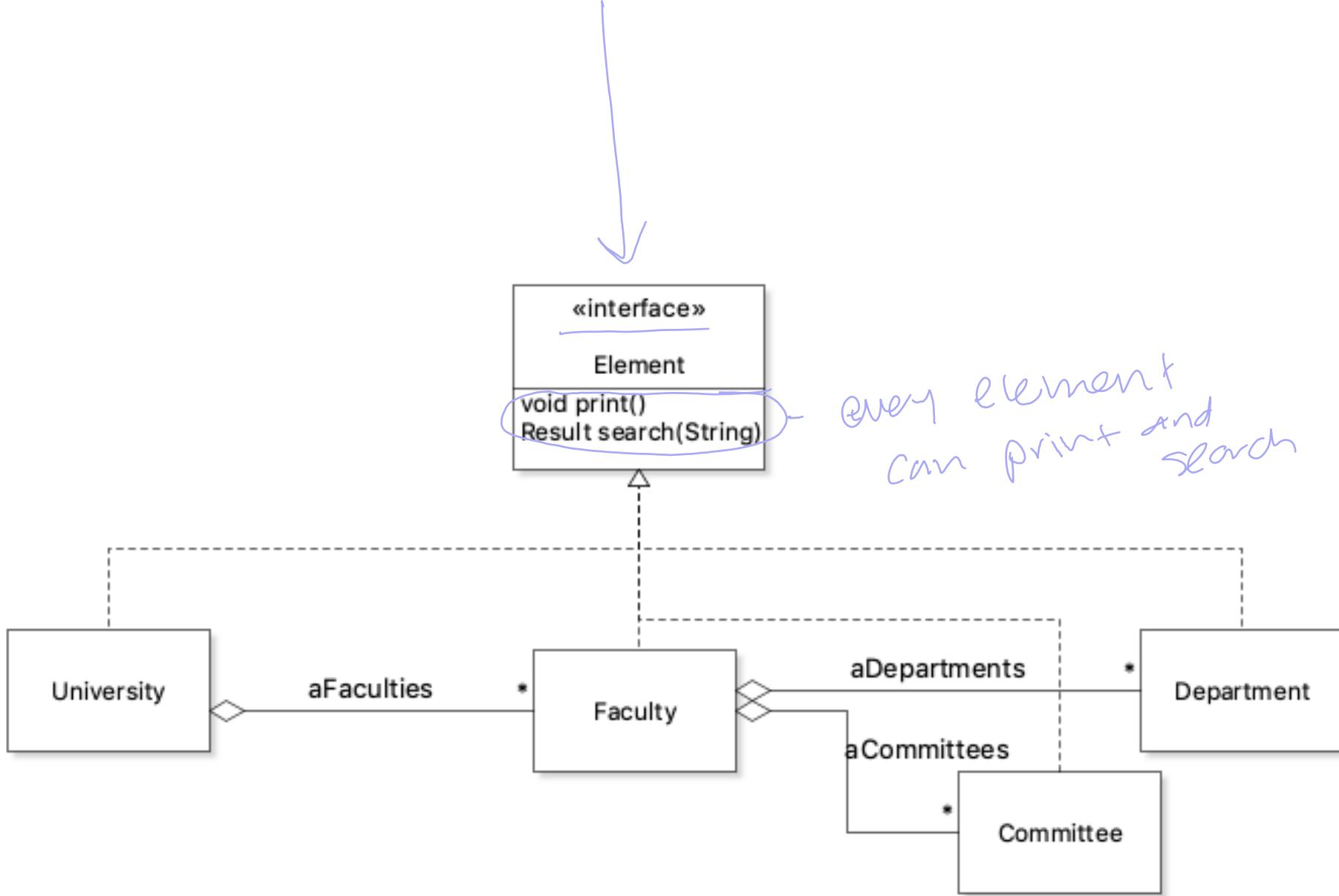
Search if one person belongs to any organization in university

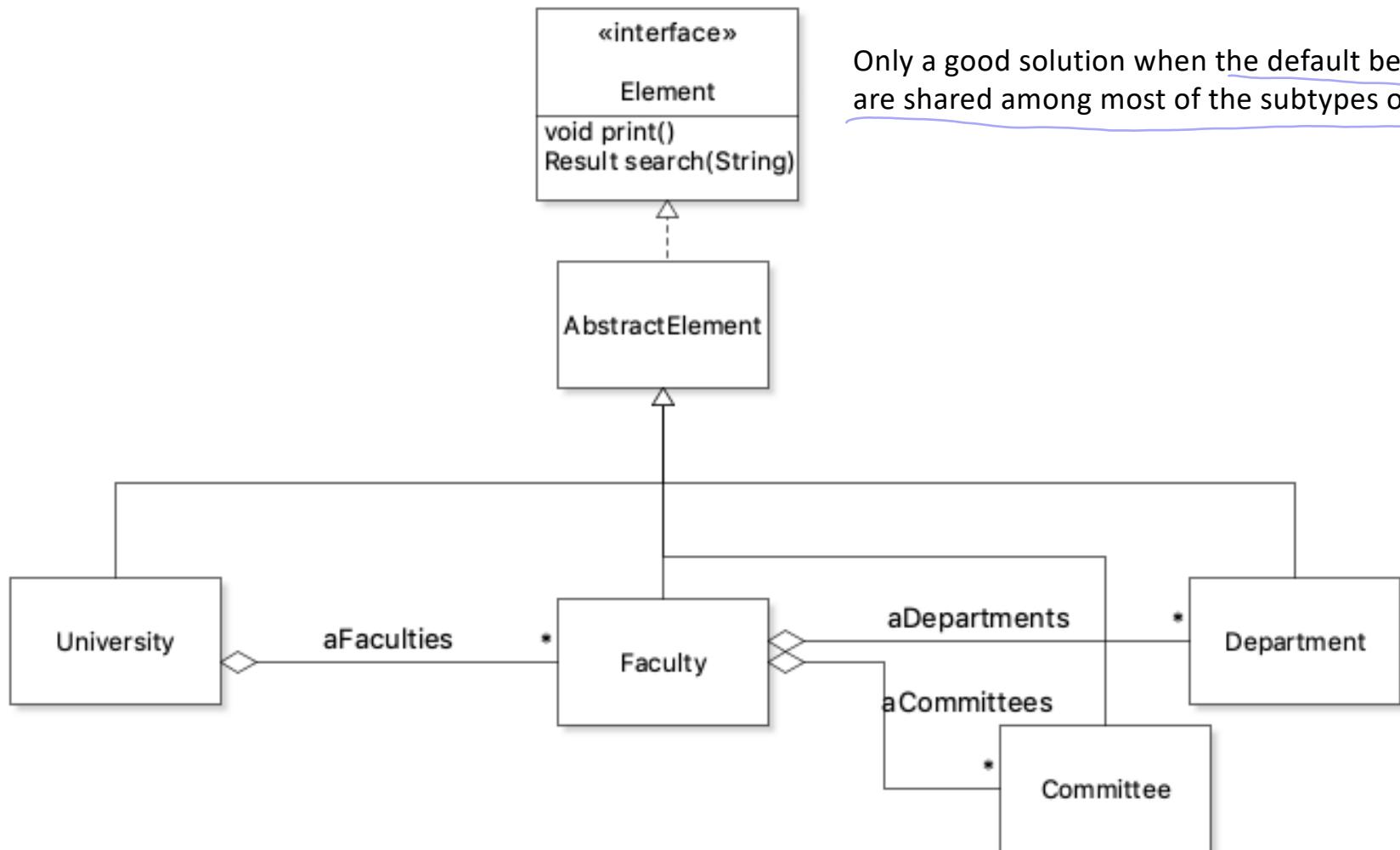
... ...

All requires traverse all the elements in University,  
and process them (potentially differently)

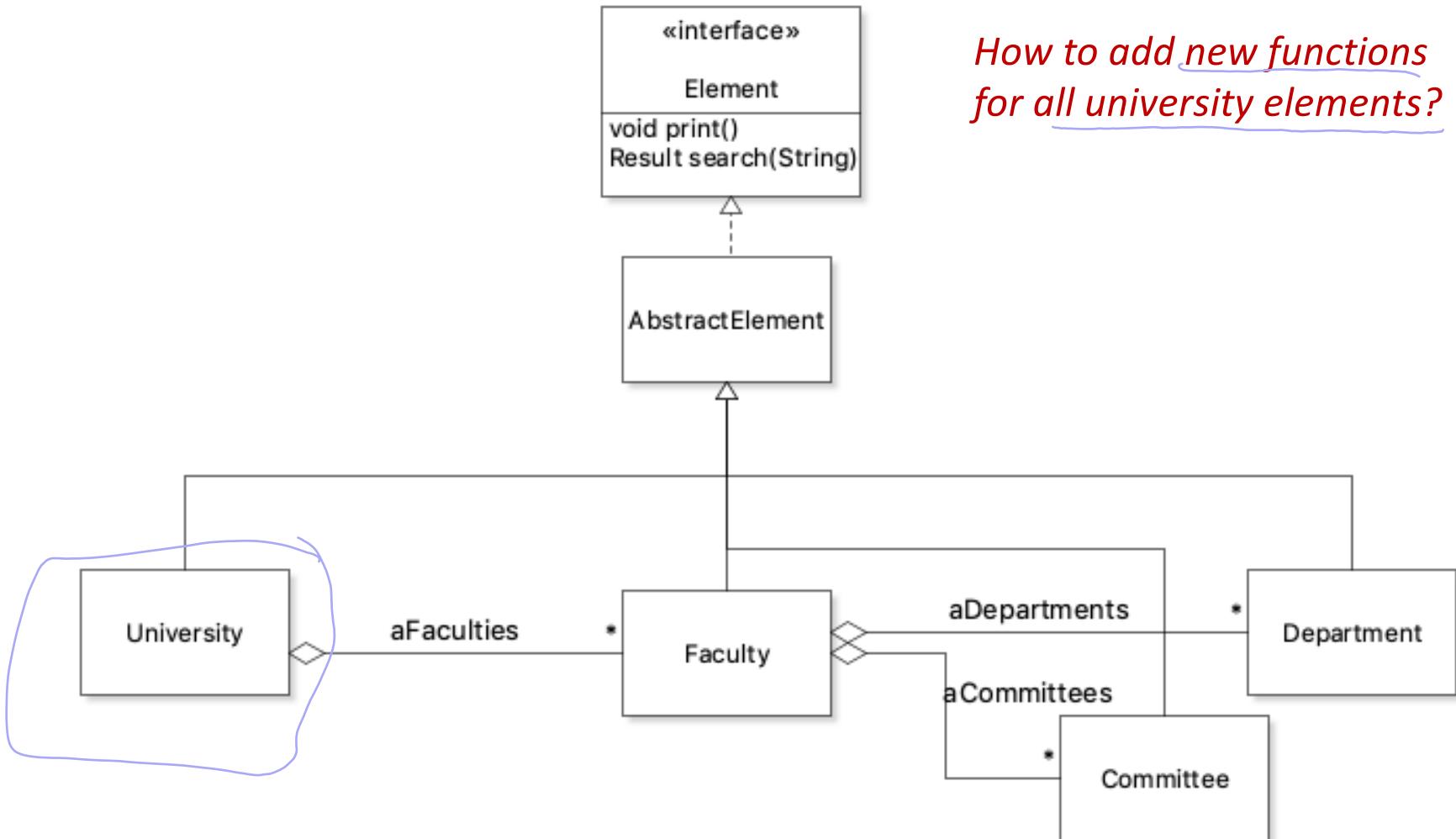


want to  
print,  
search  
in every element



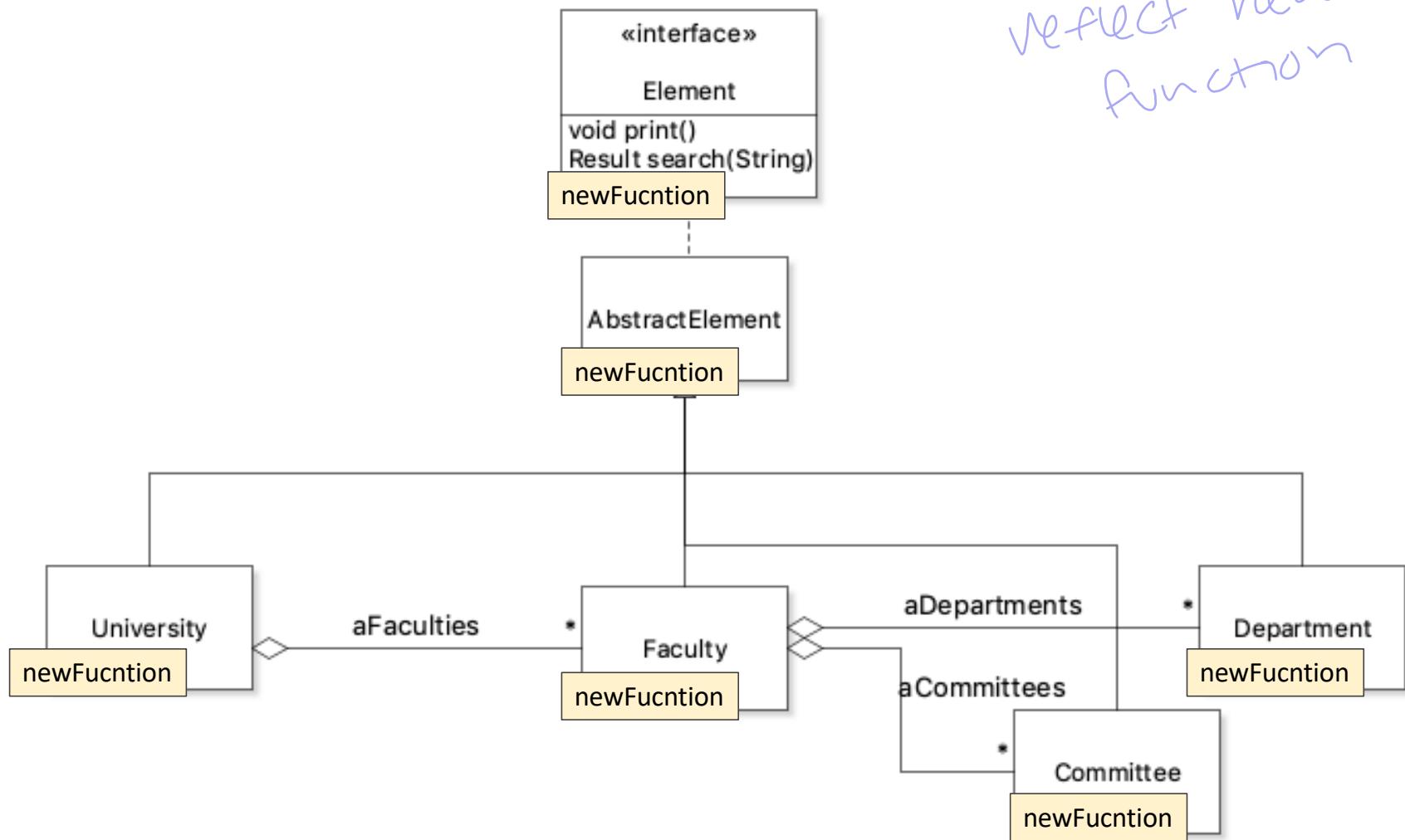


Only a good solution when the default behaviors are shared among most of the subtypes of element.



*How to add new functions for all university elements?*

Change every element to reflect new function

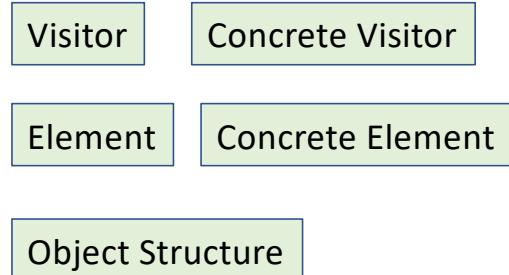


# Visitor

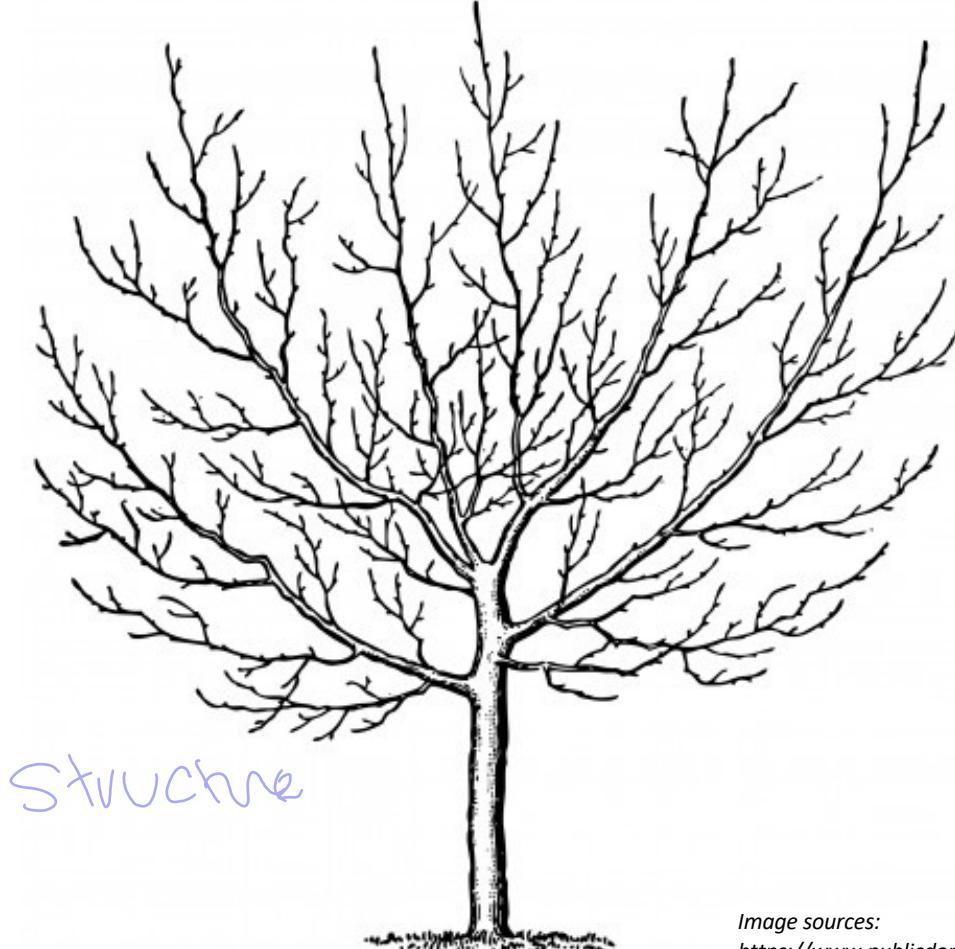
- Intent:

*Represent an operation to be performed on the elements of an object structure.  
Visitor lets you define a new operation without changing the classes of the elements  
on which it operates.*

- Participants:



don't have  
to change  
every element



Structure



can add new visitors  
easily

Image sources:

<https://www.publicdomainpictures.net/pictures/100000/nahled/tree-1409250600Al7.jpg>

<https://www.publicdomainpictures.net/pictures/140000/nahled/black-elf.jpg>

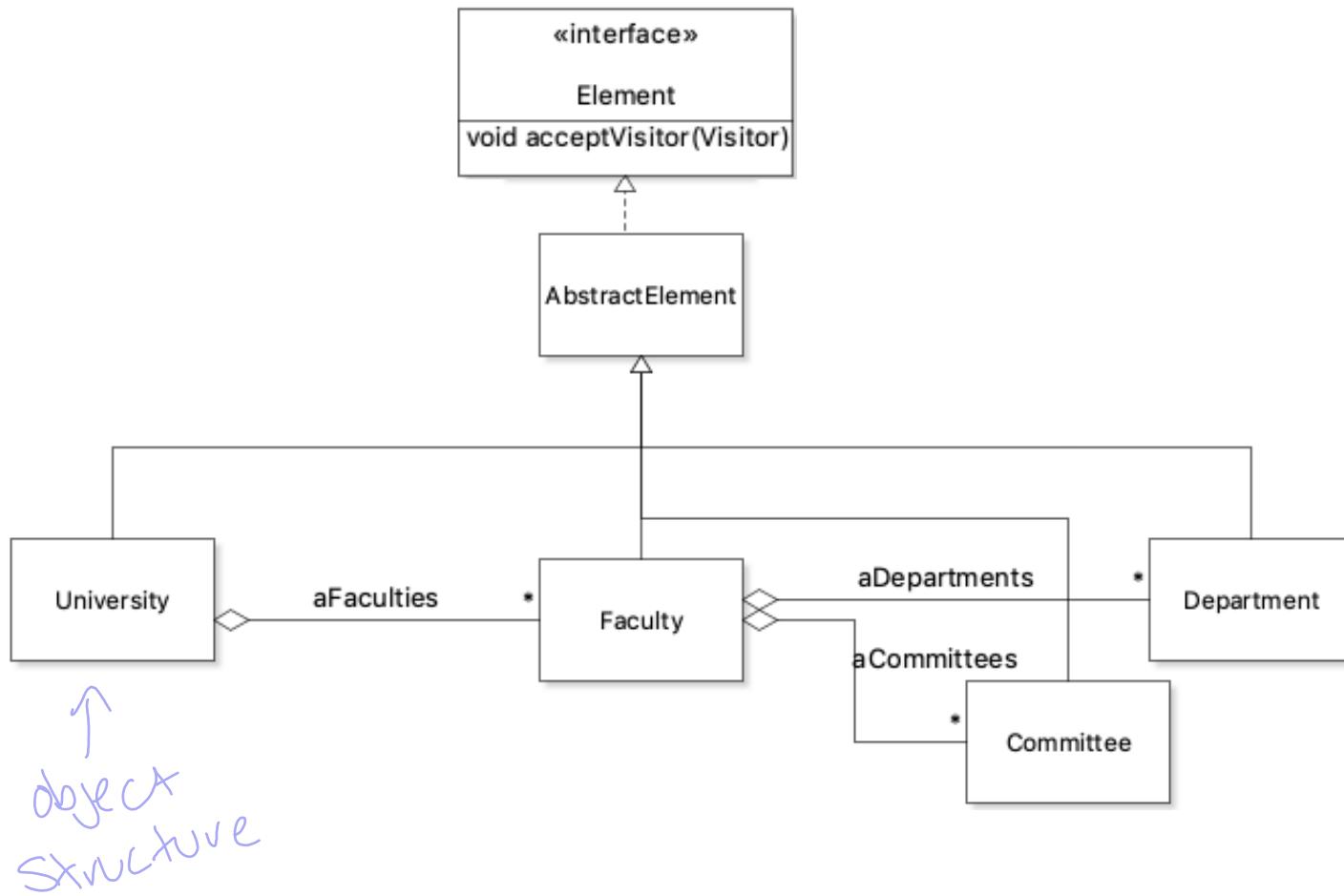
<https://www.publicdomainpictures.net/pictures/190000/nahled/squirrel-silhouette-1469799208bea.jpg>

```

@Override
public void accept(Visitor pVisitor)
{
    // Use callback functions from pVisitor
}

```

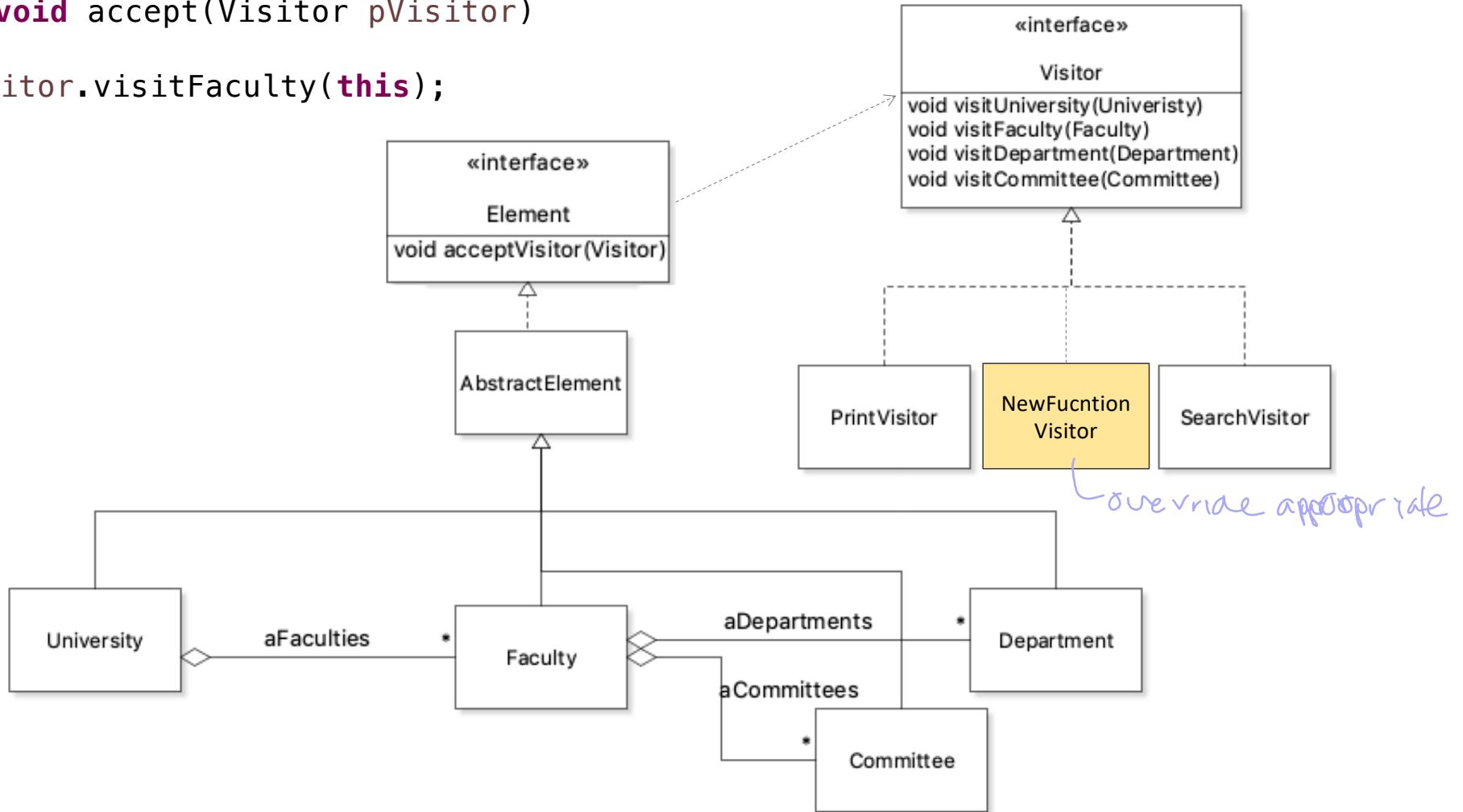
*How to pass around functions?*

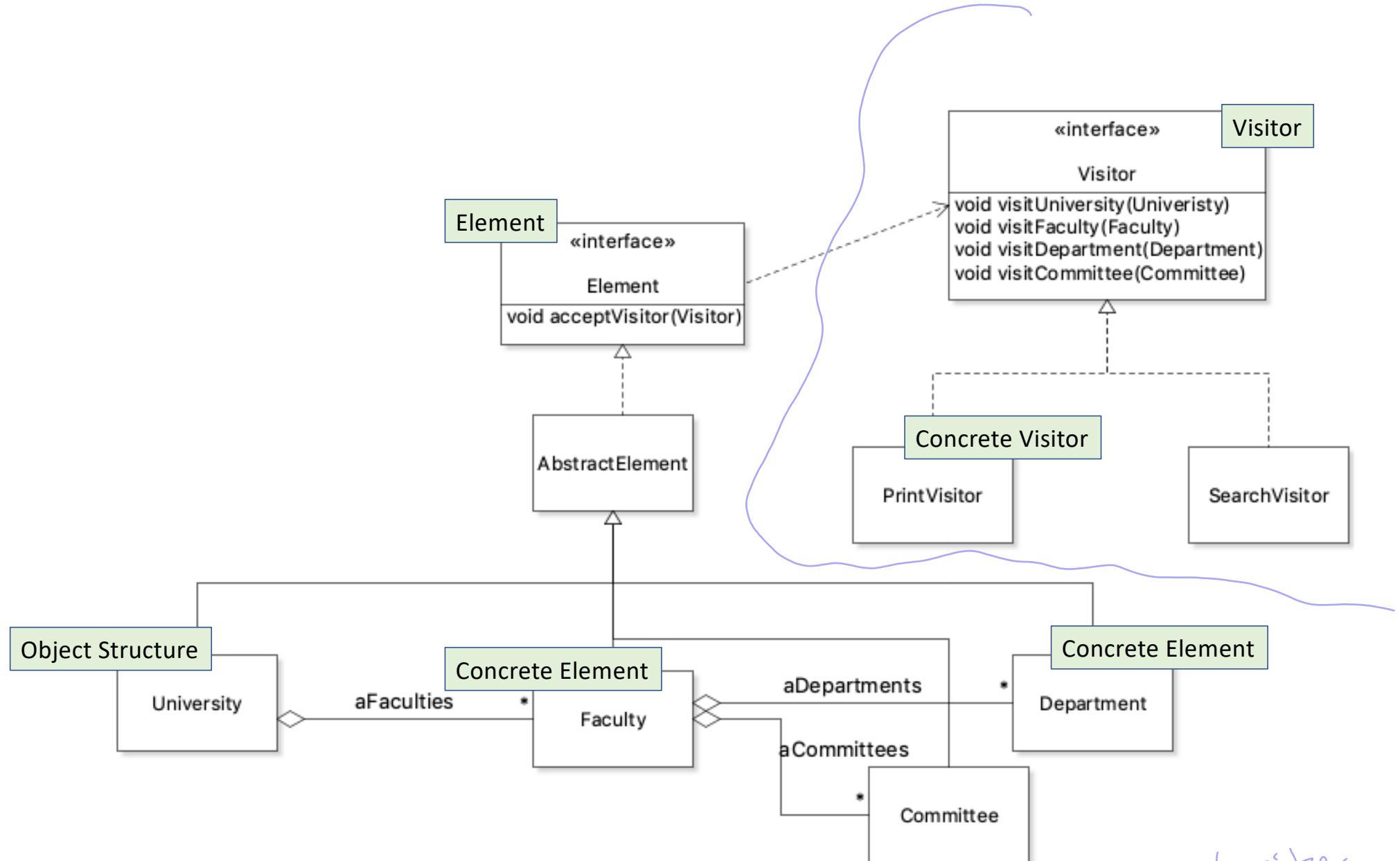


```

@Override
public void accept(Visitor pVisitor)
{
    pVisitor.visitFaculty(this);
}

```





add a function/visitor  
in one class instead of  
all

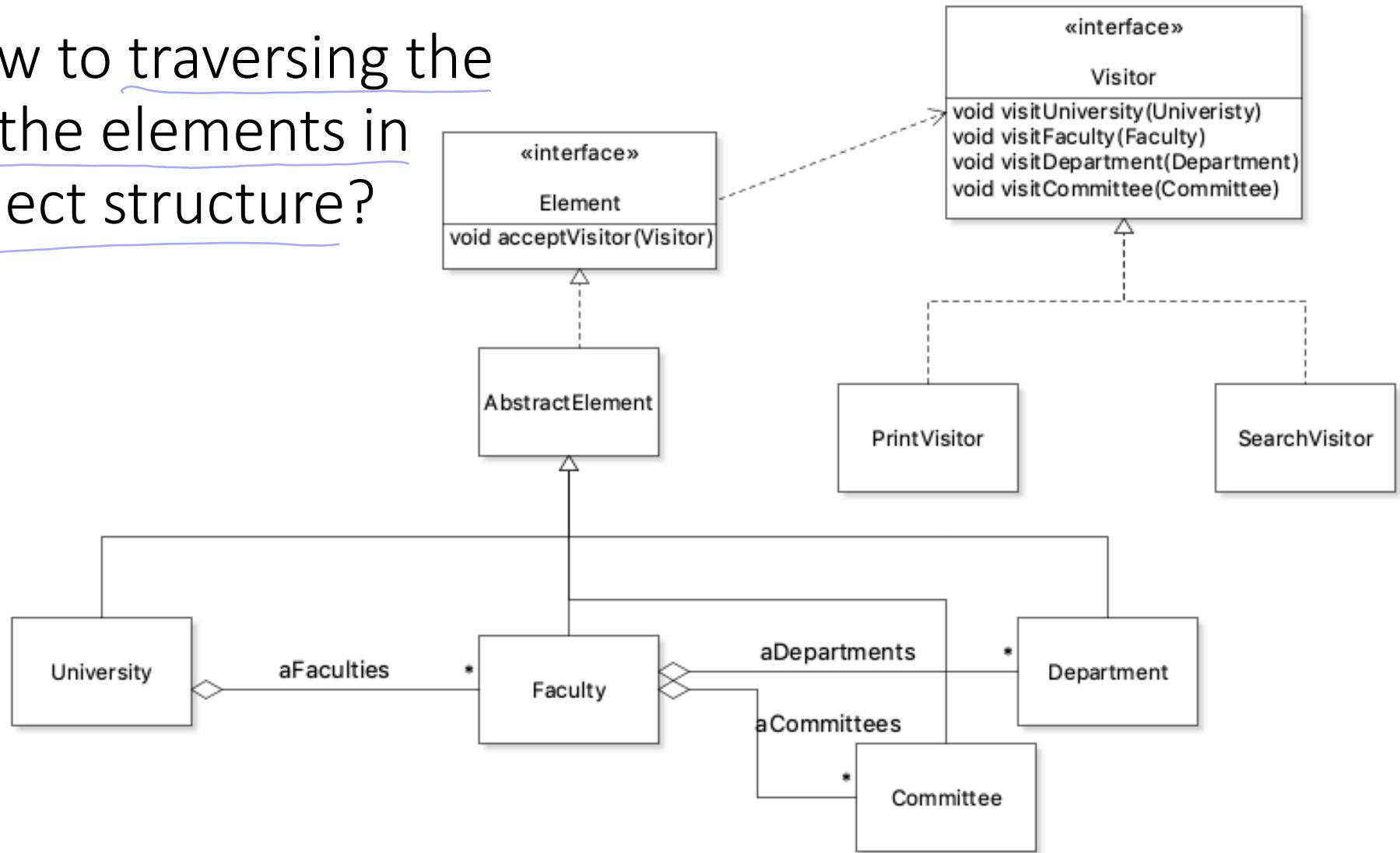
Please think and discuss the consequences when applying Visitor pattern.

Activity 1:

adding a new element  
to object structure  
requires changing every  
visitor

best to use when object structure is stable

How to traversing the  
all the elements in  
object structure?



## Method1:

```
class University extends AbstractElement
{
    private final List<Faculty> aFaculties = new ArrayList<Faculty>();
    public University(String pName) { super(pName); }
    public void addFaculty(Faculty pFaculty) { aFaculties.add(pFaculty); }

    @Override
    public void accept(Visitor pVisitor)
    {
        pVisitor.visitUniversity(this);

        for(Faculty f : aFaculties)
        {
            f.accept(pVisitor);
        }
    }
}
```

← Traverse in the aggregate object structure.

↳ traverse faculties

## Method1:

```
public class PrintVisitor implements Visitor
{
    @Override
    public void visitUniversity(University pUniversity)
    {
        // Printing operation for University
    }

    @Override
    public void visitFaculty(Faculty pFaculty)
    {
        // Printing operation for Faculty
    }
}
```

## Method2:

```
class University extends AbstractElement
{
    private final List<Faculty> aFaculties = new ArrayList<Faculty>();
    public University(String pName) { super(pName); }
    public void addFaculty(Faculty pFaculty) { aFaculties.add(pFaculty); }
    public Iterator<Faculty> getFaculties() { return aFaculties.iterator(); }

    @Override
    public void accept(Visitor pVisitor)
    {
        pVisitor.visitUniversity(this);
    }
}
```

↑ But provide a traversal mechanism to its elements

← Not in the object structure.

move traversal into

## Method2:

```
public class PrintVisitor implements Visitor
{
    @Override
    public void visitUniversity(University pUniversity)
    {
        // Printing operation for University
        for( Iterator<Faculty> i = pUniversity.getFaculties();
            i.hasNext(); )
        {
            i.next().accept(this);
        }
    }
}
```

Traverse in the visitor→

iterator

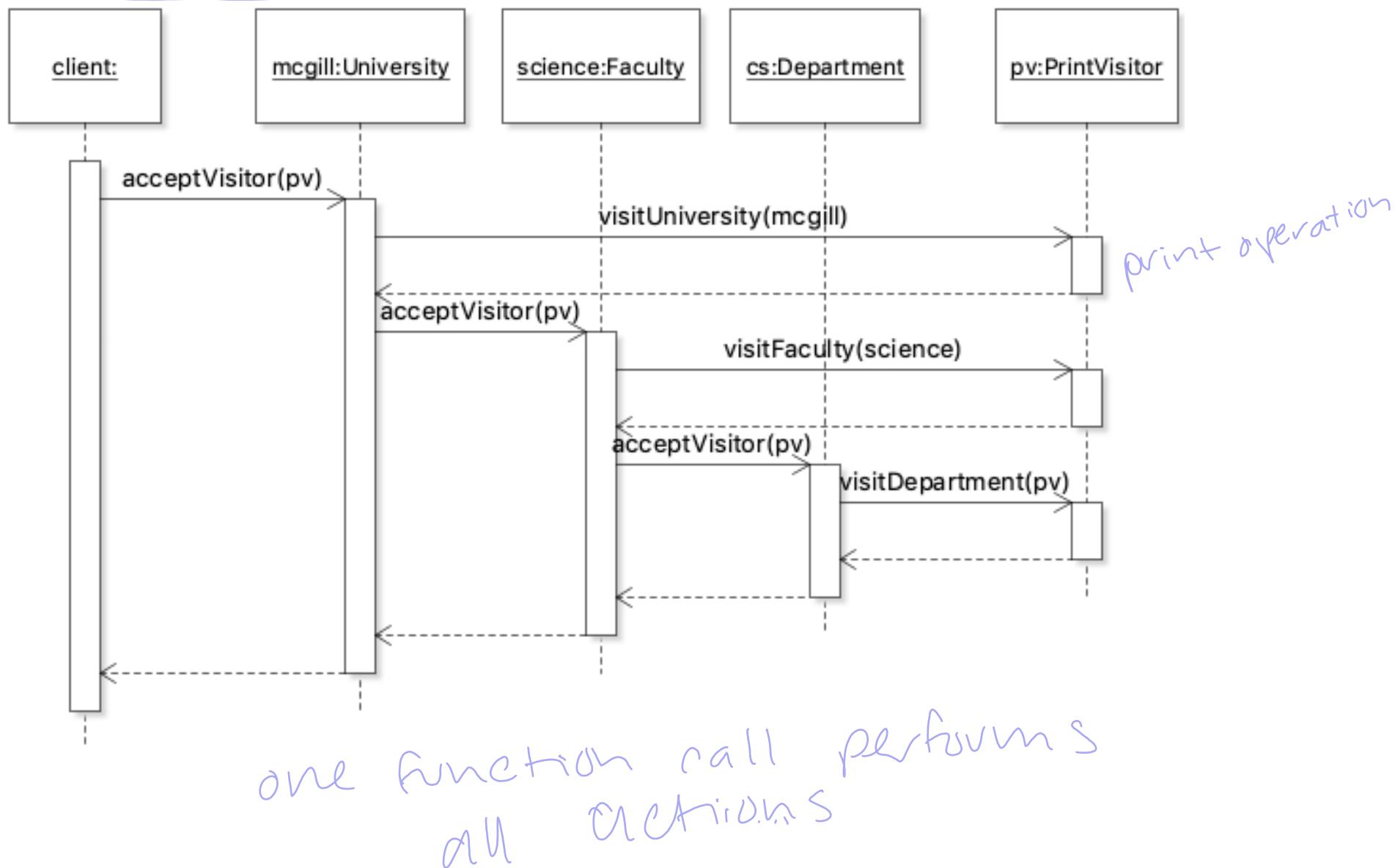
## Activity2: Sequence Diagram for Visitor pattern

```
University mcGill = new University("McGill");
Faculty science = new Faculty("Science");
Department cs = new Department("Computer Science");

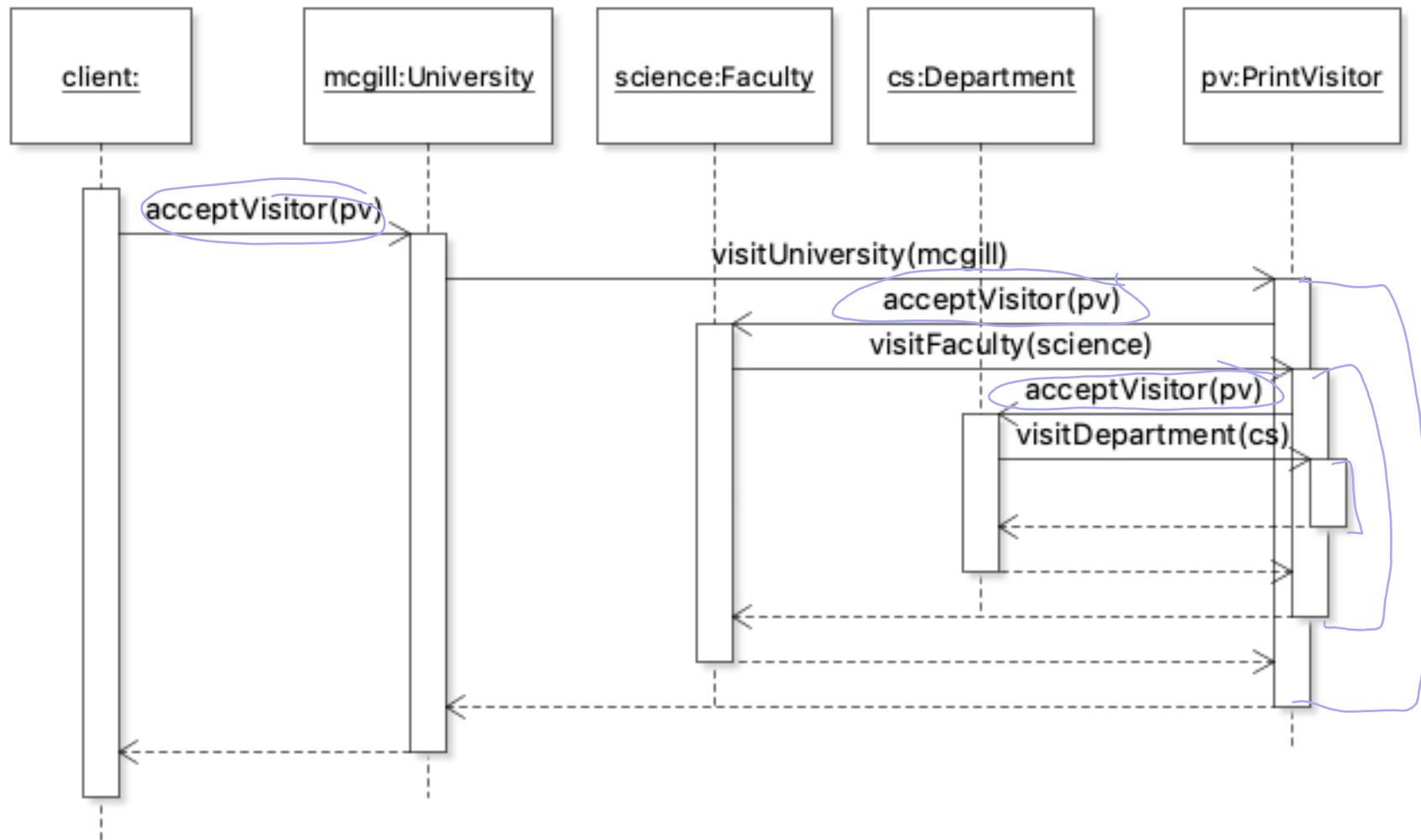
mcGill.addFaculty(science);
science.addDepartment(cs);

Visitor pv = new PrintVisitor();
mcGill.acceptVisitor(pv);
```

## Traverse in the object structure



Sequence diagram for traversal in Visitor?



Demo on McGill Visitor

on github ↴

# What is design pattern again?



- A standard solution to a common programming problem
  - A technique for making code more flexible
  - Shorthand for describing program design
  - Vocabulary for communication & documentation
- 

	<b>Creational</b>	<b>Structural</b>	<b>Behavioral</b>
<b>Class</b>	Factory Method	Adapter (class)	Intregerter Template Method ✓
<b>Object</b>	Abstract Factory	Adapter (class)	Chain of Responsibility
	Builder	Bridge	Command ✓
	Prototype ✓	Composite ✓	Iterator ✓
	Singleton ✓	Decorator ✓	Mediator
		Flyweight ✓	Memento
		Façade	Observer ✓
		Proxy	State
			Strategy ✓
			Visitor ✓

# Creational Patterns

prototype  
singleton

- Creational Patterns control object creation
  - encapsulate knowledge about which concrete classes the system uses
  - hide how instances of concrete classes are created and combined.

Why not using the constructors?

Cannot return a subtype

Always create a new object

	<b>Creational</b>	<b>Structural</b>	<b>Behavioral</b>
<b>Class</b>	Factory Method	Adapter (class)	Interpreter Template Method ✓
<b>Object</b>	Abstract Factory	Adapter (object)	Chain of Responsibility
	Builder	Bridge	Command ✓
	Prototype ✓	Composite ✓	Iterator ✓
	Singleton ✓	Decorator ✓	Mediator
		Flyweight ✓	Memento
		Façade	Observer ✓
		Proxy	State
			Strategy ✓
			Visitor ✓

# Structural Patterns

composite  
decorator  
flyweight

- Concerned with how classes and objects are composed to form larger structures.
- Patterns are similar on structure:
  - single and multiple inheritance for class-based patterns
  - object composition for object patterns.
- Patterns have distinct intention *from each other*

	<b>Creational</b>	<b>Structural</b>	<b>Behavioral</b>
<b>Class</b>	Factory Method	Adapter (class)	Interpreter Template Method ✓
<b>Object</b>	Abstract Factory	Adapter (object)	Chain of Responsibility
	Builder	Bridge	Command ✓
	Prototype ✓	Composite ✓	Iterator ✓
	Singleton ✓	Decorator ✓	Mediator
		Flyweight ✓	Memento
		Façade	Observer ✓
		Proxy	State
			Strategy ✓
			Visitor ✓

## Behavioral Patterns

- Concerned with algorithms and the assignment of responsibilities between objects.
- Describe not just patterns of objects or classes but also the patterns of communication between them.

# Encapsulating Variation

- When an aspect of a program changes frequently, the behavioral patterns define an object that encapsulate that aspect:
  - A Strategy object encapsulates content of operation;
  - An iterator object encapsulates traversal;
  - A command object encapsulates performance; of operations
  - A visitor object encapsulates new operations;

# Passing around object

- Object is returned to the client and will be used at later time.
  - An iterator
  - A command
- Object is passed into the system as argument
  - A strategy
  - A visitor

# Review Objectives of “Inversion of control”

- Be able to Use Callback to achieve decoupling
- Be able to use the Observer design pattern effectively;
- Event Handling in GUI applications
- Understand the concept of an application framework;
- Understand the Model-View-Controller Decomposition;
- Be able to use the Visitor Design Pattern effectively;
- Be able to determine when to used different design patterns effectively.