

A photograph of a red squirrel climbing a tree trunk. The squirrel is facing away from the camera, its reddish-brown fur contrasting with the dark, textured bark of the tree. It is gripping the trunk with its front paws and tail. The background is filled with green foliage and other tree trunks, suggesting a dense forest environment.

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

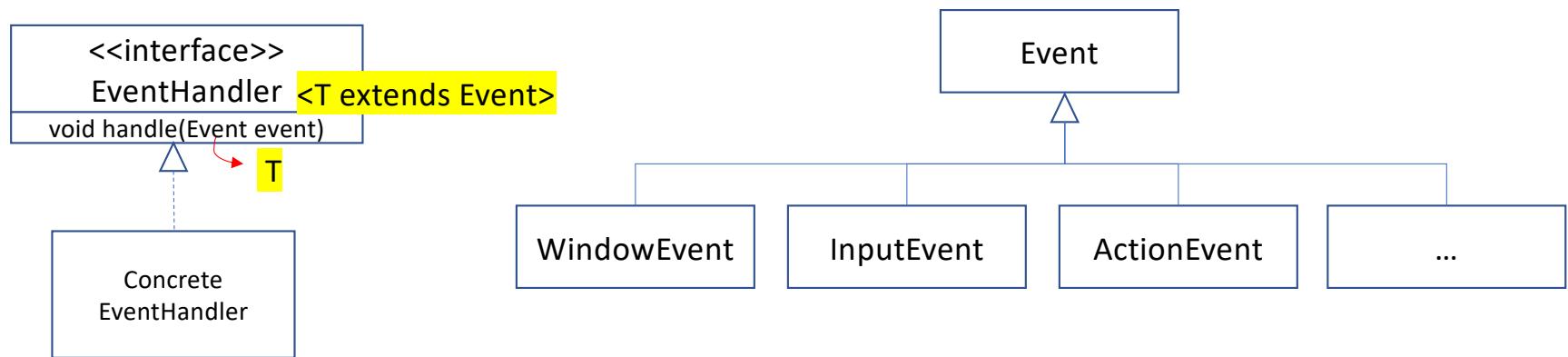
Menu item is selected

Window is closed

Popup window is hidden



How to capture event and act accordingly

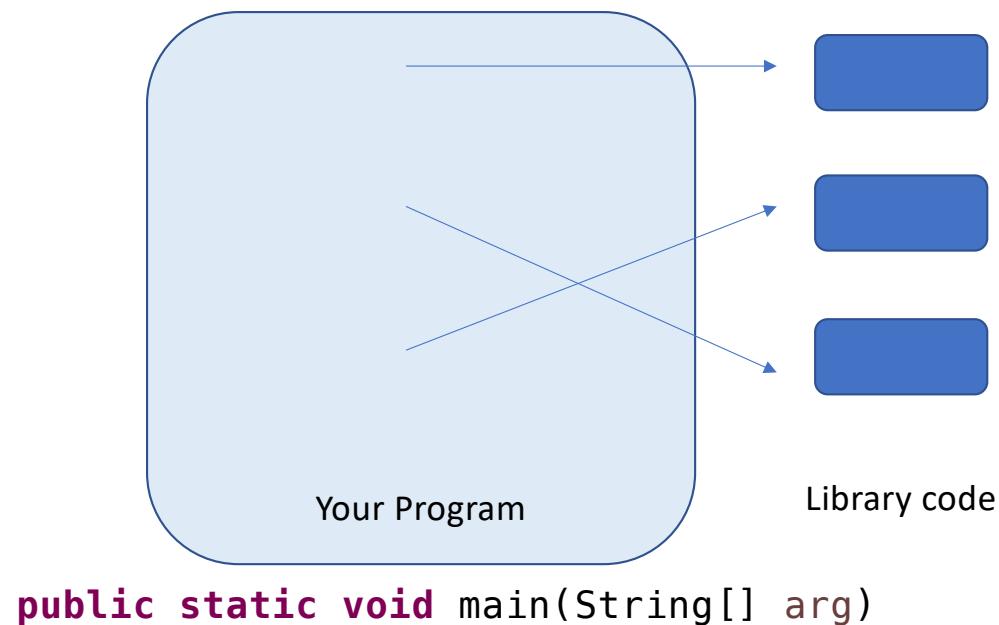


How to capture event and act accordingly

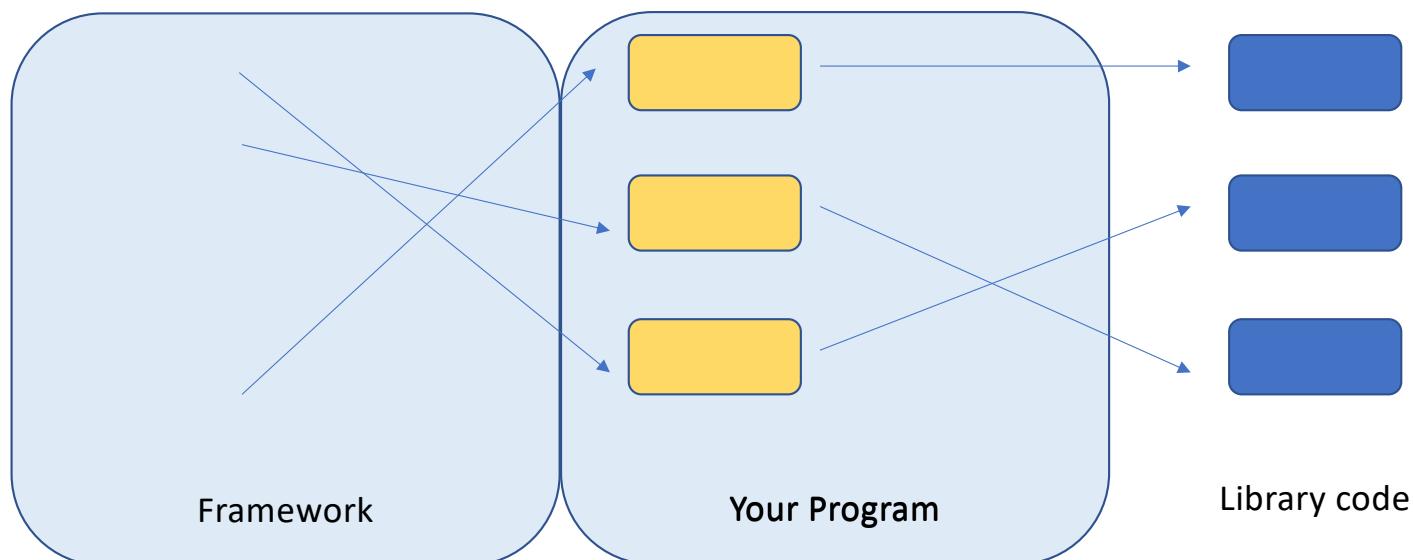
- Instantiate and register the event handler

```
Button btn = new Button();  
anonymous class  
Button  
btn.setOnAction(new EventHandler<ActionEvent>() {  
    public void handle(ActionEvent event) {  
        //Event Handling steps  
    }  
});
```

Library vs Framework



Library vs Framework

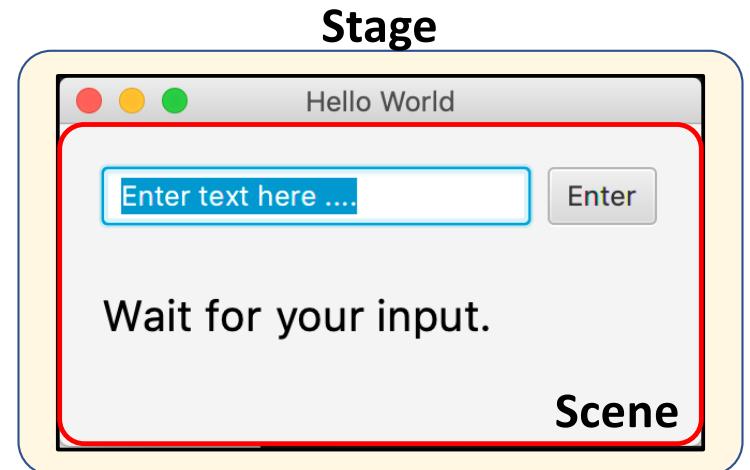


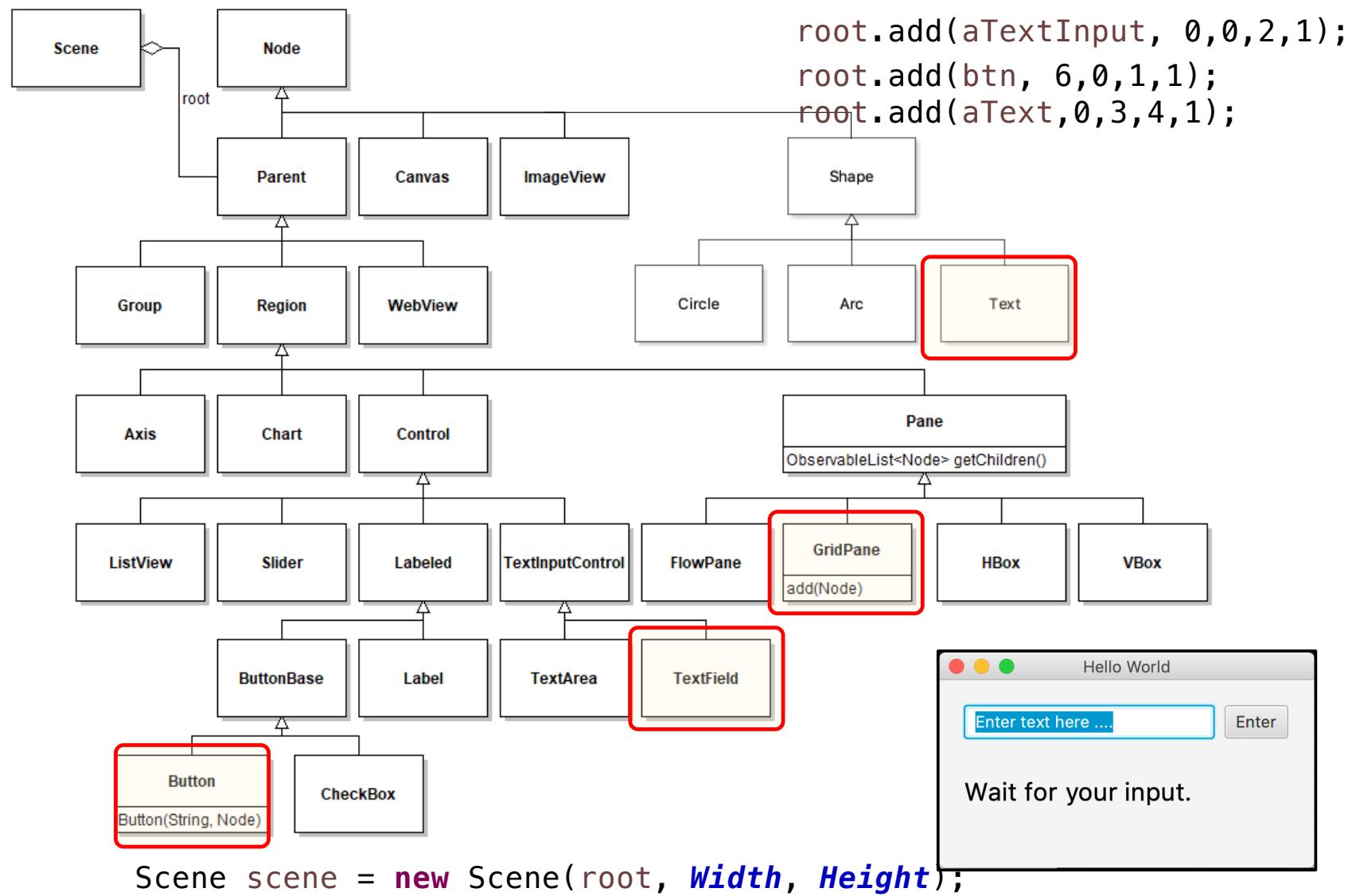
Launch JavaFX framework

```
public class MyApplication extends Application
{
    /**
     * Launches the application.
     * @param pArgs the command line arguments passed to the
     * application.
     */
    public static void main(String[] pArgs)
    {
        launch(pArgs);
    }

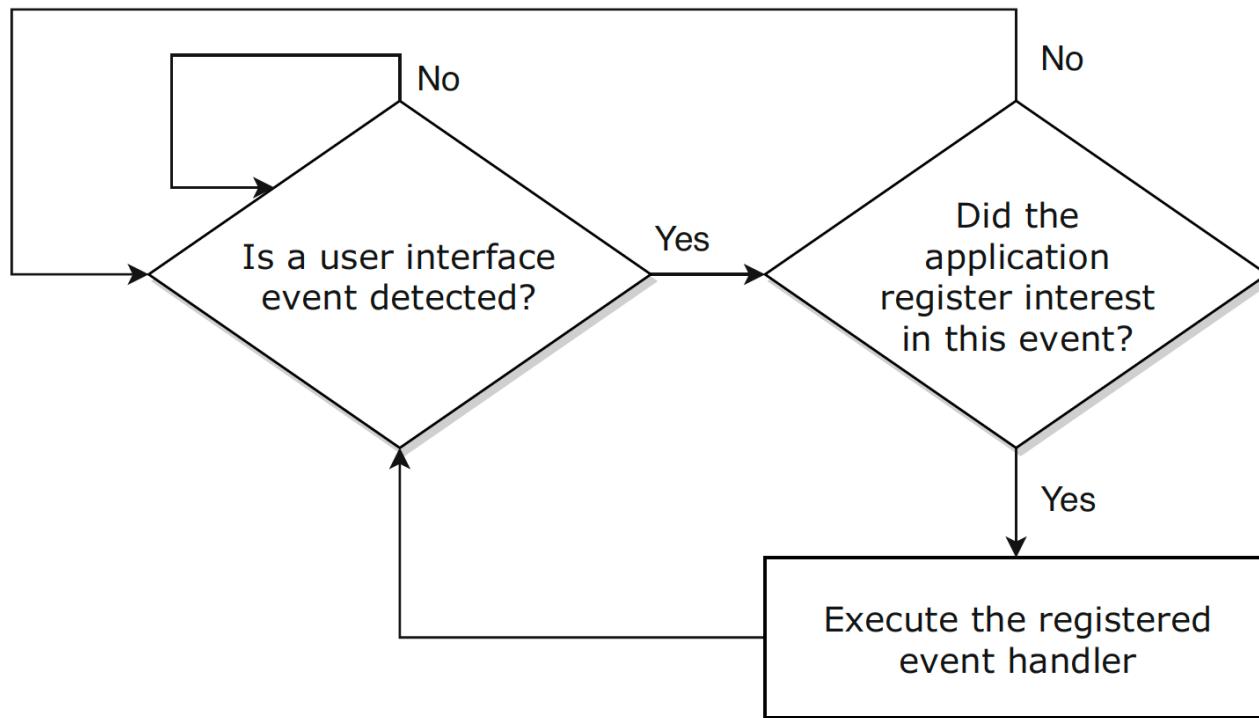
    @Override
    public void start(Stage pPrimaryStage)
    {
        //Setup the stage
        pPrimaryStage.show();
    }
}
```

```
@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);  
    root.add(aText,0,3,4,1);  
  
    Scene scene = new Scene(root, Width, Height);  
    ... //Other setup steps  
  
    primaryStage.setScene(scene);  
  
    pPrimaryStage.show();  
}
```





When does event handling happen?



Text Display Example



```
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()));
```

Recap: Objective

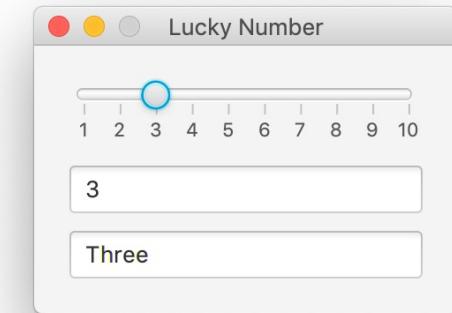
- Be able to Use Callback to achieve decoupling
- Be able to use the Observer design pattern effectively;
- Event Handling in GUI applications
- Understand the concept of an application framework;
- Understand the Model-View-Controller Decomposition;
- Be able to use the Visitor Design Pattern effectively;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

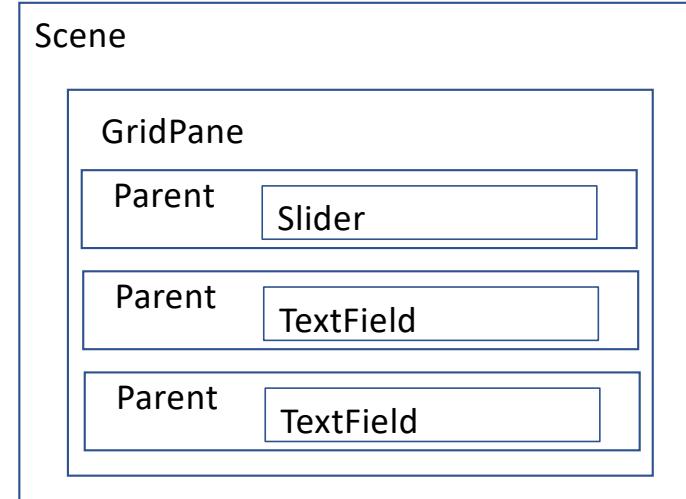
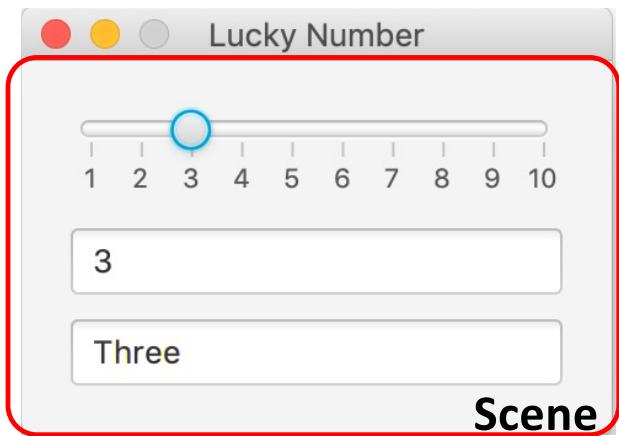
Lucky Number Example

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

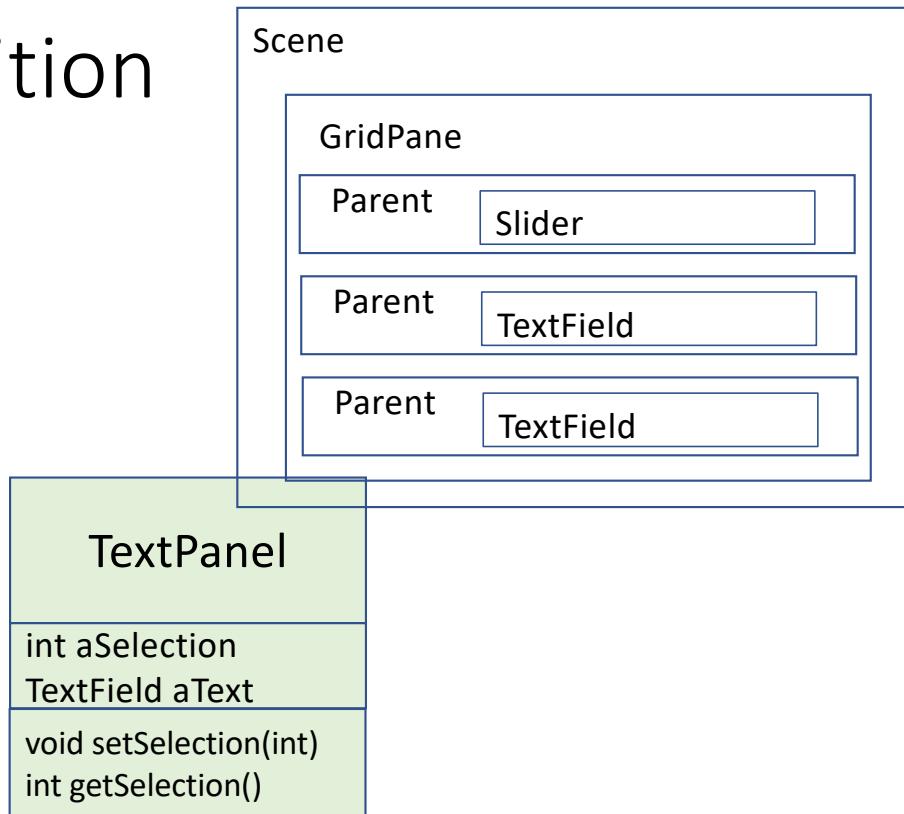
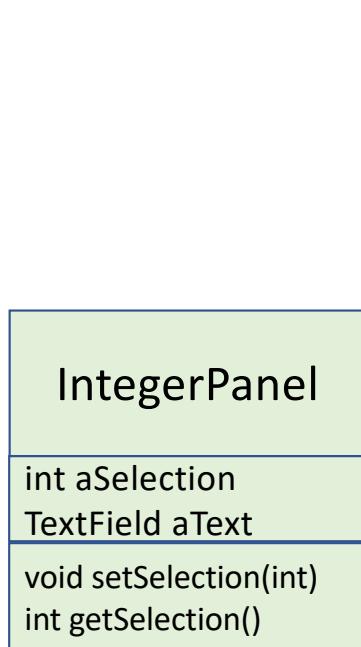
The selection should be performed through either typing it, writing it out in the corresponding fields, or selecting it from a slider.

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

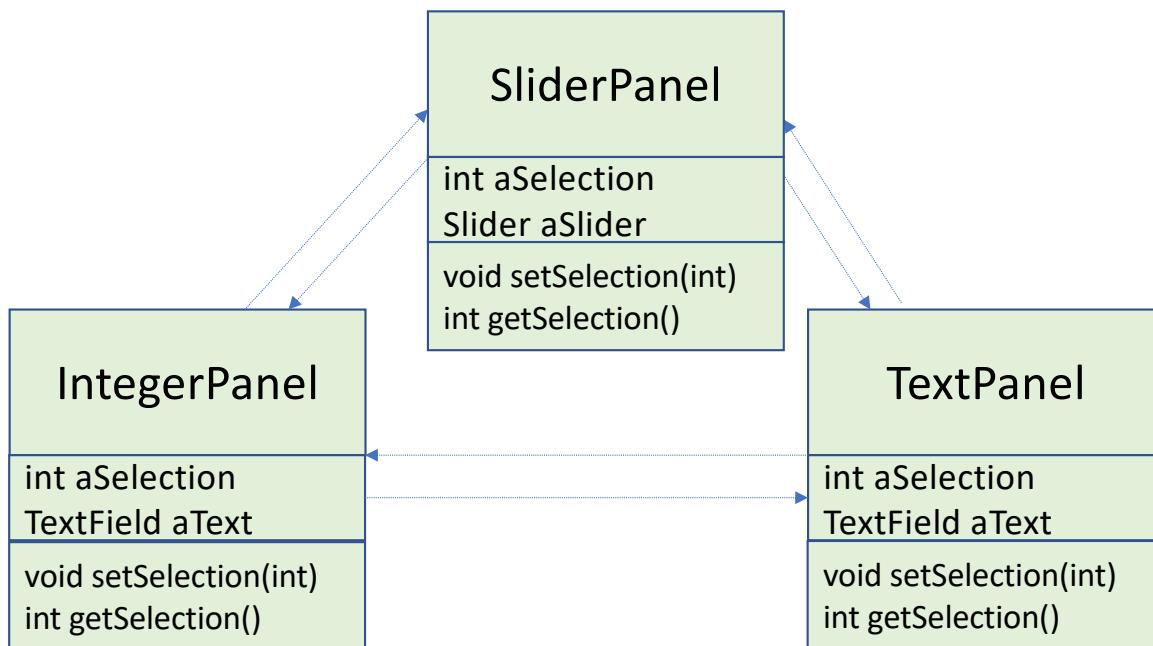




Problem Decomposition



Problem Decomposition



High Coupling

Components are inter-dependent

Low Extensibility

hard to add/remove selection mechanism

MVC Decomposition

Model – View – Controller

Design pattern

Architectural pattern

Guideline to separate concerns

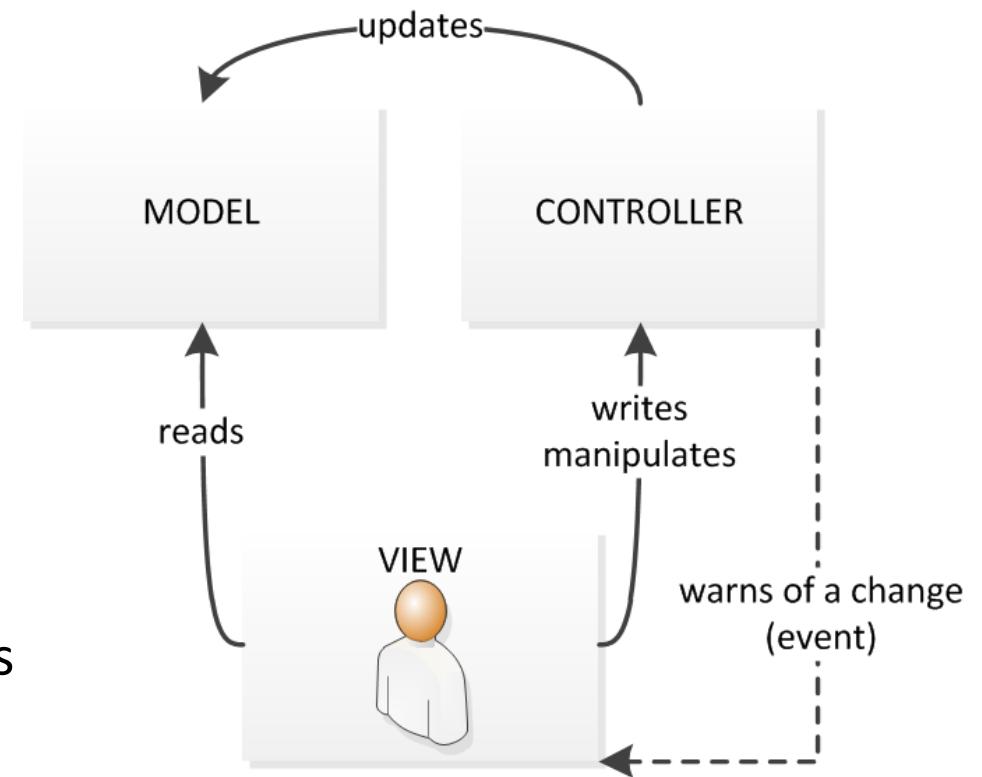
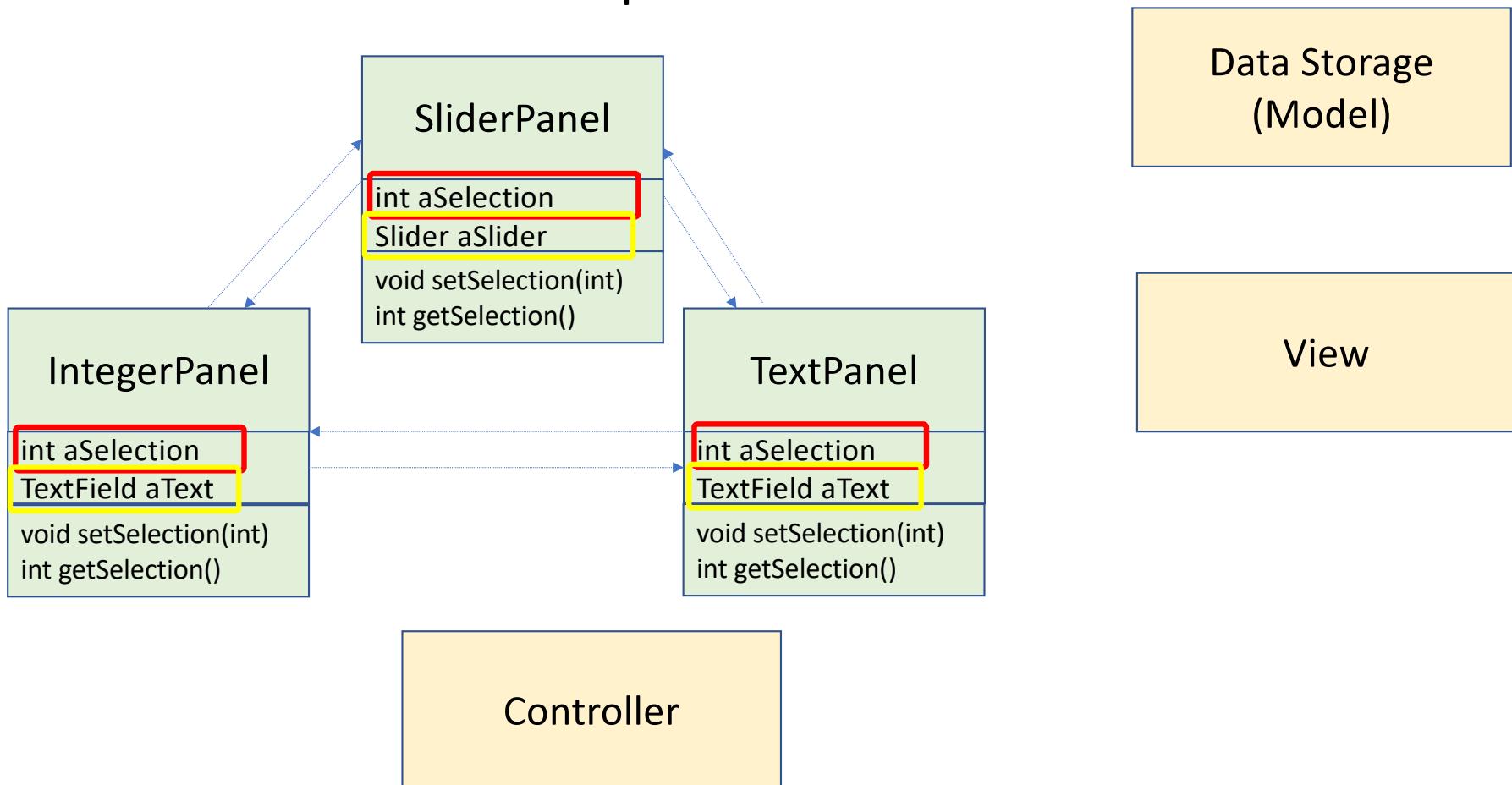
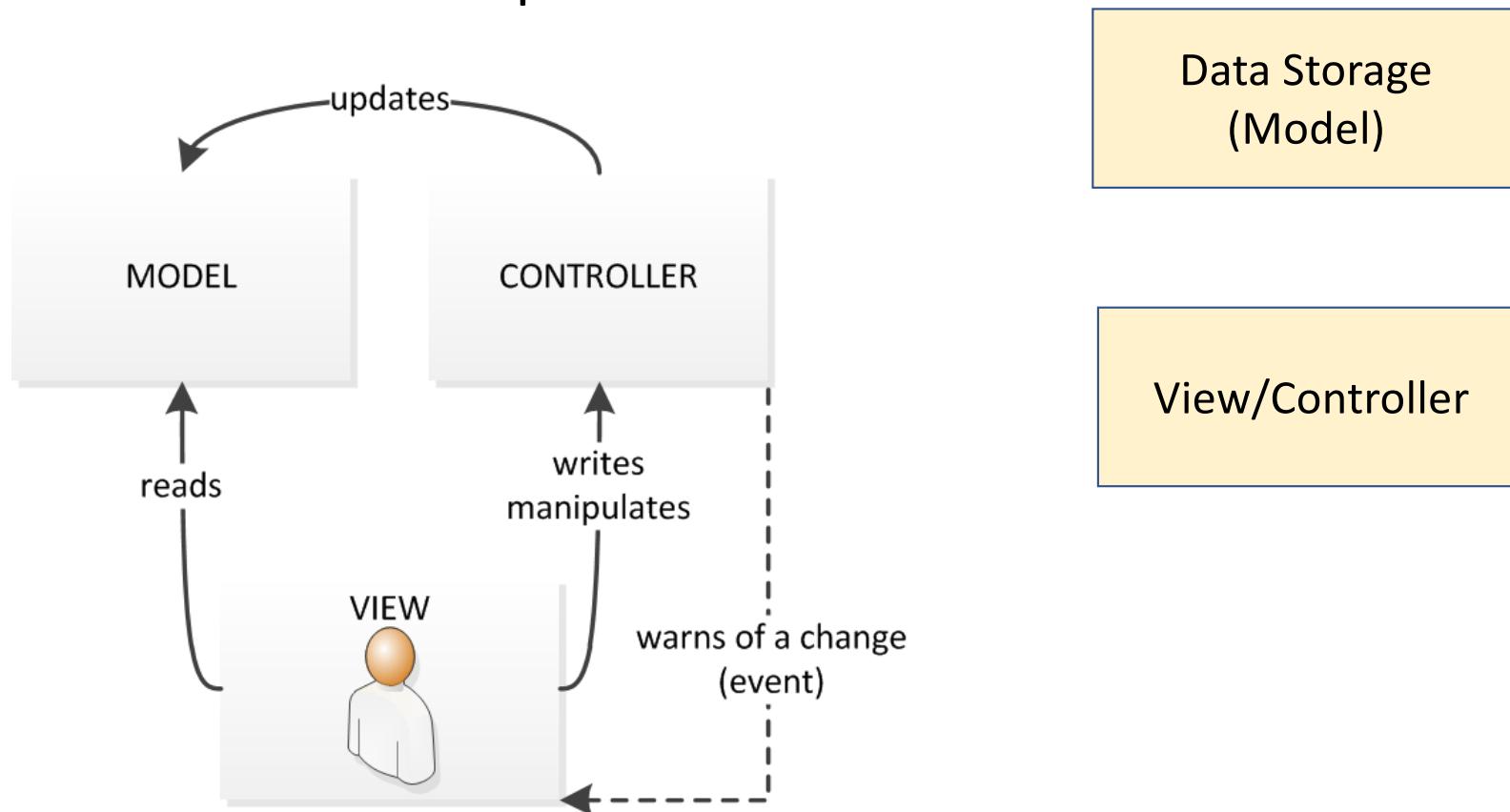


Image Source: <https://upload.wikimedia.org/wikipedia/commons/6/63/ModeleMVC.png>

Problem Decomposition



Problem Decomposition

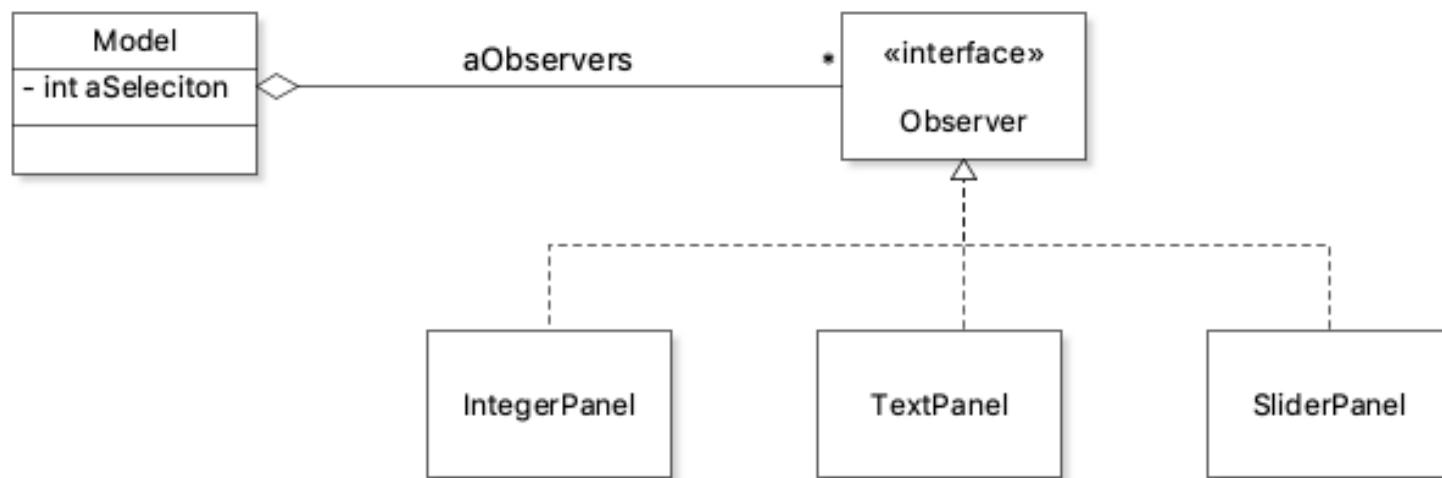


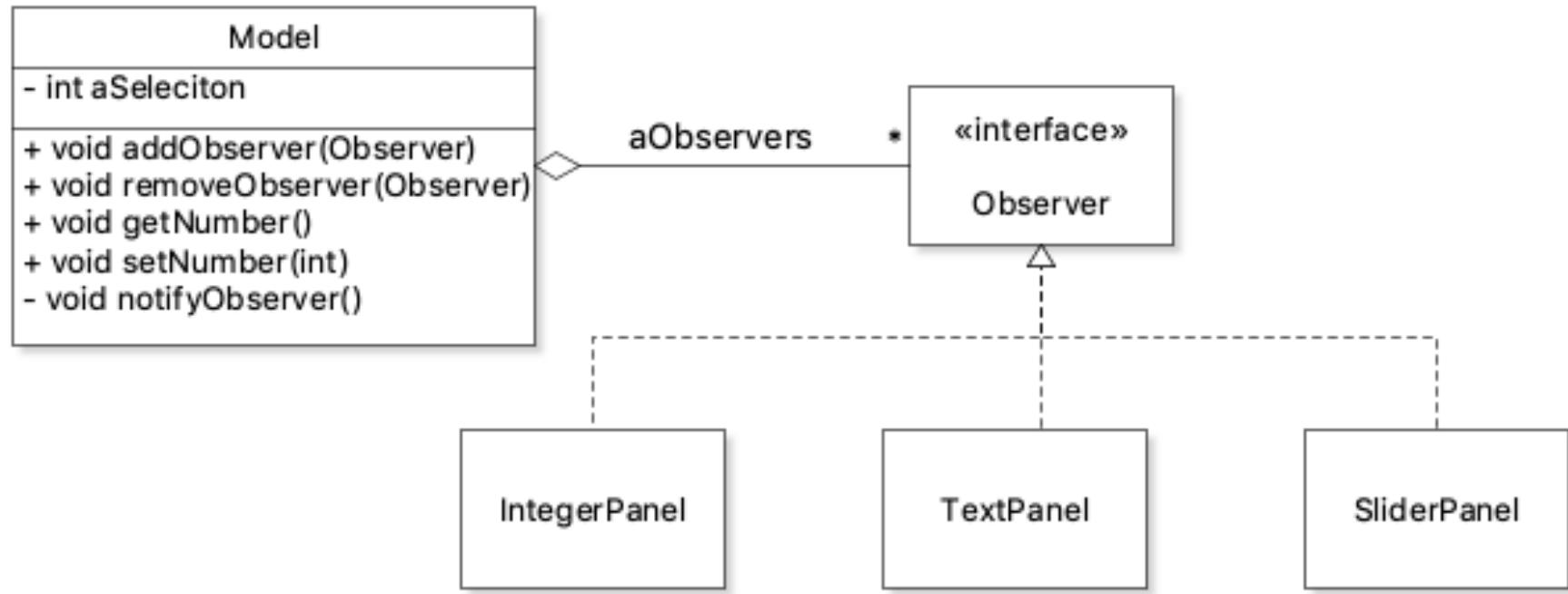
Activity

- Improve the design using Observer Pattern and MVC decomposition.

Activity: Applying Observer in MVC

- What methods should be included in Model?



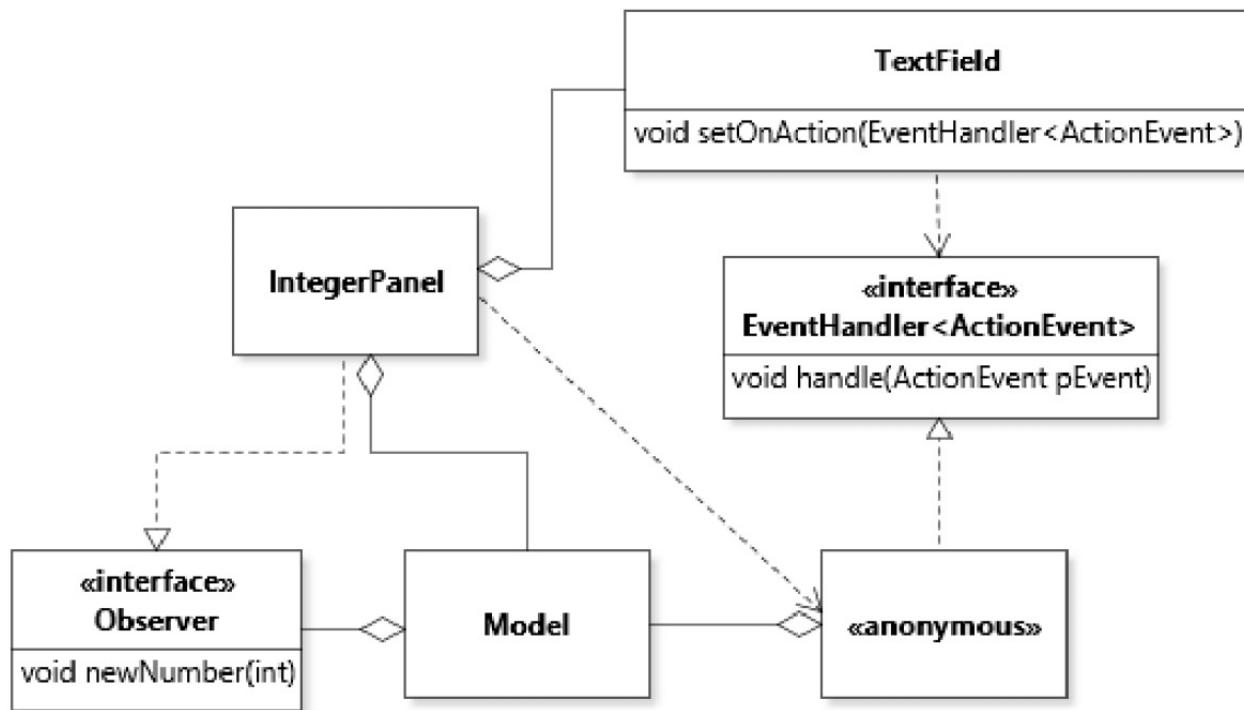


```
/**  
 * Abstract observer role for the model.  
 */  
interface Observer  
{  
    void newNumber(int pNumber);  
}
```

```
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);  
        }  
    });  
}
```



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

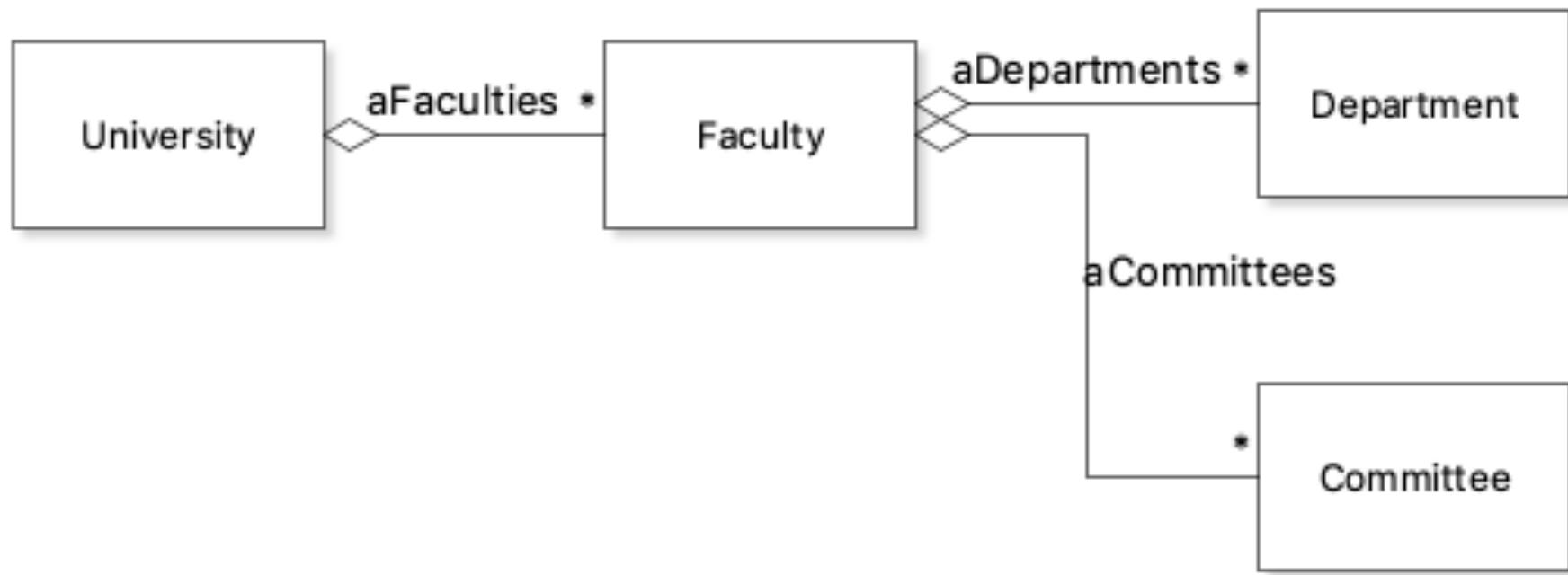
Physics

...

Academic Committee

Scholarship Committee

Students Committee



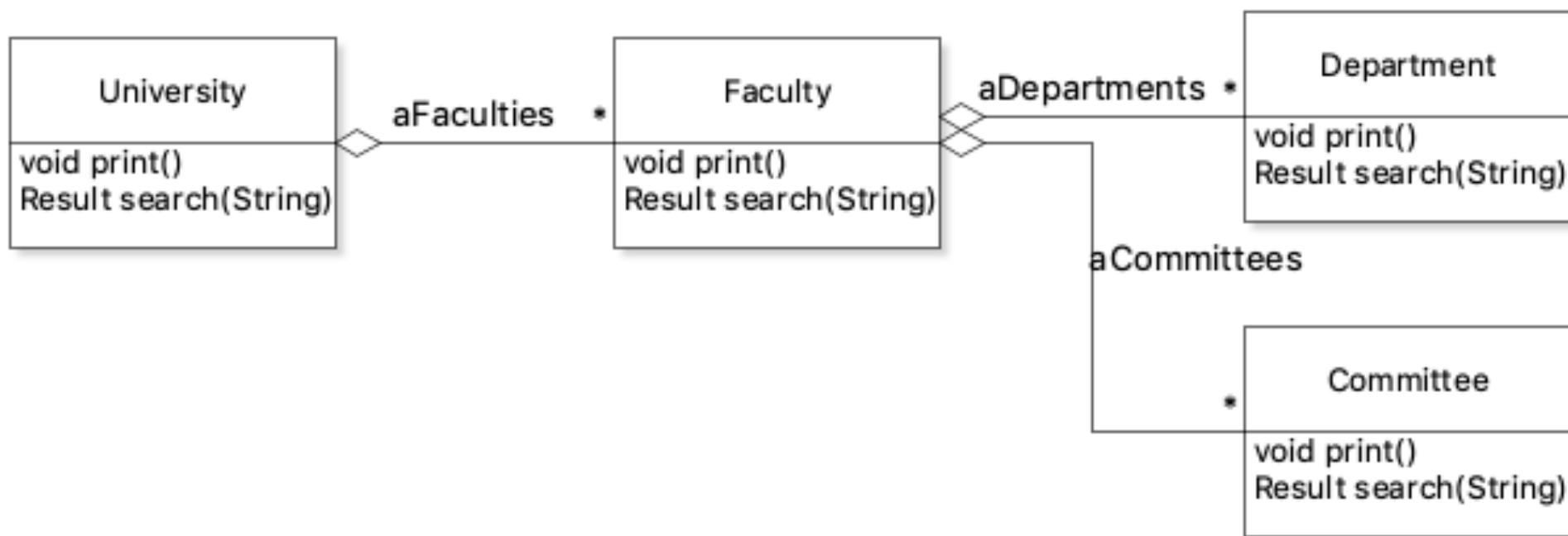
Add functionalities to aggregate

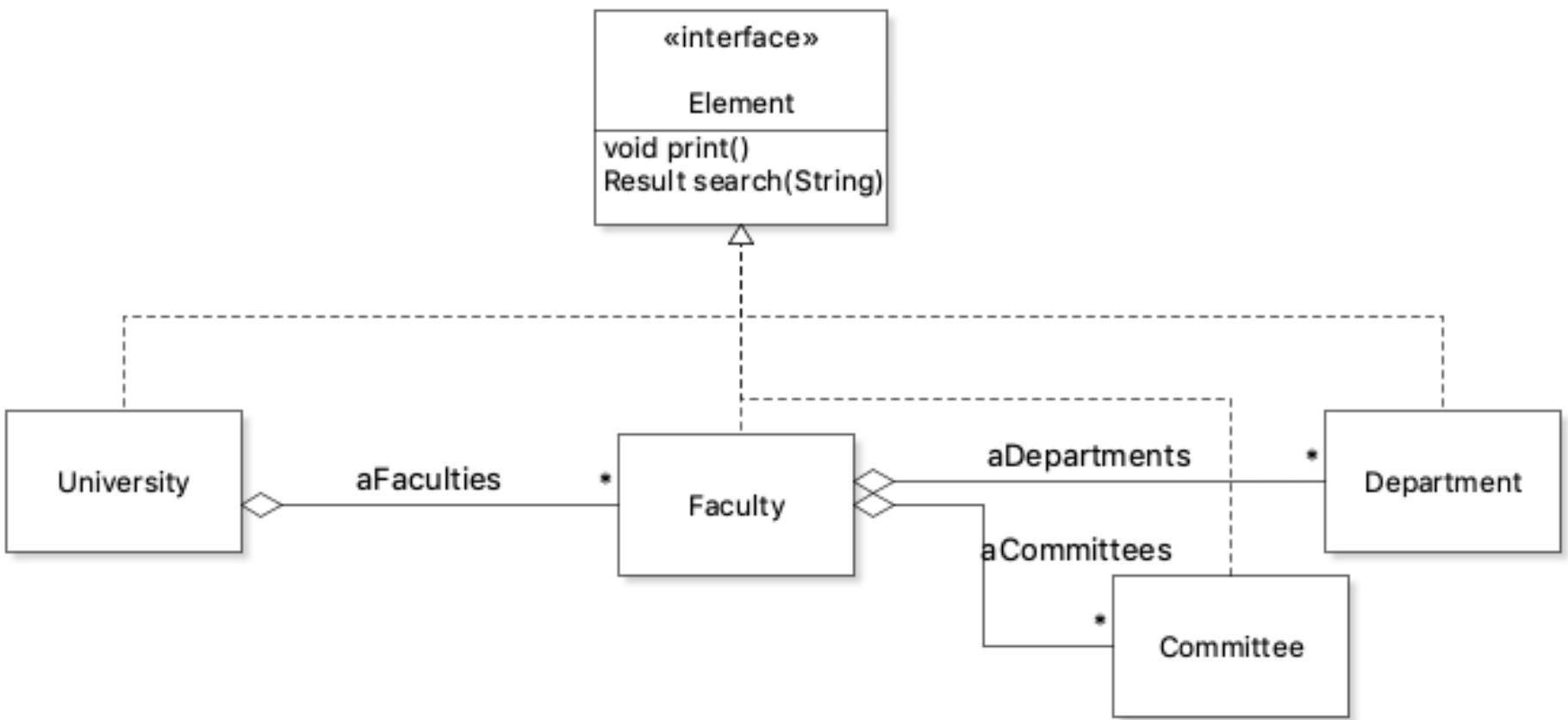
Print annual report for every organizations in university

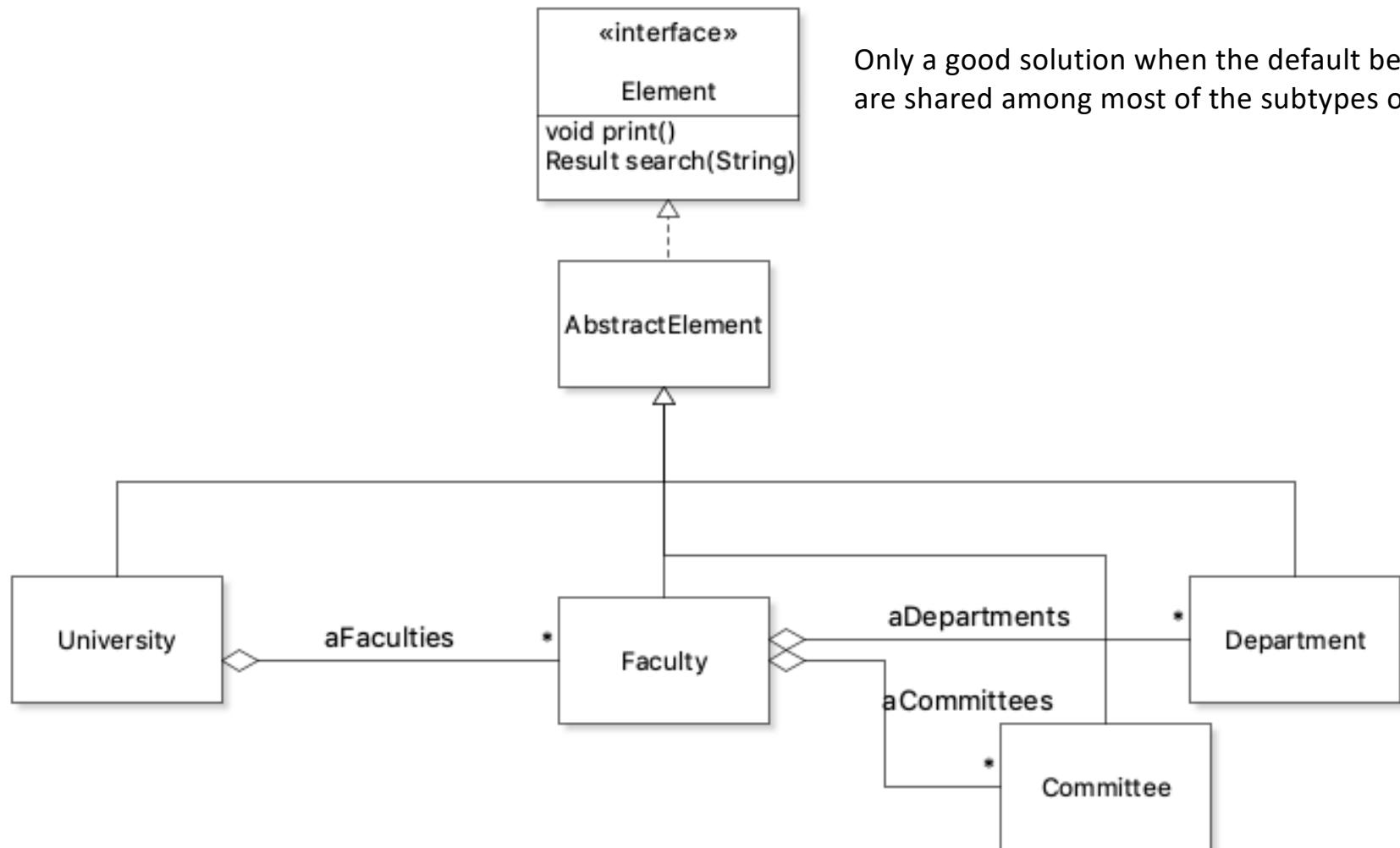
Search if one person belongs to any organization in university

... ...

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

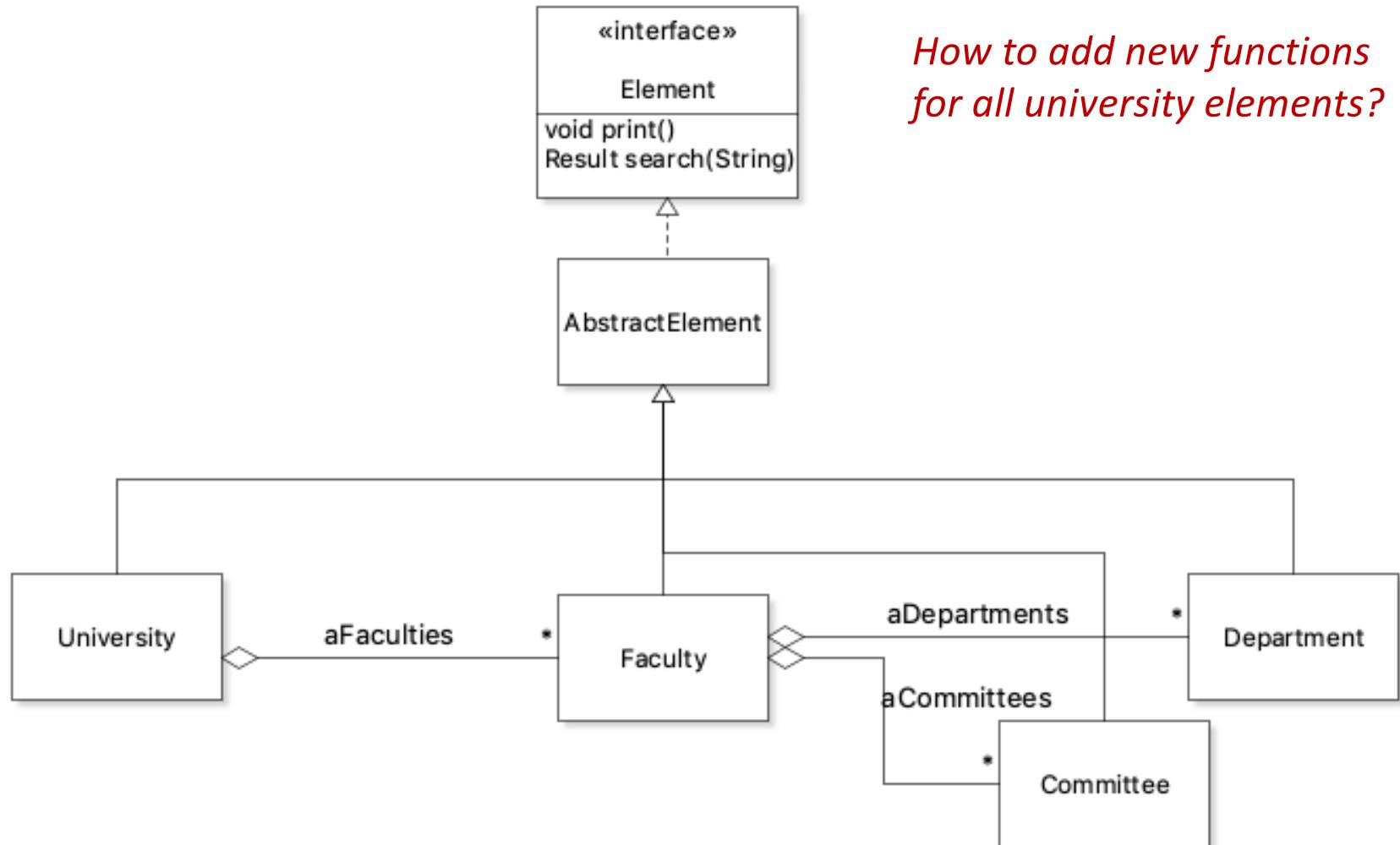


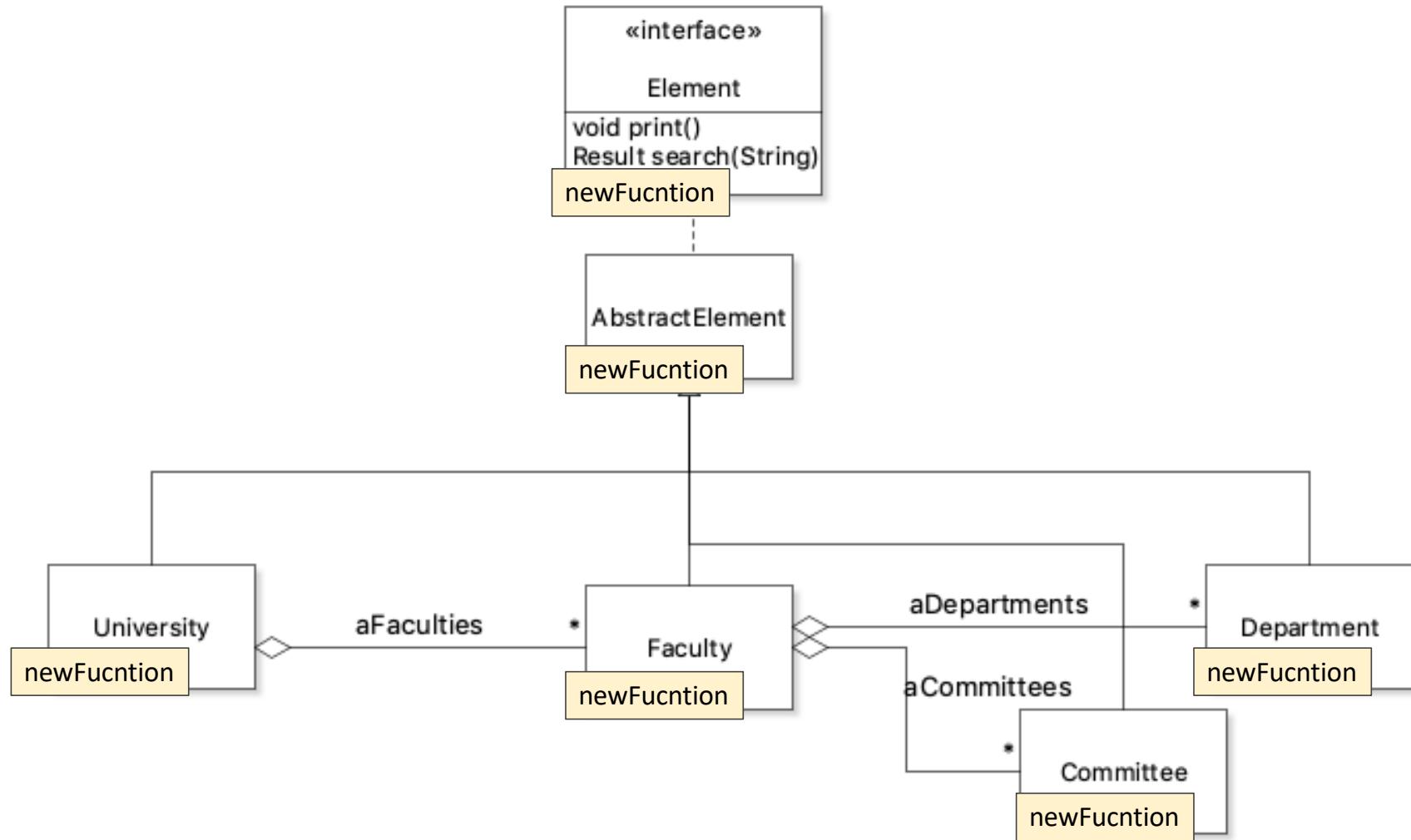




Only a good solution when the default behaviors are shared among most of the subtypes of element.

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



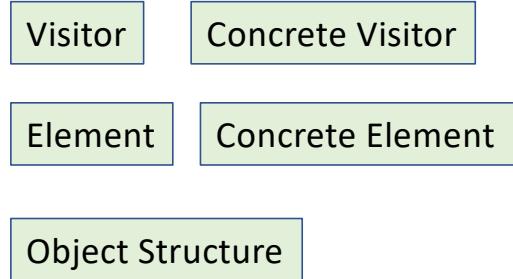


Visitor

- 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:



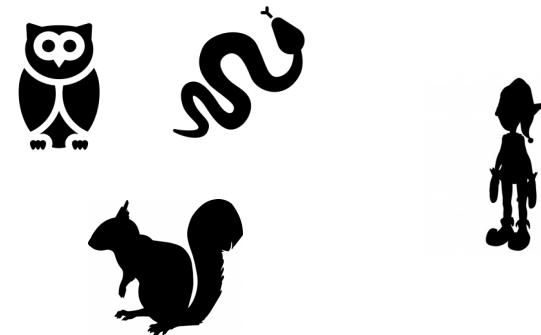
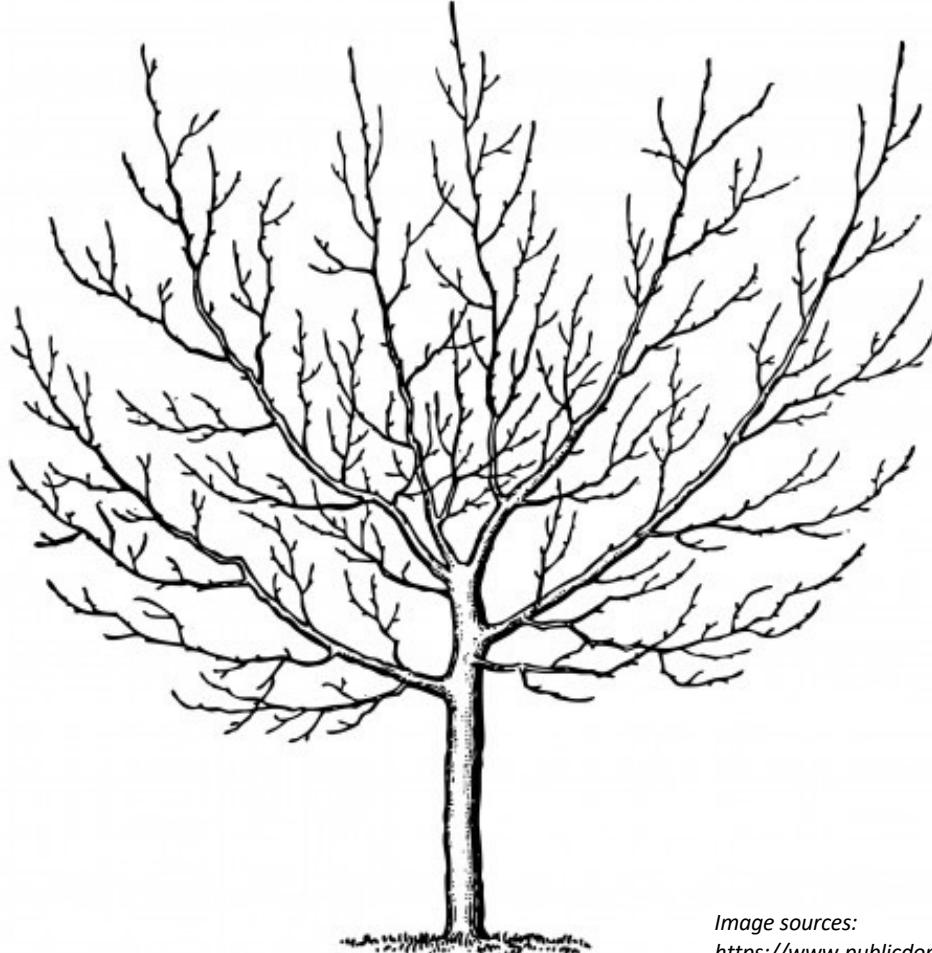


Image sources:

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

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

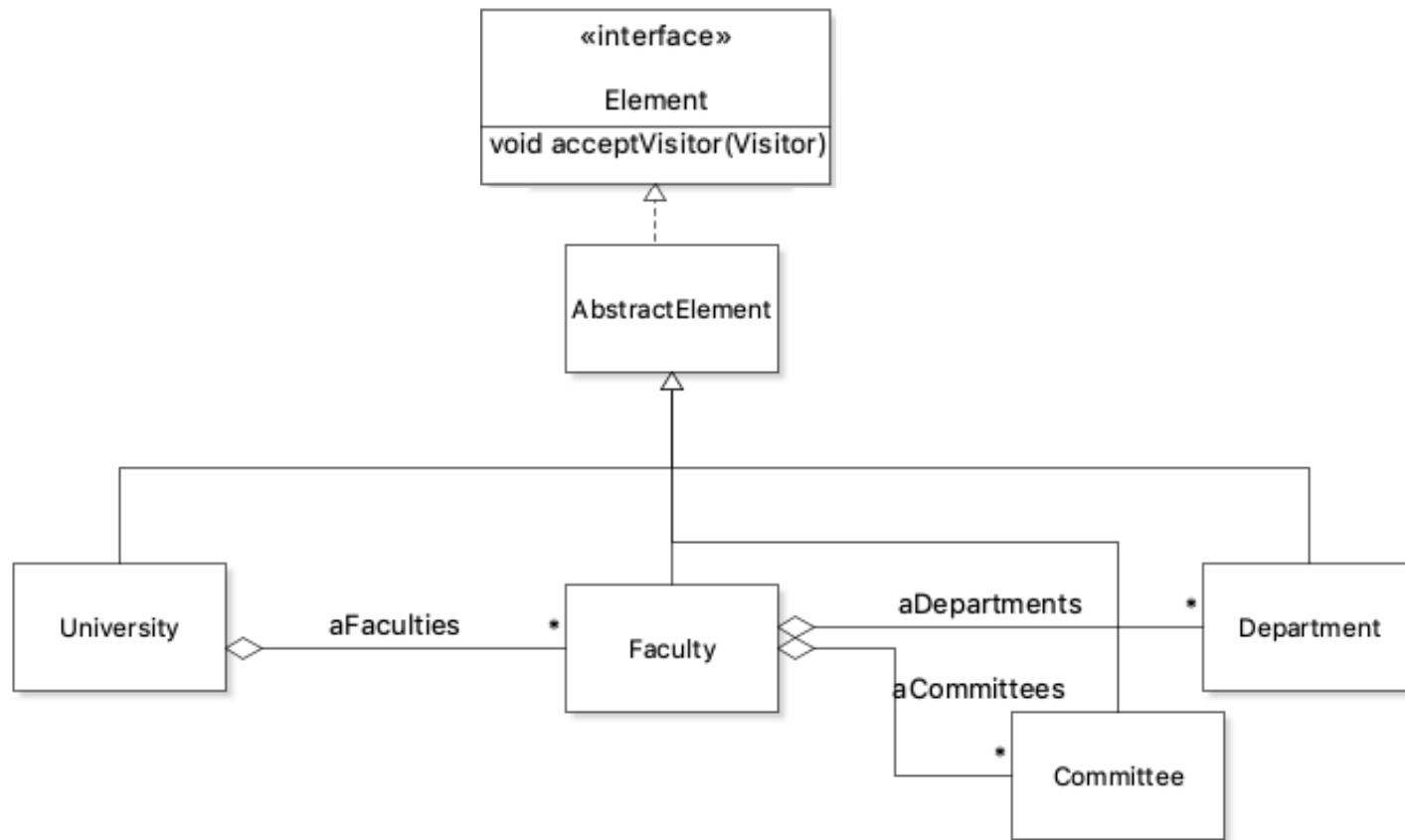
<https://www.publicdomainpictures.net/pictures/190000/nahled/squirrel-silhouette-1469799208bea.jpg>

```

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

```

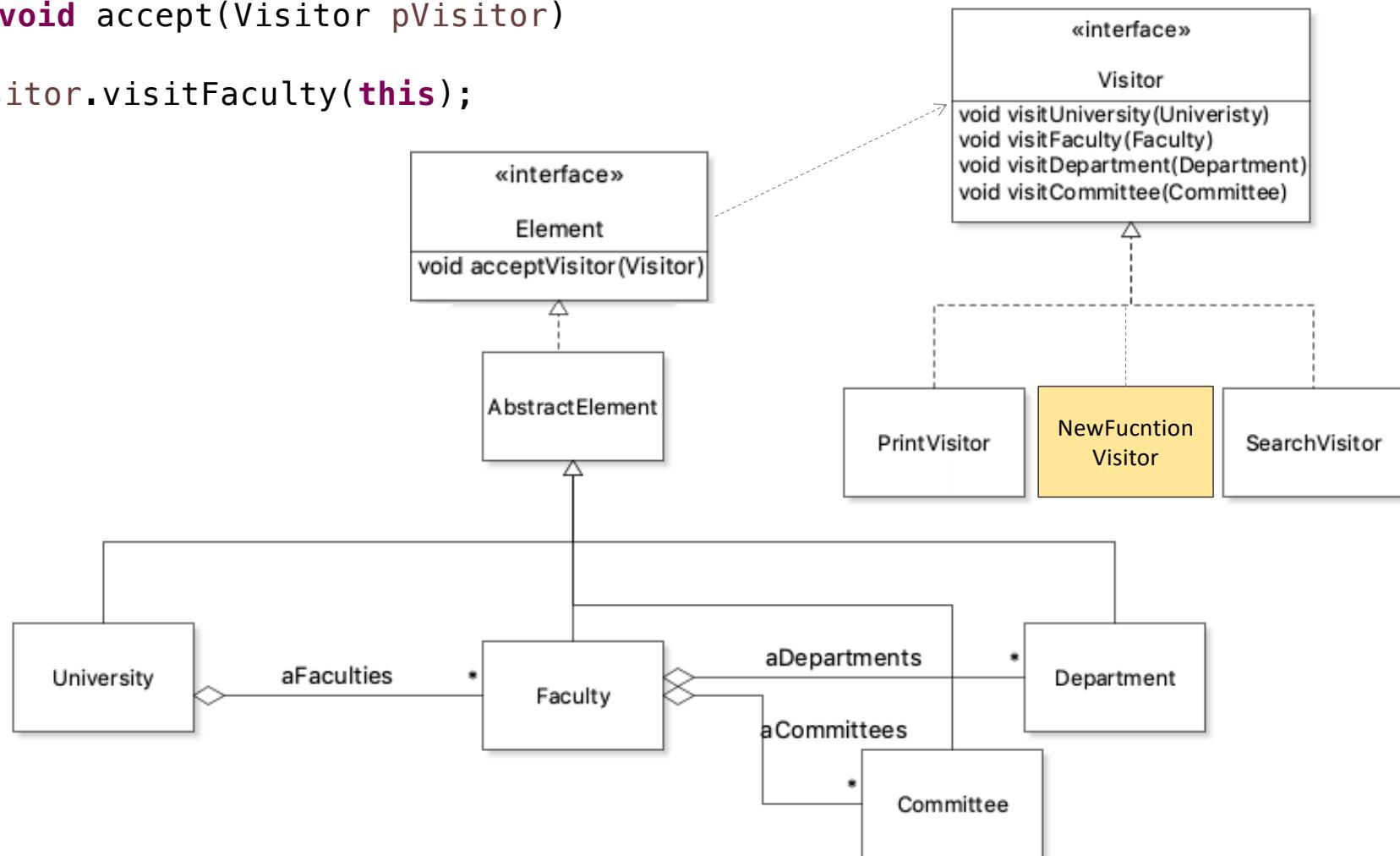
How to pass around functions?

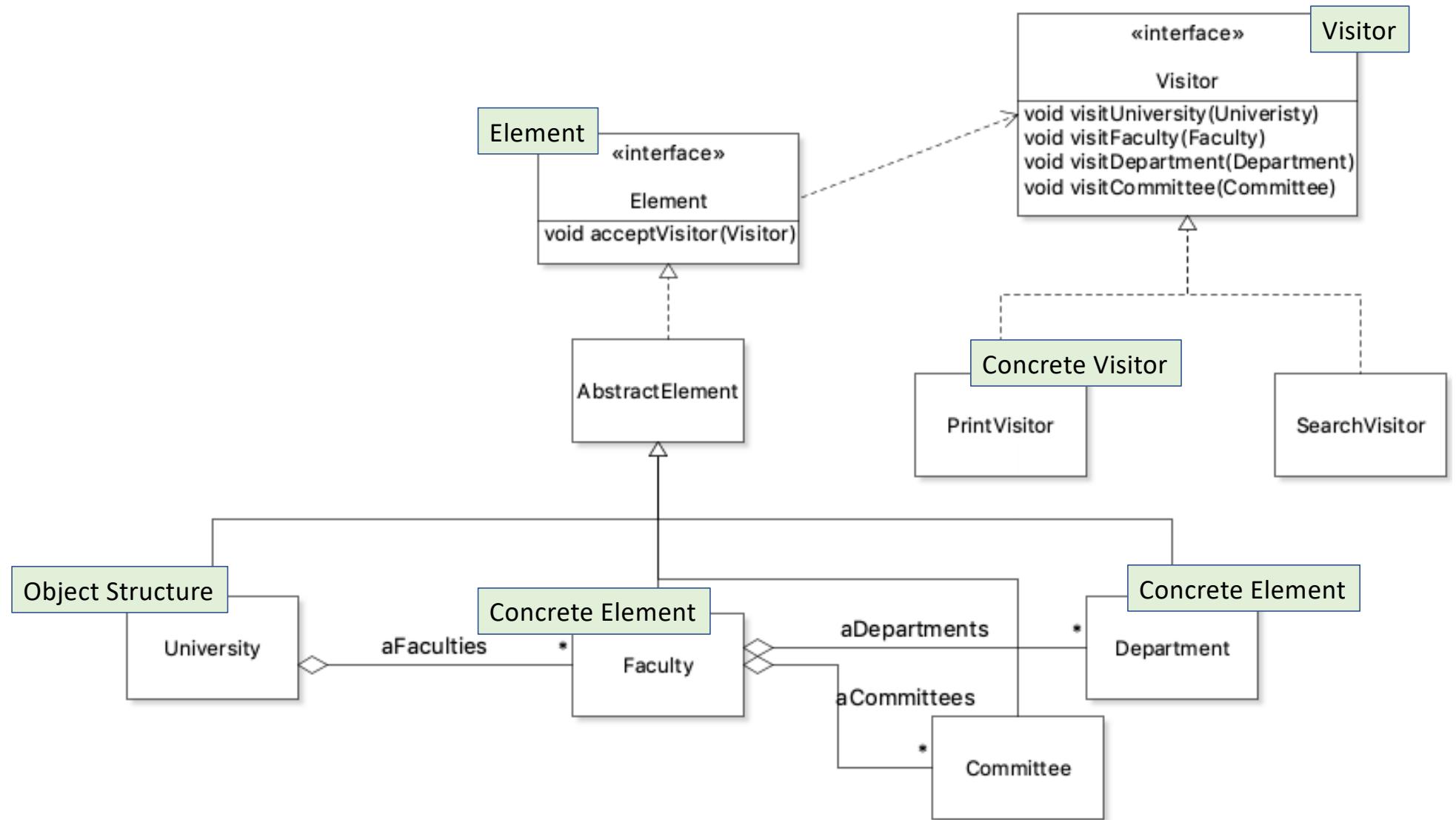


```

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

```

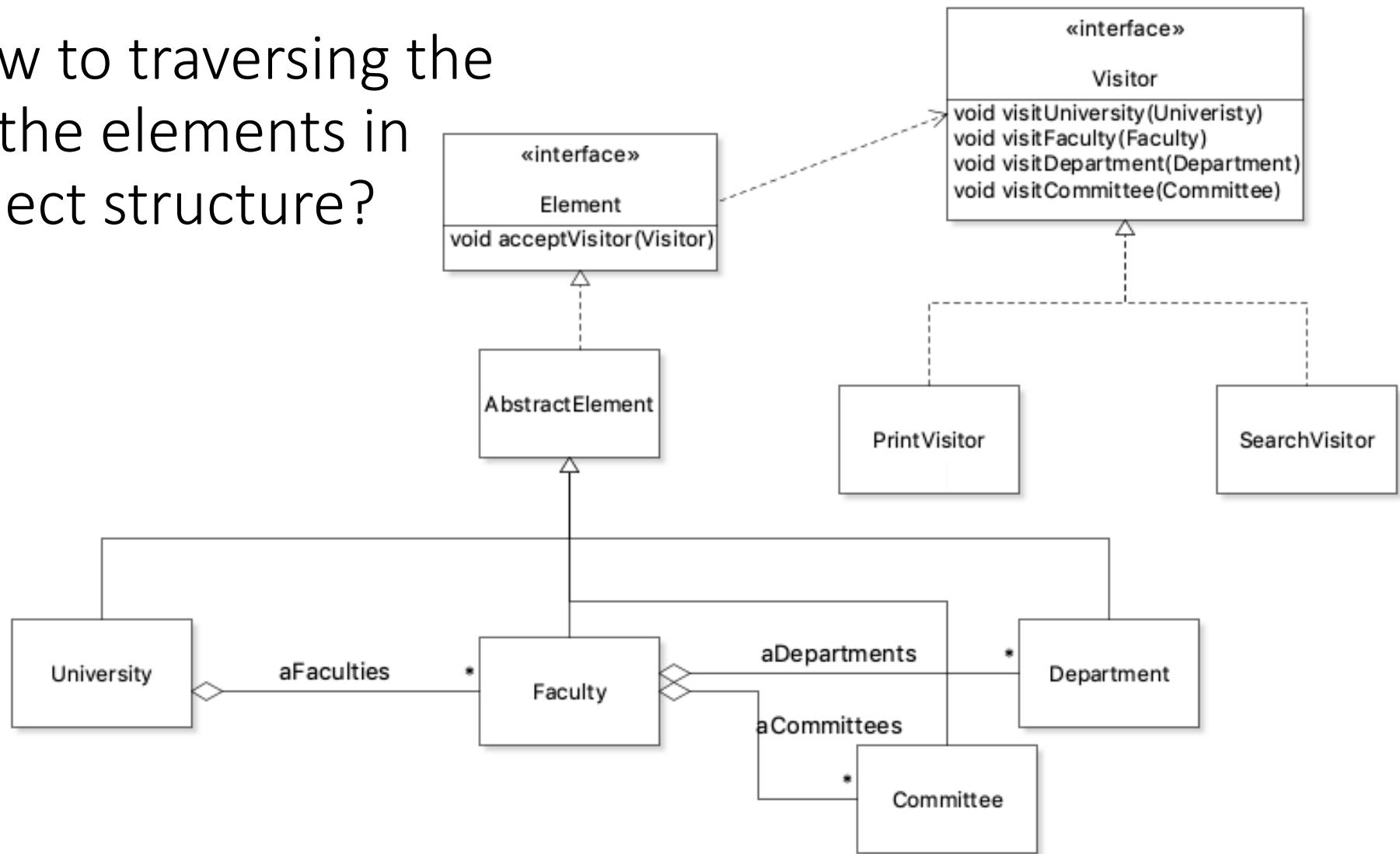




Please think and discuss the consequences
when applying Visitor pattern.

Activity 1:

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.

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);  
}  
}
```

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→

```
for( Iterator<Faculty> i = pUniversity.getFaculties();
    i.hasNext(); )
{
    i.next().accept(this);
}
```

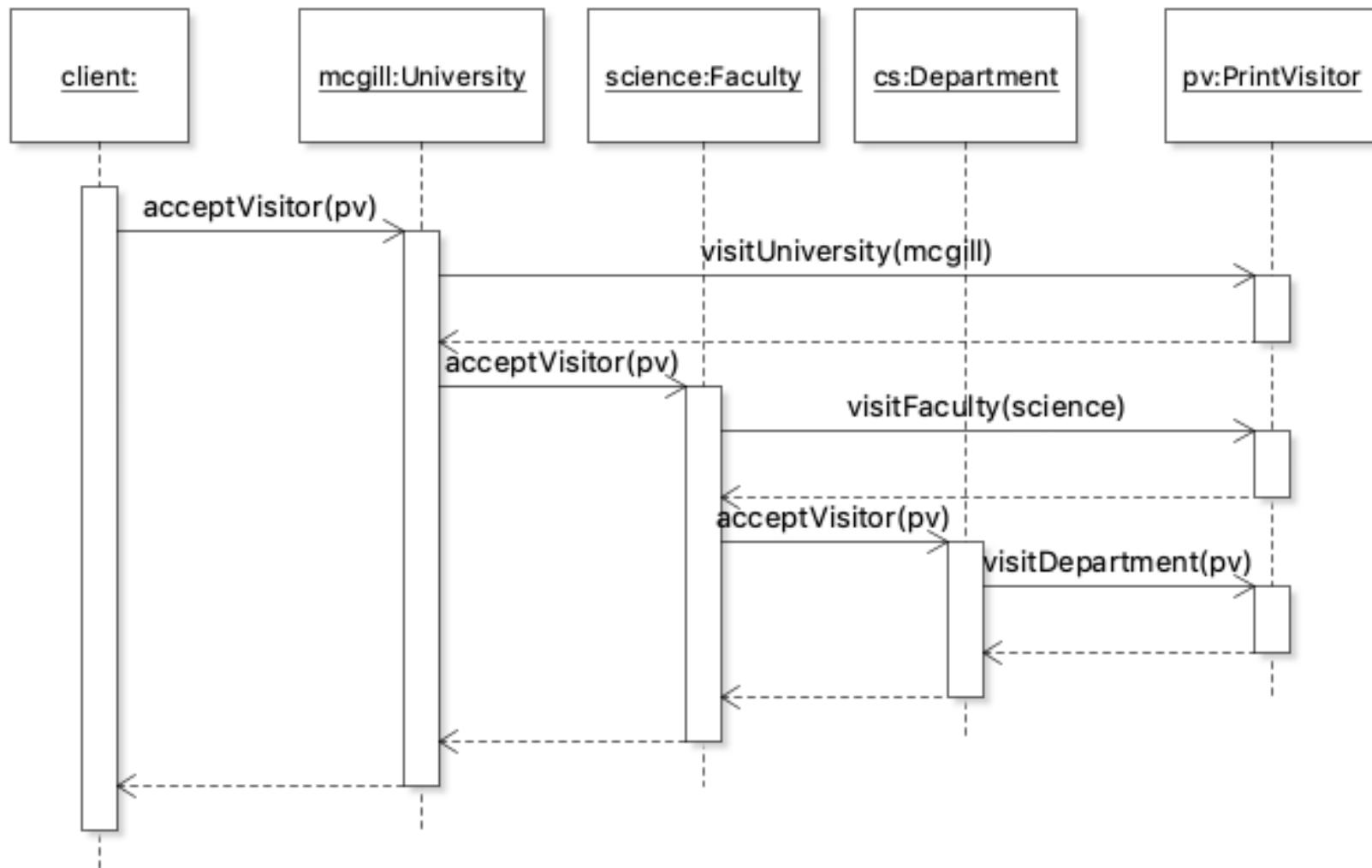
Activity2: Sequence Diagram for Visitor pattern

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

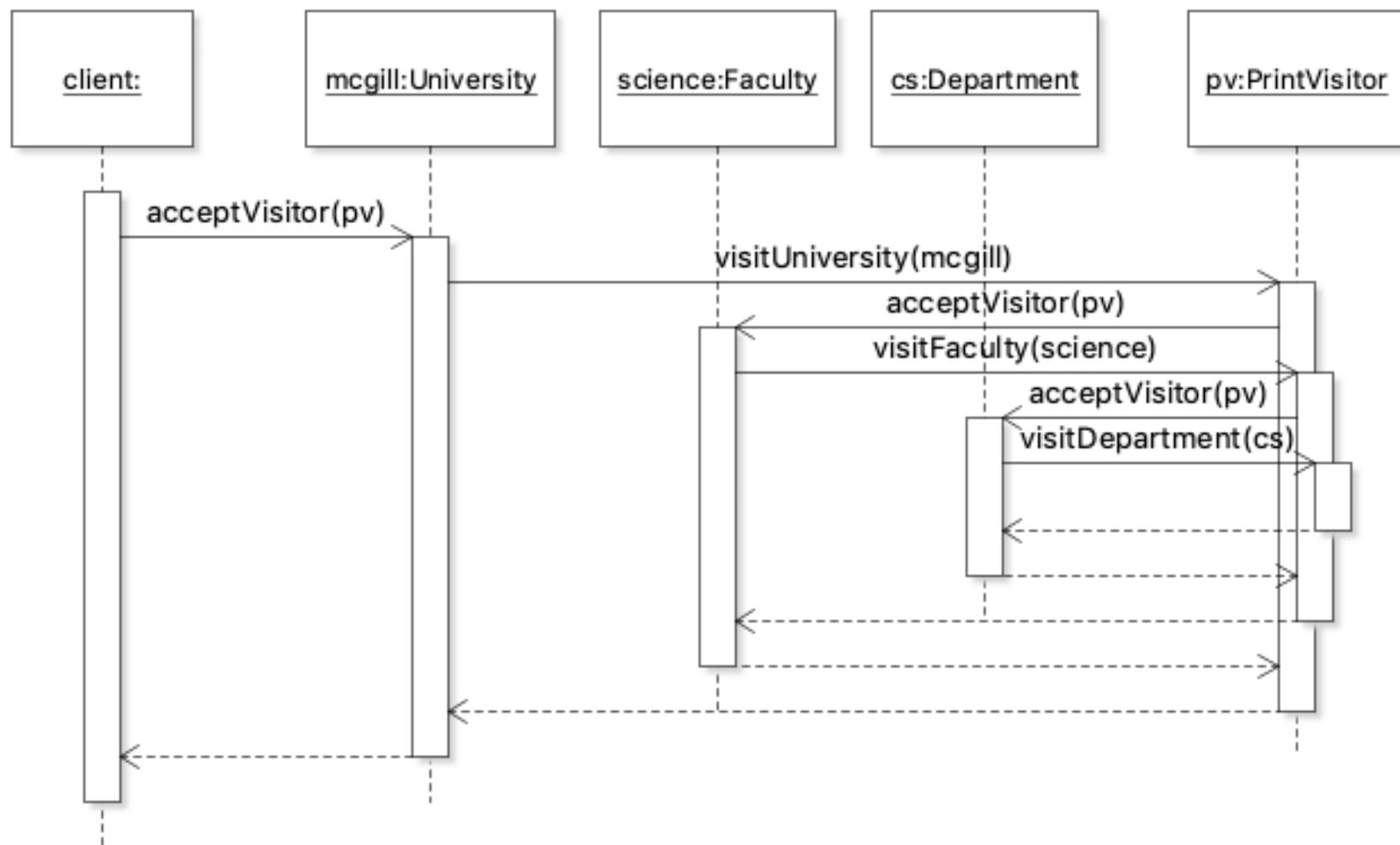
mcGill.addFaculty(science);
science.addDepartment(cs);

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

Traverse in the object structure



Sequence diagram for traversal in Visitor?



Demo on McGill Visitor

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

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Intrepreter Template Method ✓
Object	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

- 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

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter
			Template Method ✓
Object	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

- Concerned with how classes and objects are composed to form larger structures.
- Patterns have distinct intention
- Structural object patterns can change the composition at run-time, which is impossible with static class composition.

	Creational	Structural	Behavioral
Class	Factory Method	Adapter (class)	Interpreter
Object			Template Method ✓
	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 _____;
 - An iterator object encapsulates _____;
 - A command object encapsulates _____;
 - A visitor object encapsulates _____;

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.