M8 (a) – Inversion of Control

Jin L.C. Guo

Image Source: https://c1.staticflickr.com/9/8363/29350436510_efe6626995_b.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.

Job Hunting Example

```
public interface JobSeeker
{
	public void noticeMe();
}
```

```
public interface JobProvider
{
	public void acceptApplication(JobSeeker pJobSeeker);
	public void noticeCandidates();
}
```

```
public class Company implements JobProvider
{
   private JobSeeker aJobseeker;
   private boolean applicationAccepted=false;
   @Override
   public void acceptApplication(JobSeeker pJobseeker)
   {
      assert pJobseeker != null;
      aJobseeker = pJobseeker;
      applicationAccepted = true;
   }
   @Override
   public void noticeCandidates() {
      if(applicationAccepted)
         aJobseeker.noticeMe();
   }
}
```

Callback method

```
public class UndergradJobSeeker implements JobSeeker
{
    private int aSkillLevel = 5;

    @Override
    public void noticeMe()
    {
        practiceDesignPatterns();
    }

    private void practiceDesignPatterns()
    {
        aSkillLevel++;
    }

}
```

Provide the interview schedule to JobSeeker?

```
public class Company implements JobProvider
{
   private LocalDateTime aInterviewSchedule;
   ……
   @Override
   public void noticeCandidates() {
      if(acceptApplication)
         aJobseeker.noticeMe(); //Callback method
   }
   /**
   * Setup interview date is three days from today
   */
   private void scheduleInterview() {
      aInterviewSchedule = LocalDateTime.now().plusDays(3);
   }
}
```

Provide the interview schedule to JobSeeker?

```
public class Company implements JobProvider
{
    private LocalDateTime aInterviewSchedule;
    ……
    @Override
    public void noticeCandidates() {
        if(acceptApplication)
            aJobseeker.noticeMe(aInterviewSchedule); //Callback method
    }

    /**
    * Setup interview date is three days from today
    */
    private void scheduleInterview() {
        aInterviewSchedule = LocalDateTime.now().plusDays(3);
    }

}
```

Provide the interview schedule to JobSeeker?

```
public class Company implements JobProvider
{
    private LocalDateTime aInterviewSchedule;
    ……
    @Override
    public void noticeCandidates() {
        if(acceptApplication)
            aJobseeker.noticeMe(this); //Callback method
    }
    /**
    * Setup interview date is three days from today
    */
    private void scheduleInterview() {
        aInterviewSchedule = LocalDateTime.now().plusDays(3);
    }


}
```

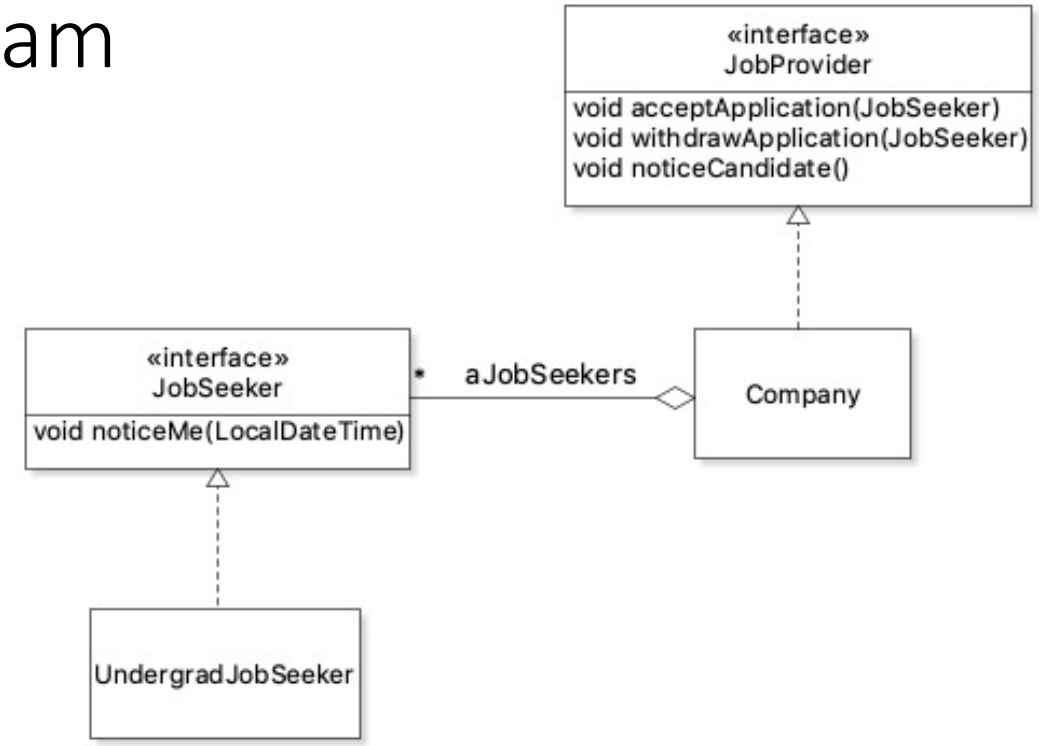
Plus, a public method to get `aInterviewSchedule`

JOBSEEKER CAN WITHDRAW APPLICATION

JOBPROVIDER ACCEPT MORE THAN ONE APPLICATIONS

Activity 1: Add additional functions to the current design.

Class diagram



JobSeeker and JobProvider are loosed-coupled

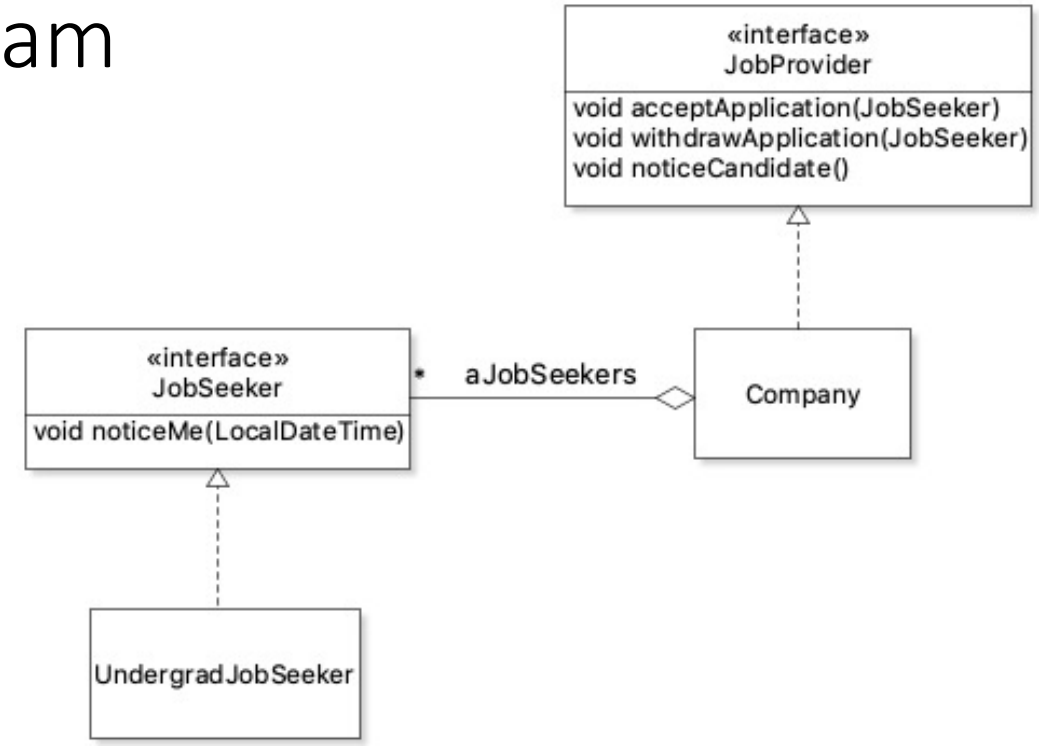
Observer Pattern

- Intent

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

- Participants:
  - Subject
  - Observer
  - Concrete Subject
  - Concrete Observer

Class diagram



JobSeeker and JobProvider are loosed-coupled

Activity 2: Matching Participants with (potential) Responsibilities

Subject **Observer** **Concrete Subject** **Concrete Observer**

defines an updating interface for objects that should be notified of changes in a subject.

implements the updating interface to keep its state consistent with the subject's.

stores state that should stay consistent with the subject's.

maintains a reference to a ConcreteSubject object.

sends a notification to its observers when its state changes.

provides an interface for attaching and detaching Observer objects

stores state of interest to ConcreteObserver objects.

Observer Pattern for more complex situations

- Different departments/teams in the company need the information of jobseekers:

  Design team in SE development department
  Needs candidates who are specialized in design with minimal 5-year experience

  Testing team in SE development department
  Needs candidate who are specialized in testing with reference letters.

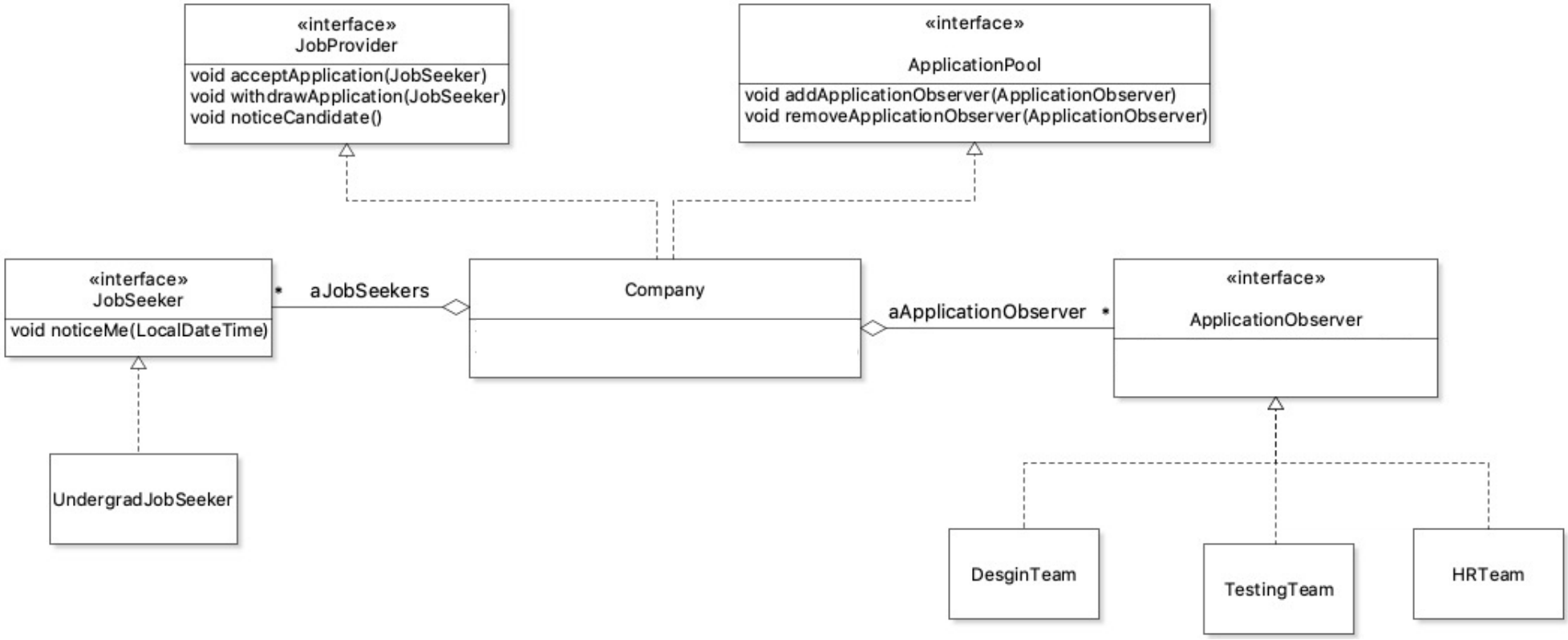
  HR departments
  Performs analysis on the statistics of all job seekers

```
public interface JobSeeker
{
   public void noticeMe(LocalDateTime date);
   public TechSpecialty getTechSpecialty();
   public int getYearOfExperience();
   public boolean haveReference();
}
```

provides an interface for attaching and detaching Observer objects?

```
public class Company implements JobProvider , ApplicationPool
{
    List<JobSeeker> aJobseekers;
    boolean acceptApplication=false;
    Map<JobSeeker, LocalDateTime> aInterviewSchedules;
    private List<ApplicationObserver> aApplicationObservers;

    @Override
    public void addApplicationObserver(ApplicationObserver pApplicationObservers)
    {


    }

    @Override
    public void removeApplicationObserver(ApplicationObserver pApplicationObservers)
    {


    }
```

What is the state of interest for those teams


«interface»
JobProvider
void acceptApplication(JobSeeker)
void withdrawApplication(JobSeeker)
void noticeCandidate()
«interface»
ApplicationPool
void addApplicationObserver(ApplicationObserver)
void removeApplicationObserver(ApplicationObserver)
«interface»
JobSeeker
void noticeMe(LocalDateTime)
*
aJobSeekers
Company
aApplicationObserver
*
«interface»
ApplicationObserver
UndergradJobSeeker
DesginTeam
TestingTeam
HRTeam

When and how to send Notification

- Requirements:

Design team in SE development department

Needs candidates who are specialized in design with minimal 5-years experience

Testing team in SE development department

Needs candidate who are specialized in testing with reference letters.

HR departments

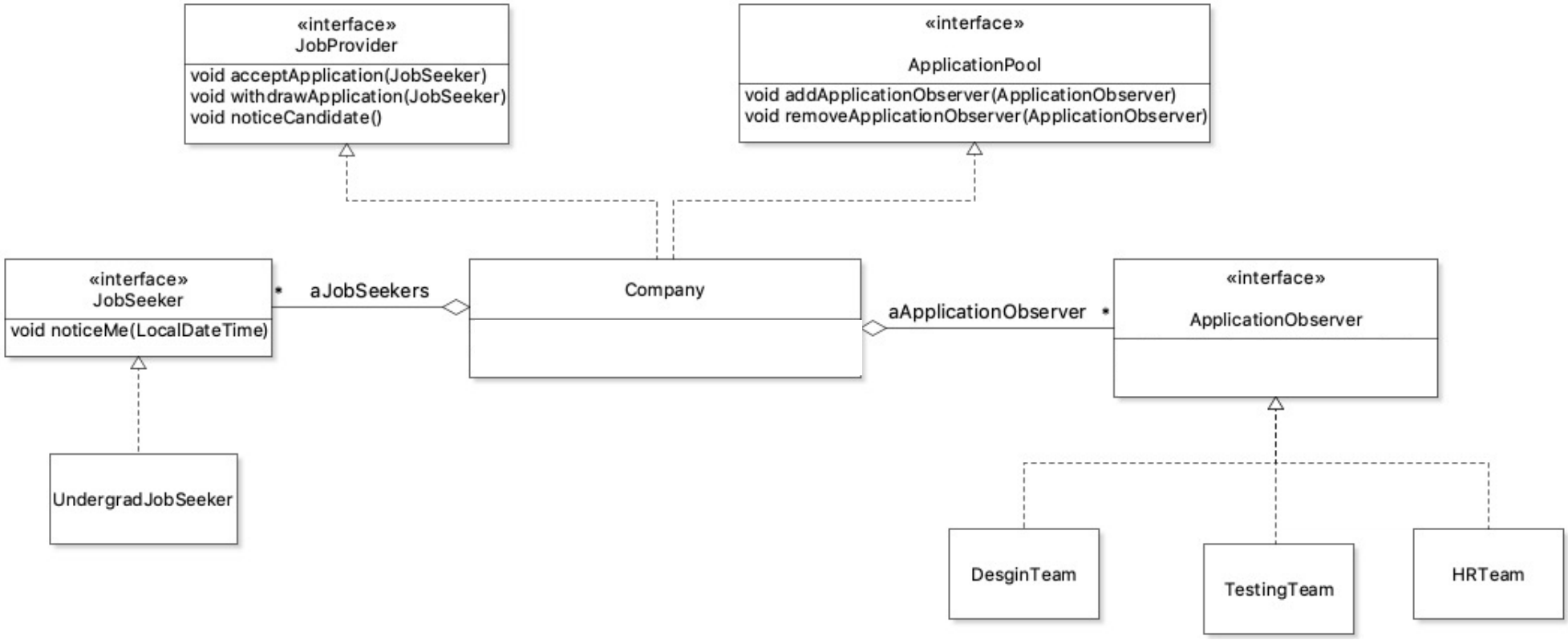
Performs analysis on the statistics of all job seekers

When and how to send Notification

Who should trigger the notification?

`ApplicationPool` sends notification as soon as an application is added or removed.

`ApplicationPool` provides a notification method to be called by client


«interface»
JobProvider
void acceptApplication(JobSeeker)
void withdrawApplication(JobSeeker)
void noticeCandidate()
«interface»
ApplicationPool
void addApplicationObserver(ApplicationObserver)
void removeApplicationObserver(ApplicationObserver)
«interface»
JobSeeker
void noticeMe(LocalDateTime)
*
aJobSeekers
Company
aApplicationObserver
*
«interface»
ApplicationObserver
UndergradJobSeeker
DesginTeam
TestingTeam
HRTeam

When and how to send Notification

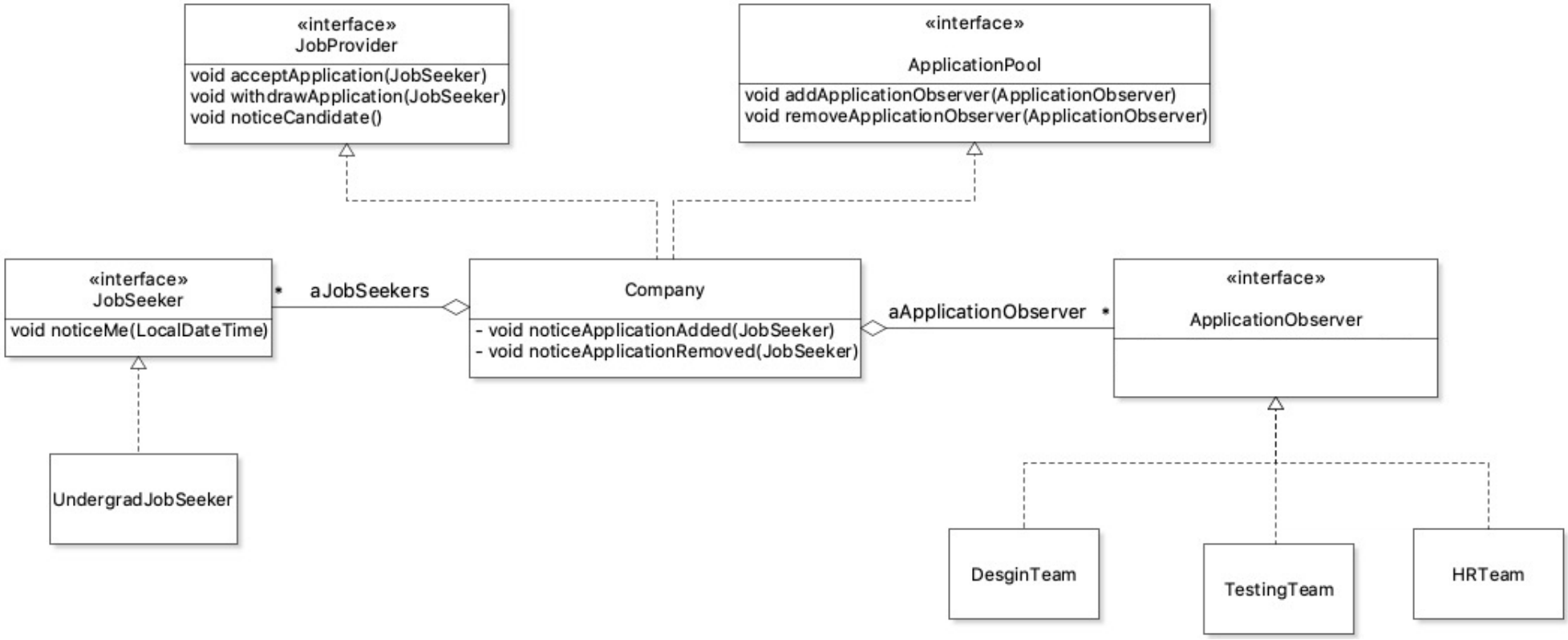
Data Flow Strategy?

`ApplicationPool` sends observers detailed information about the change, whether `ApplicationObserver` want it or not

Push model

`ApplicationPool` sends the most minimal notification, and `ApplicationObserver` ask for details explicitly thereafter.

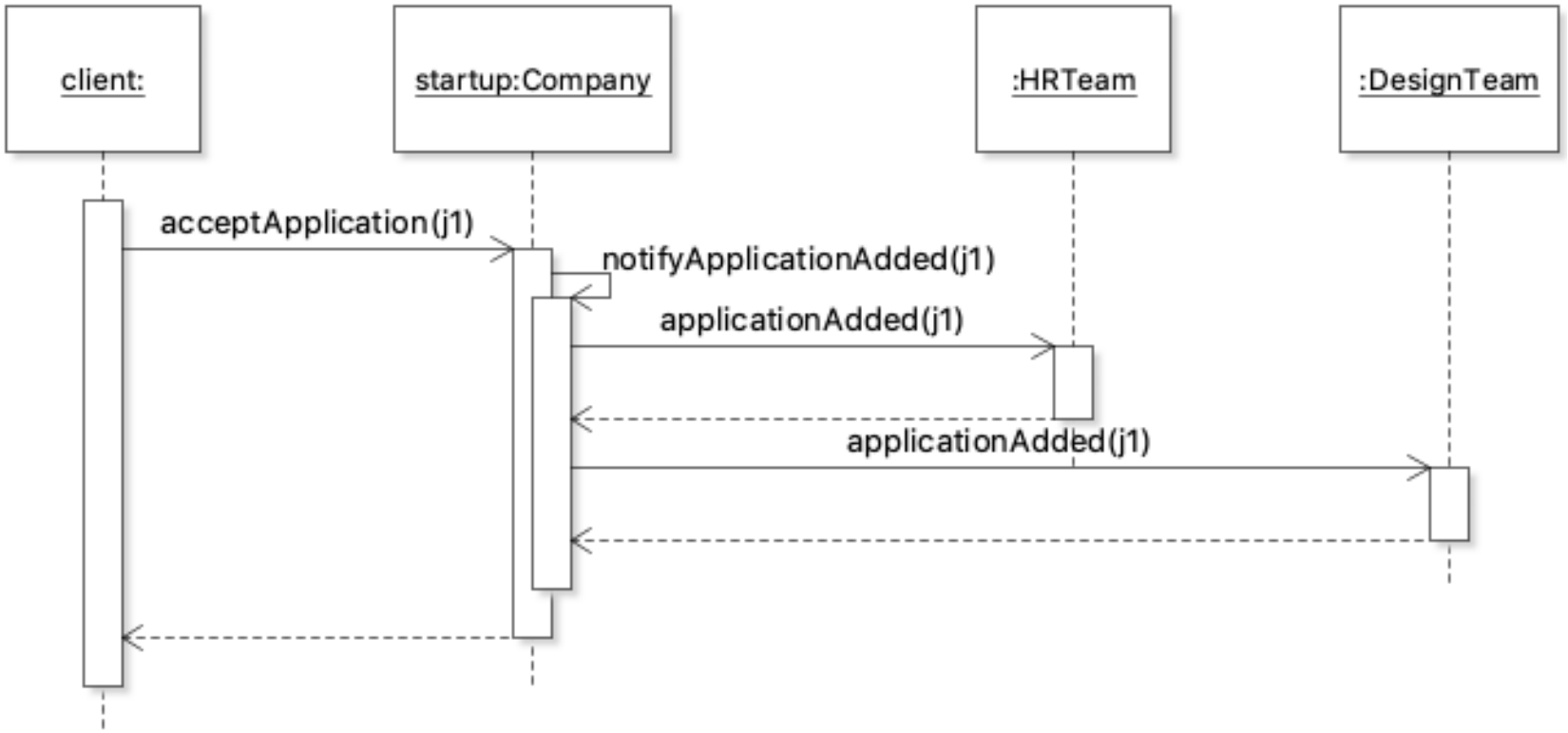
Pull model


«interface»
JobProvider
void acceptApplication(JobSeeker)
void withdrawApplication(JobSeeker)
void noticeCandidate()
«interface»
ApplicationPool
void addApplicationObserver(ApplicationObserver)
void removeApplicationObserver(ApplicationObserver)
«interface»
JobSeeker
void noticeMe(LocalDateTime)
*
aJobSeekers
Company
- void noticeApplicationAdded(JobSeeker)
- void noticeApplicationRemoved(JobSeeker)
aApplicationObserver
*
«interface»
ApplicationObserver
UndergradJobSeeker
DesginTeam
TestingTeam
HRTeam

Acitivty3: Draw sequence diagram

```
Company startup = new Company();
ApplicationObserver hrTeam = new HRTeam();
ApplicationObserver designTeam = new DesignTeam();
startup.addApplicationObserver(hrTeam);
startup.addApplicationObserver(designTeam);

JobSeeker j1 = new UndergradJobSeeker(TechSpecialty.UI_Design, 10, true);


startup.acceptApplication(j1);
```

<= When this statement is executed


client:
startup:Company
:HRTeam
:DesignTeam
acceptApplication(j1)
notifyApplicationAdded(j1)
applicationAdded(j1)
applicationAdded(j1)

Push model

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

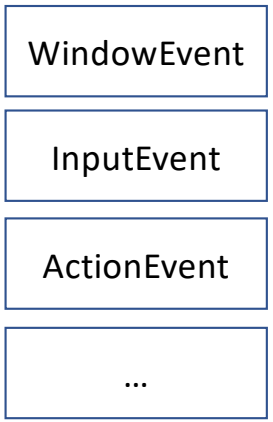
Popup window is hidden

How to capture event and act accordingly

- Define an event handler

implement

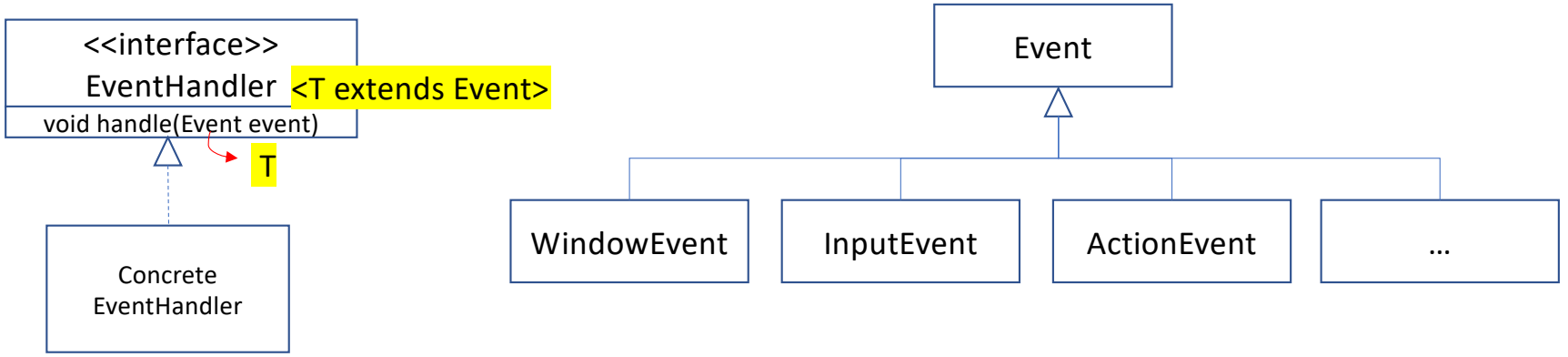
Interface EventHandler<T extends Event>



void handle(T event) **<= Callback method**

Invoked when a specific event of the type for which this handler is registered happens.

How to capture event and act accordingly



```
Public class MyEventHandler implements EventHandler<ActionEvent>
{
   @Override
   public void handle(ActionEvent event)
   {
       //Event Handling steps
   }
}
```

How to capture event and act accordingly

- Instantiate and register the event handler


```
MyEventHandler eventHandler = new MyEventHandler();
Button btn = new Button();
btn.setOnAction(eventHandler);
```

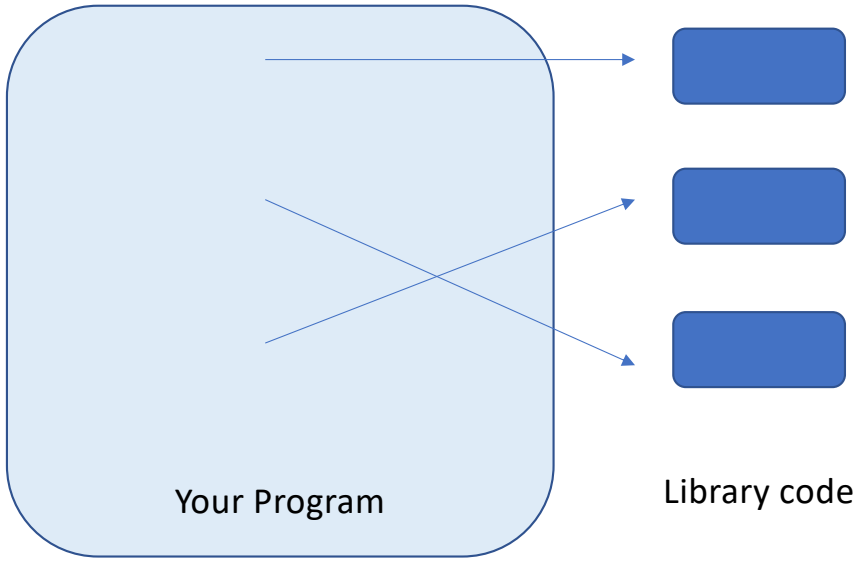
How to capture event and act accordingly

- Instantiate and register the event handler


```
Button btn = new Button();

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

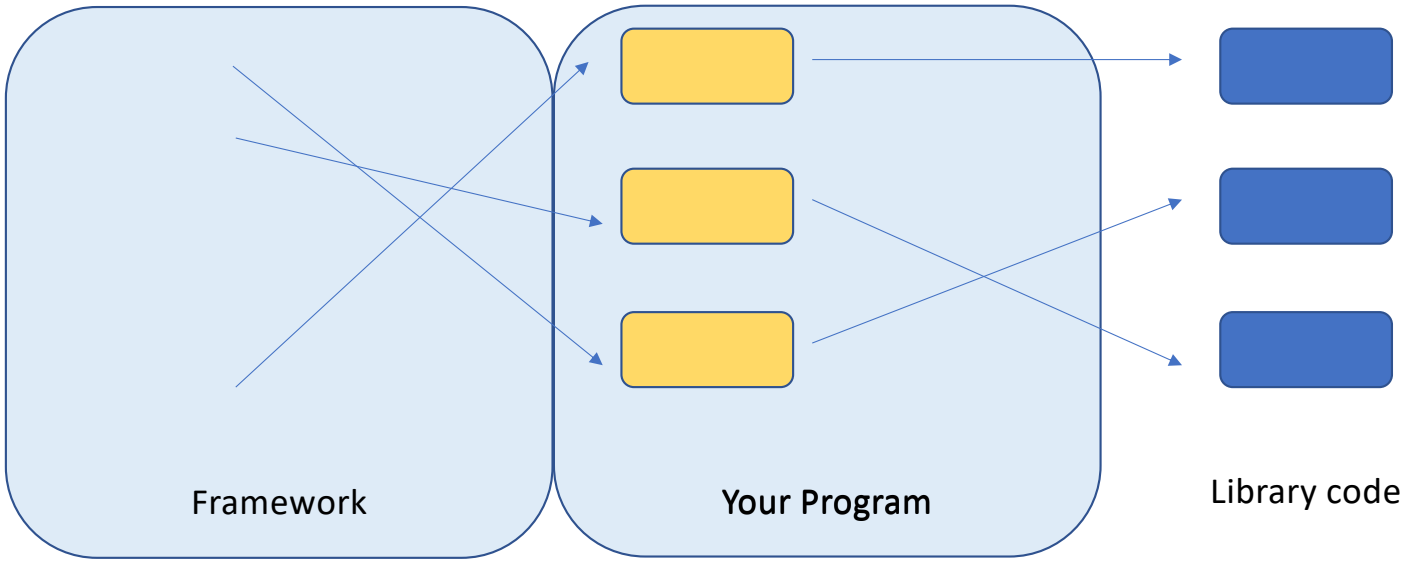
anonymous class

Library vs Framework



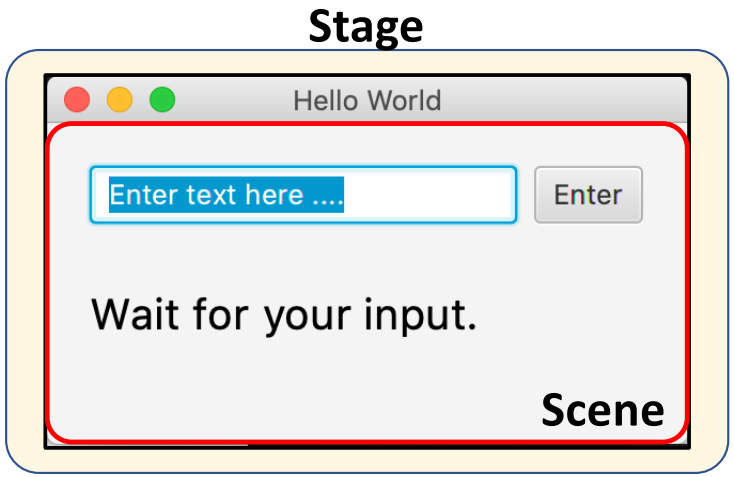

```
public static void main(String[] arg)
```

Library vs Framework

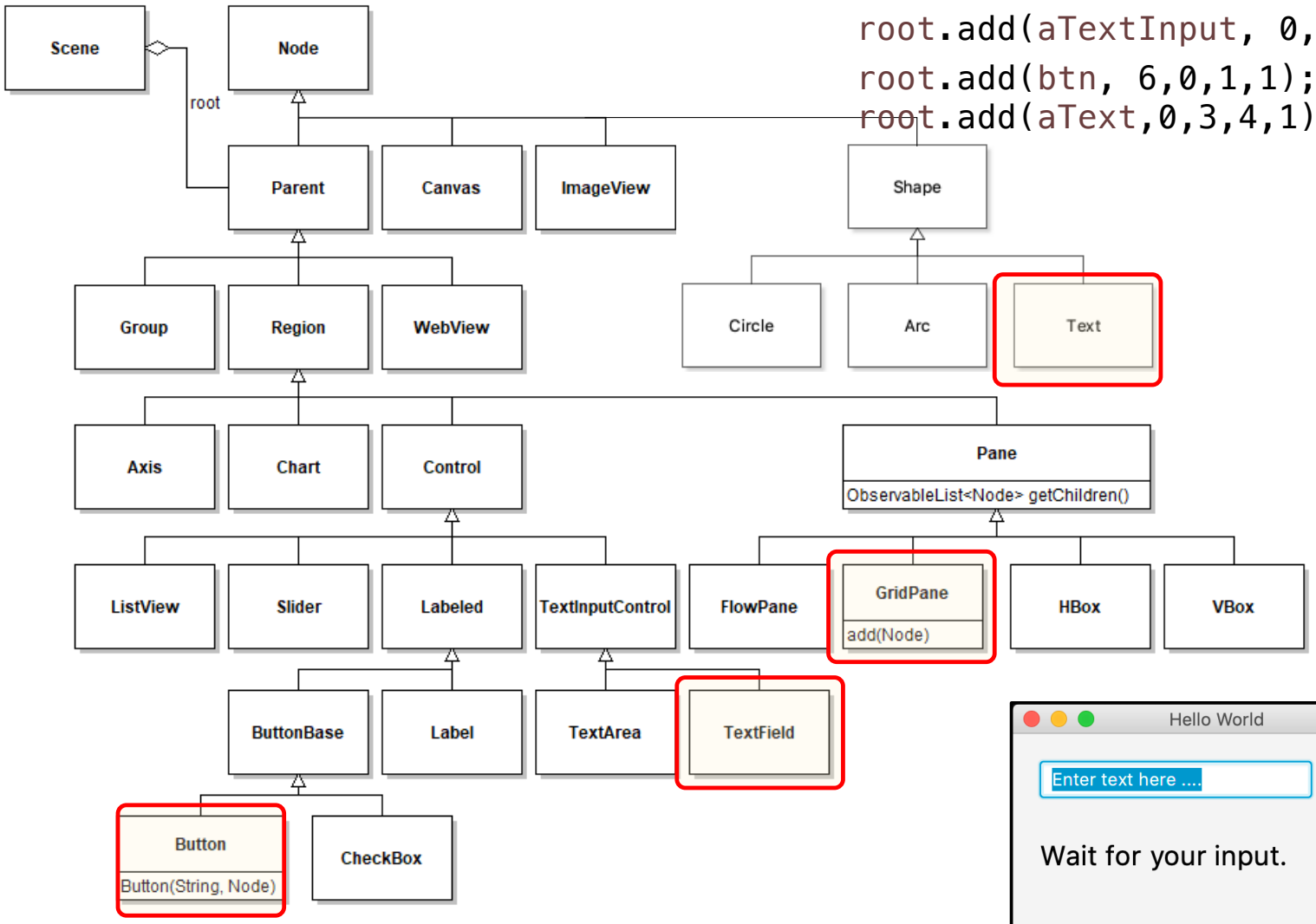


Launch JavaFX framework

```
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();
     }
}
```



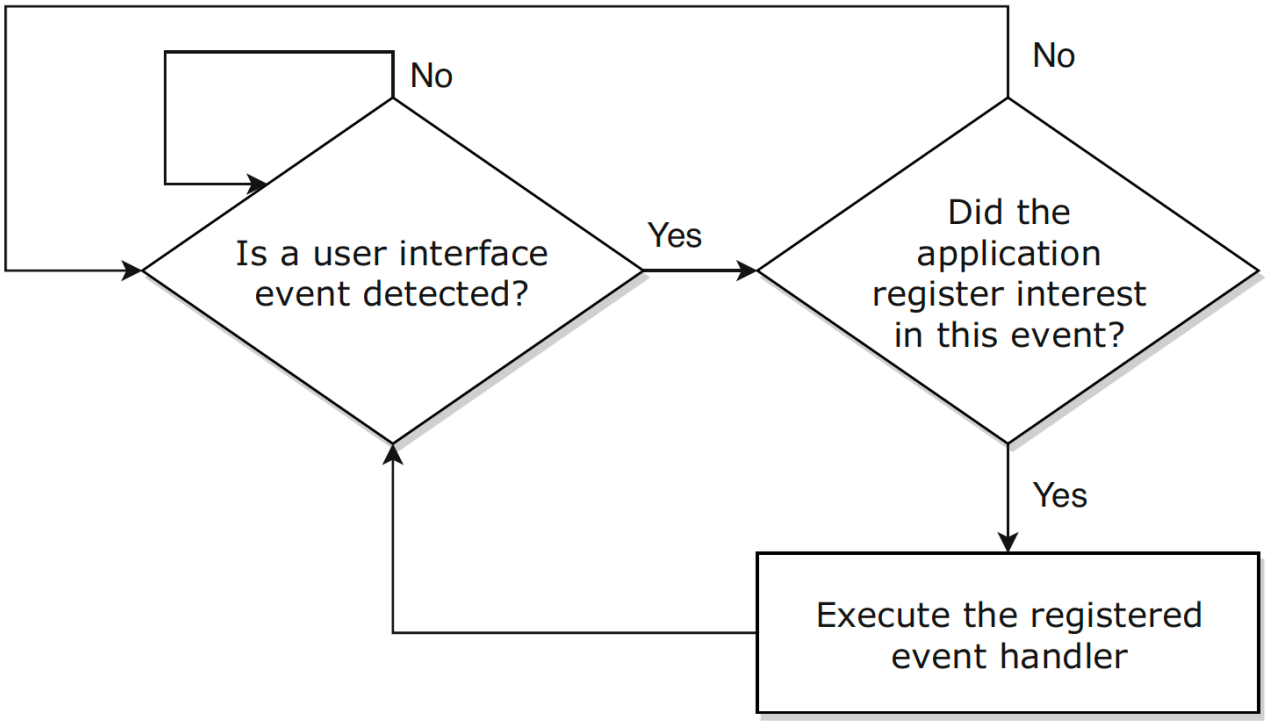

```
GridPane root = new GridPane();
root.add(aTextInput, 0,0,6,1);
root.add(btn, 6,0,1,1);
root.add(aText,0,3,4,1);

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

primaryStage.setScene(scene);
```




```
root.add(aTextInput, 0,0,2,1);
root.add(btn, 6,0,1,1);
root.add(aText,0,3,4,1);
```

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

When does event handling happen?



Text Display Demo


```
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.