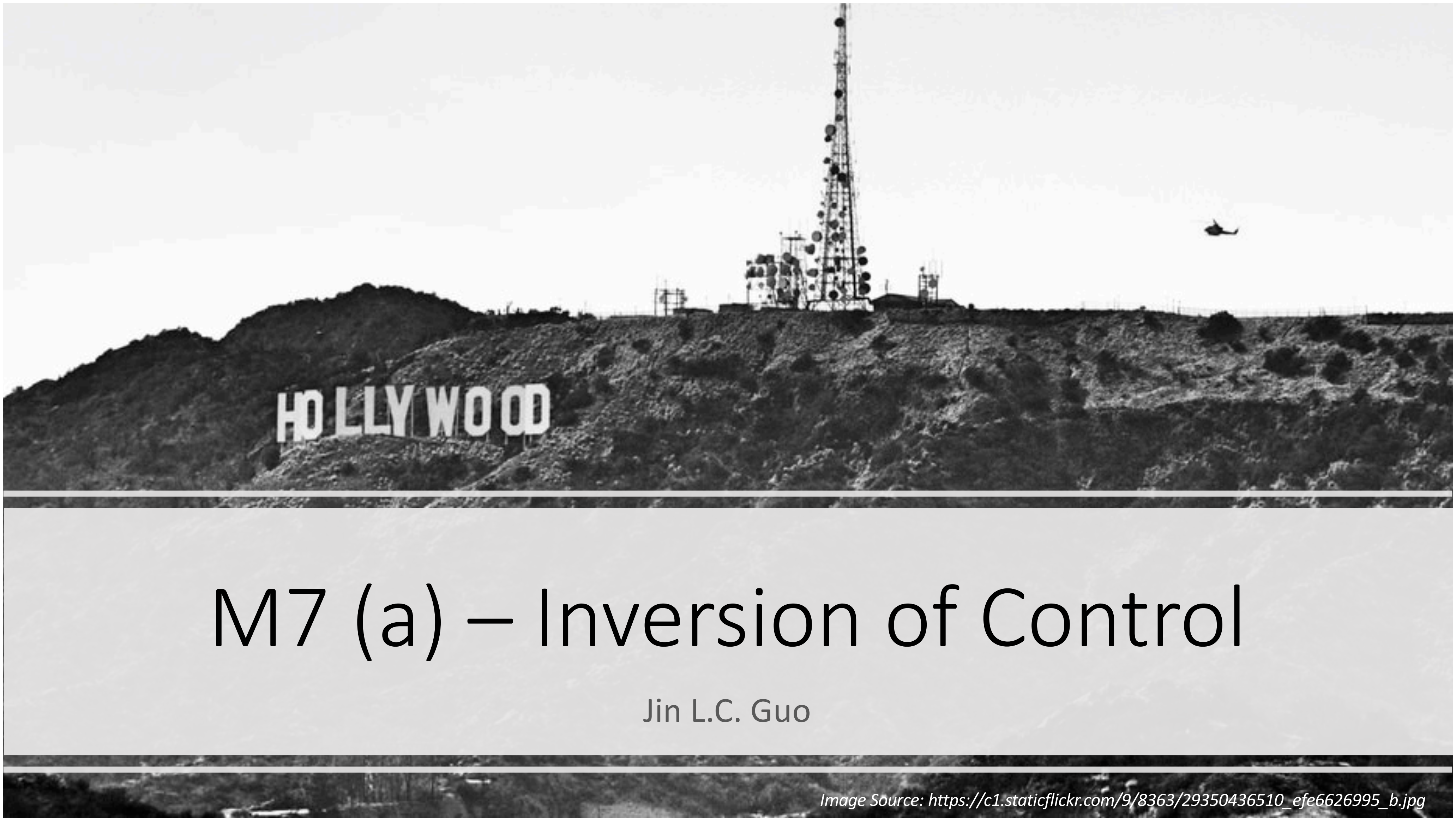


M7 (a) – Inversion of Control

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

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

Job Hunting Example

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

```java
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

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

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

    }

    private void practiceDesignPatterns()
    {
        aSkillLevel++;
    }

}
```

Provide the interview schedule to JobSeeker?

```java
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
   }
```

Plus a public method to get `aInterviewSchedule`

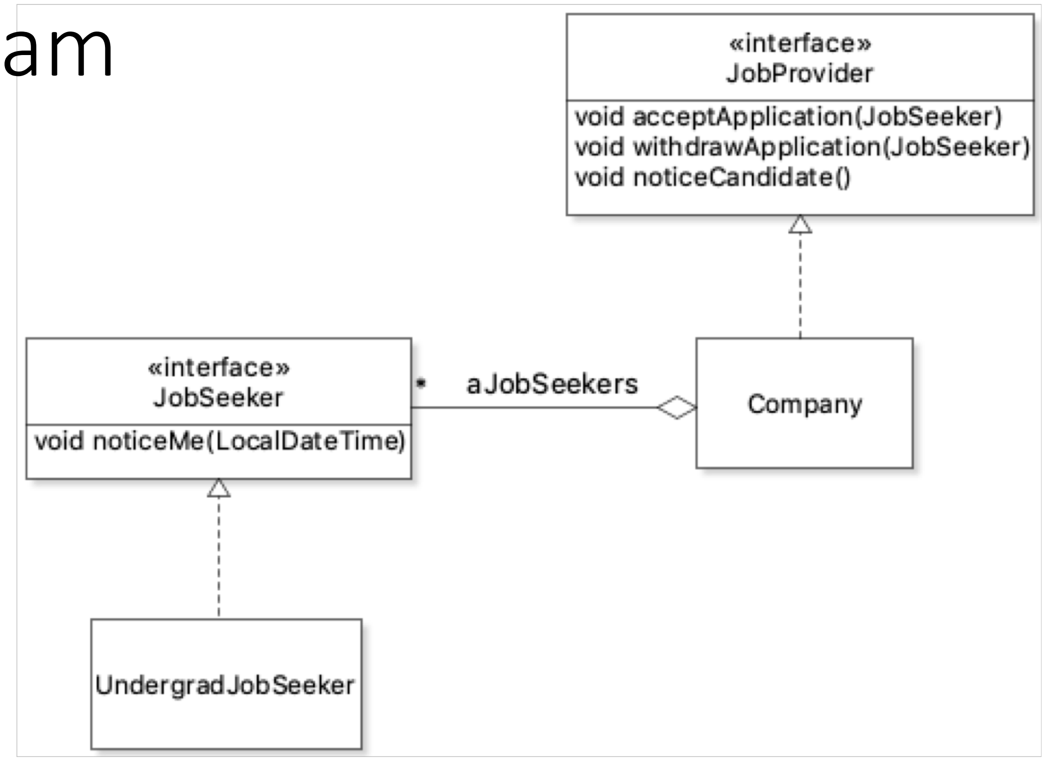
```
   /**
   * Setup interview date is three days from today
   */
   private void scheduleInterview() {
      aInterviewSchedule = LocalDateTime.now().plusDays(3);
   }


}
```

Additional changes

- JobSeeker can withdraw application

- JobProvider accept more than one applications

Class diagram



JobSeeker and JobProvider are loosed-coupled

Demo1

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

Activity1: Matching Participants with 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.

provides an interface for notifying observers.

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

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

```java
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)
    {
        assert pApplicationObservers!=null;
        aApplicationObservers.add(pApplicationObservers);
    }

    @Override
    public void removeApplicationObserver(ApplicationObserver pApplicationObservers)
    {
        assert pApplicationObservers!=null;
        aApplicationObservers.remove(pApplicationObservers);
    }
```

What is the state of interest for those teams

provides an interface for attaching and detaching Observer objects?

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

When and how to send Notification

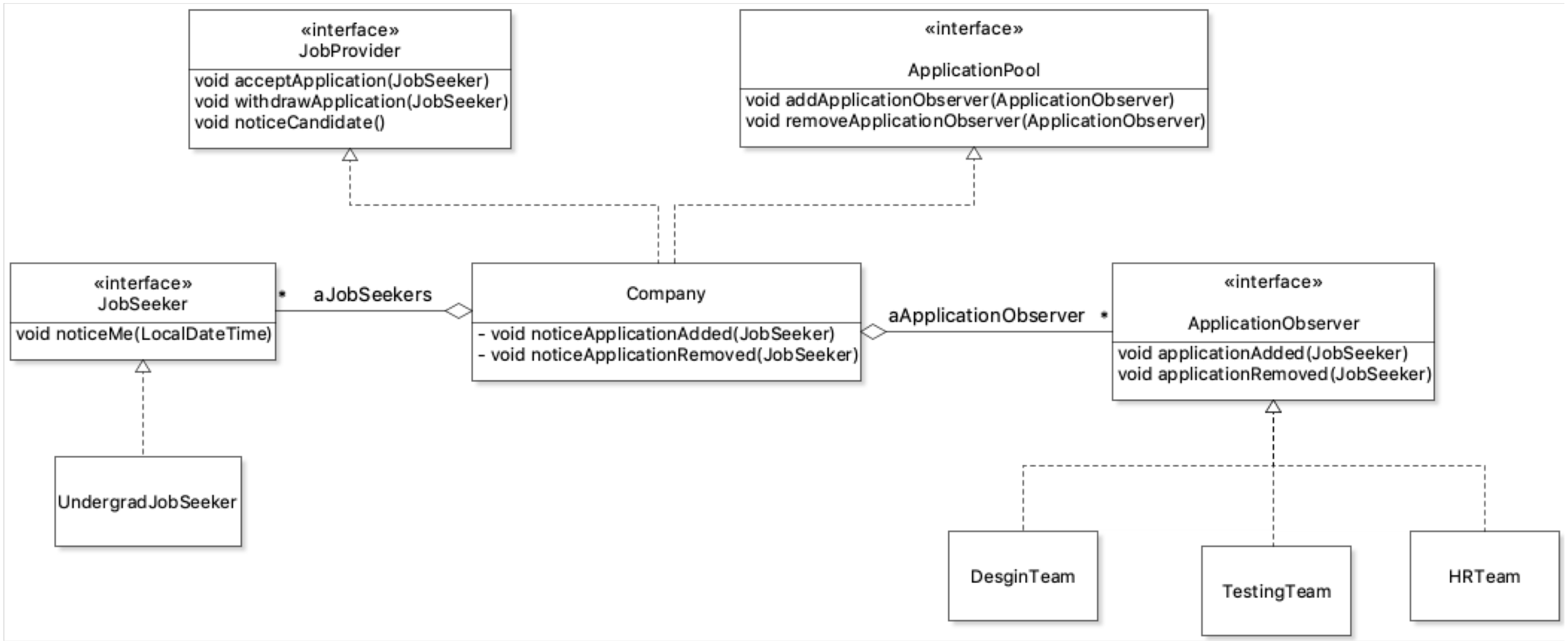
Data Flow Strategy?

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

Push model

`ApplicationPool` sends nothing but the most minimal notification, and `ApplicationObserver` ask for details explicitly thereafter.

Pull model

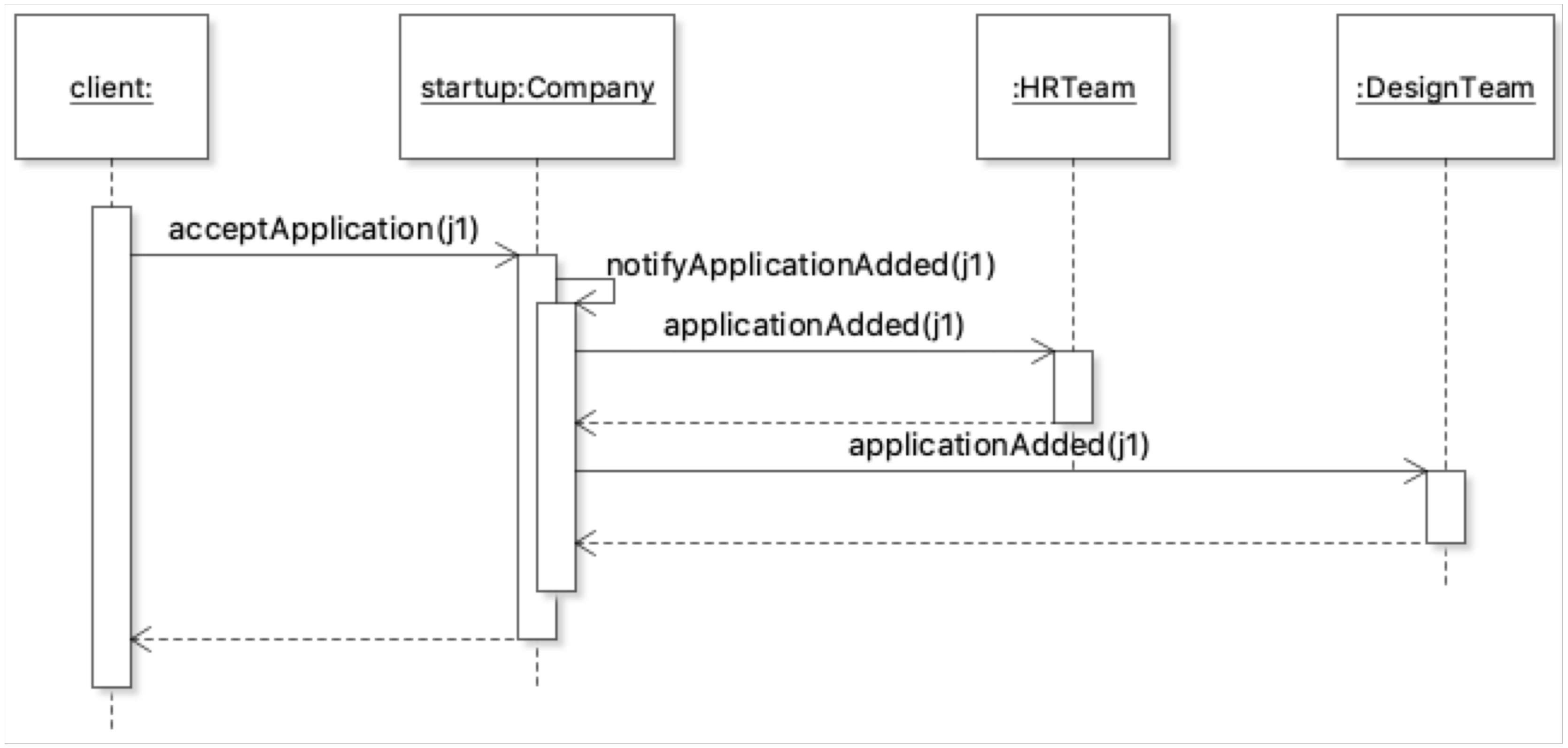

«interface»
JobProvider
void acceptApplication(JobSeeker)
void withdrawApplication(JobSeeker)
void noticeCandidate()
«interface»
ApplicationPool
void addApplicationObserver(ApplicationObserver)
void removeApplicationObserver(ApplicationObserver)
«interface»
JobSeeker
void noticeMe(LocalDateTime)
*
aJobSeekers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

Demo2

Acitivty2: 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)

Demo3 on Pull model