Dependency Injection Design Pattern

Saturday, December 12, 2020 11:23 PM

Dependency Injection Design Pattern

Usage Example 1: We avoid the anti-pattern of this design pattern.

In Talk.java: the constructors of Talk:

```
public class Talk extends Event {

    public Talk(String title, String roomID, String...
                int capacity, ArrayList<String> speakerIDs) {

        super(title, roomID, duration, restriction, capacity);
        setType("TALK");
        setSpeakerUserIDs(speakerIDs);
    }
```

Because each Talk has a list of speakers for it.
We stored an Arraylist of String representing the IDs of the speakers.
Therefore, in the constructor, we passed in as parameter a list of SpeakerID, instead of using new operator inside the Talk class to create an instance of the class Speaker, which cannot be used nor tested independently.

In lecture, we've learned that if we were to create an instance of Speaker inside a method in the class Talk, then, the Talk would be dependent on the Speaker class.
This kind of "hard dependency" is an anti-pattern which should be avoided, since usually we would like to use and test the class Speaker independently from the class Talk and avoid hard dependencies.

In order to eliminate the constructor call to create an instance of Speaker, we pass in the ID of the speaker we need directly as a parameter in the constructor of Talk, and in this way, we were able to assign Speakers to the talk without creating the speakers inside the constructor of Talk.
This loosens coupling between Speaker and Talk classes.

Note that strictly speaking, we could have created the Speakers outside of the class Talk, and pass a list of Speakers into Talk through a parameter, this fits the definition of Dependency Injection Design Pattern. However, in the case of this method, we make it simpler by just passing in the ID of the speaker, without creating an instance. Nevertheless, we have achieved the same goal with implementing this design pattern:
"The goal of Dependency Injection is always to eliminate unnecessary dependencies."
With this principle in mind, we successfully avoid "hard dependencies" in our project by never implementing the anti-pattern of the Dependency Injection design pattern.

Usage example 2: We implement this design pattern:

Inside gateway/FileReadWriter.java, we have several methods that implement the dependency injection design pattern:

For example, we have the following methods from line 54:

```
public void UserReader(AttendeeManager attendeeManager,
                       OrganizerManager organizerManager,
                       SpeakerManager speakerManager) {

    ArrayList<String> lines = new ArrayList();
    try {
        FileInputStream in = context.openFileInput("Users.txt");
        :
        // Reading in file contents
        :
        if (type.equals("ATTENDEE")) {
            attendeeManager.loadAttendee(username, email, ...);
            ...

        else if (type.equals("Organizer")) {
            organizerManager.loadOrganizer(username, .... );
        }
    }
}
```

In the above method, instead of calling constructors to creat three new objects of type AttendeeManager, OrganizerManager, and SpeakerManager, it takes the three object, which are created elsewhere, as parameters. This way, we have injected three instances of the usecase classes into the gateway class.

Therefore, by putting instances of the three manager classes as parameters, we implement the dependency injection design pattern. It solves the problem of too much coupling and as a result, we have less dependency between the gateway and usecase classes.

Another example: from line 118:

```
public void UserWriter(UserManager userManager) {
    List<String> userIDs = userManager.getUserIDs();
    // Writing to file
    ...
}
```

In this method, we inject an object of type UserManager as a parameter instead of calling the constructor of UserManager inside the method to create an object of UserManager.

We also used this design pattern when implementing the methods: // line 140

```
public void EventReader(EventManager eventManager,
                        RoomManager roomManager) {...}
// line 233
public void EventWriter(EventManager eventManager) {...}
// line 255
public void RoomReader(RoomManager roomManager) {...}
```

// line 283 RoomWriter
// line 301 ConversationReader
// line 361 ConversationWriter

By Jiayi Su, please feel free to contact me if you spot an error, or you think there is something to improve.

I am happy to improve it and make it better.
My email address is: sjy.su@mail.utoronto.ca