Lab 2: Implicit-invocation System

Sousa Discuss Feb 23, due March 8

This lab is to be done individually. If there is something you don't understand, or if you're stuck, ask the instructor for help.

Problem Description

The objective of this assignment is to better understand implicit-invocation architectures. Part of this assignment will be implementation-oriented, allowing you to experiment with a system that employs implicit invocation in order to gain a clearer understanding of the issues associated with carrying an architectural design through to code.

However, this is not a programming class, and so the emphasis of the assignment is on the architecture issues – this cannot be stressed enough. Simply re-writing the entire implementation demonstrates a lack of understanding of the core architectural concepts being presented in class. The assignment consists of two parts:

For the first part of the assignment you will be provided with a working system implemented in the implicit invocation paradigm. The base system for this assignment supports student course registration. Your responsibility in part one is to modify the existing system's source code according to the requirements below and to answer a few questions related to the modifications.

The second part of the assignment consists of analyzing the architecture of the system. After your analysis, modification, and reflection, you will answer several questions related to the design decisions you made in part one.

Functionality of the Current System

The basic function of the current system is to register students for courses. The system provides rudimentary functionality that supports student registration for courses and various queries, such as listing the courses a student has registered for. To achieve this functionality the system maintains two lists: (1) a list of students, and (2) a list of courses. A *Student* is an object in the implementation that maintains two internal lists: (1) a list of courses taken by the student, and (2) a list of courses the student has registered for. A *Course* is also represented as an object in the system that maintains an internal list of students that are registered for that course.

There are two sample input files provided with the base system; one file lists the students (Students.txt), and the other lists the scheduled courses (Courses.txt). The student file is similar to the one provided for Lab1. It now adds a column "Account Balance" that indicates how much money the student has available. The student file is field-oriented and space-separated. Each line of the file contains one student entry.

The fields are as follows:

G number	*Last/First Name*	*Program Affiliation*	*Account Balance*	*Course numbers that the student has completed (with no space between the prefix and the number)*
G00123456	Carson Kit	CS	3	CS112 CS211 CS332

Acknowledgment: Garlan & Lattanze

A second file provides scheduled course information. The course file is also field-oriented and space-separated. Each line of the file contains one course entry. The fields are as follows:

Course Number	*Section*	*Days*	*Start Time*	*End Time*	*Instructor*	*Course Title*
SWE443	A	T	430	700	Sousa	Software Architecture

The current system provides a rudimentary, text-based, menu-driven interface that offers a number of options to the user:

(1) List Students: Lists the students in the system. The students in the system are those read from the student information file, and a default one is provided (Students.txt).

(2) List Courses: Lists the courses in the system. The courses in the system are those read from the course information file, and a default one is provided (Courses.txt).

(3) List Students Registered for a Course: Prompts the user to enter a course ID and section number. The system lists the students registered for that course.

(4) List Courses a Student has Registered for: Prompts the user to enter a student ID. The system lists the courses that this student has registered for.

(5) List Courses a Student has Completed: Prompts the user to enter a student ID. The system lists the courses that this student has completed.

(6) Register a Student for a Course: Prompts the user to enter a student ID, a course ID, and a course section number. The system adds the course to the student's list of courses that he/she is registered for, and adds the student to the list of students registered for the course. Checks for duplication and schedule conflicts are performed before making an actual registration.

(X) Exit: Ends the program execution.

Architecture of the Current System

The existing system has an implicit-invocation architecture whose components are implemented as objects. An architecture diagram follows:

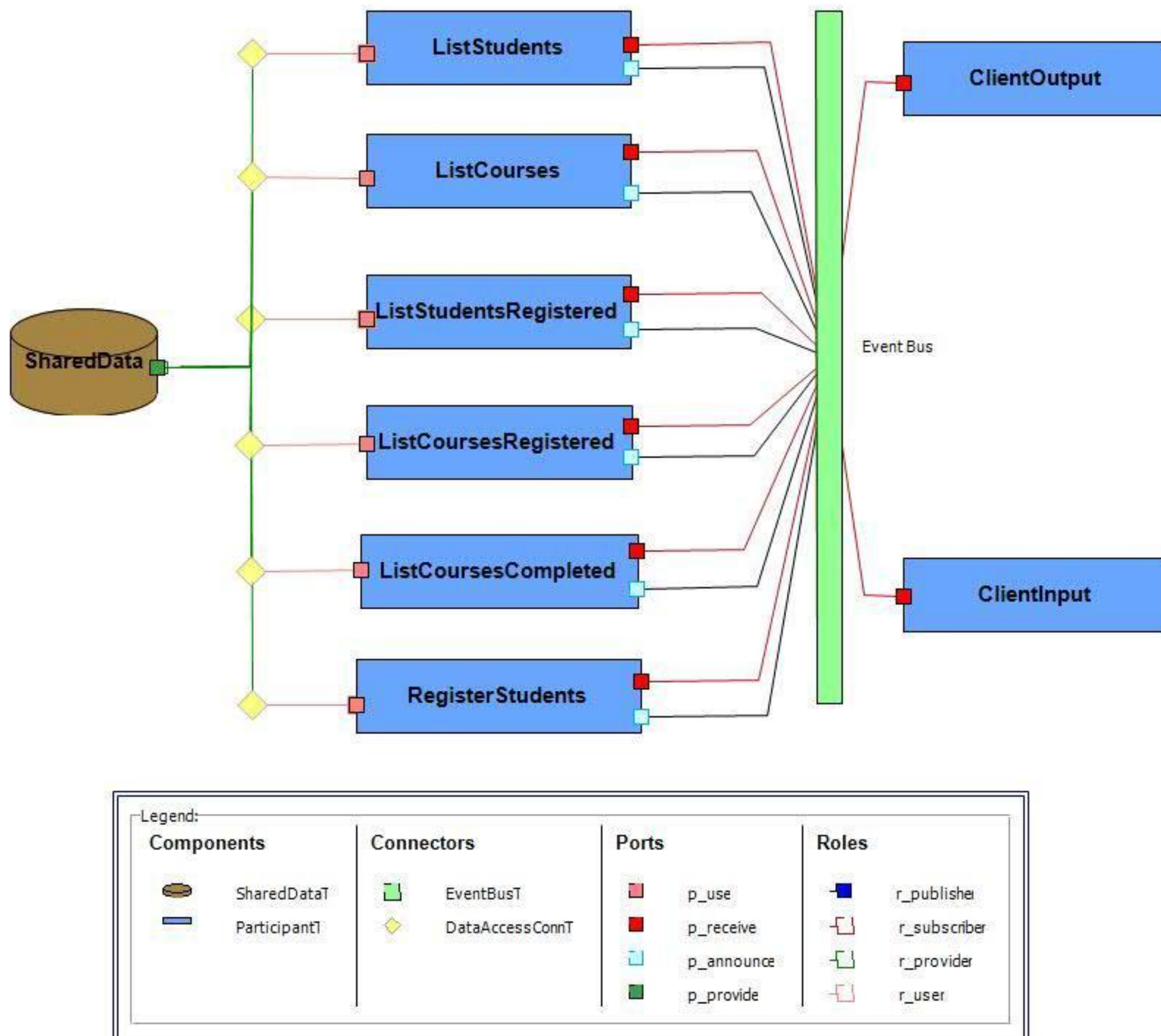

Figure 1: Architecture of the Current System

System functionality is partitioned and encapsulated within each component. Components broadcast to the event bus to request services, or listen on the bus to provide services to other components. Notice that some components only send notifications (announce) to the bus, some only receive (listen), and others do both. A shared data component stores shared state, such as students and courses. It is accessed by certain components through point-to-point data access connectors. (There are six such connectors in the provided system.)

The publish-subscribe implementation is accomplished through the use of Java *Observer* and *Observable* classes. The system is initialized by the class, SystemMain. The current system contains the following files:

SystemMain.java: Has the main() method and creates the system structure by instantiating all of the components and starting the ClientInput component.

ClientInput.java: Presents the main menu and broadcasts service requests to the other components based on user input.

ClientOutput.java: Subscribes to and receives "show" notification. The contents of these notifications are displayed onto the user console.

DataBase.java: Provides access to the student and course lists. Also provides methods for student registration.

EventBus.java: The implicit invocation architecture is implemented using Java Observer/ Observable classes. This class provides the basis for the components to be observers and to announce notifications.

CommandSet.java: Provides services to list student/course information and to register a student for a course.

***Handler.java:** Implementation of a component that handles a menu event.

Student.java: Class used to represent a Student in the system.

Course.java: Class used to represent a Course in the system.

Courses.txt: Text file that contains list of courses.

Students.txt: Text file that contains list of students.

Compiling and running the current system

NOTE: These instructions assume that you have already installed J2SE 1.6 or higher.

First, unzip (using WinZip, for example) the Lab2 - Java.zip file into a working directory. In order to compile the system, open a command prompt window (or start a Linux terminal), change the working directory to the directory containing all of the system's source files, and type the following:

```
> javac *.java
```

The compile command above creates the class files. After that, you can run the system by typing:

```
> java SystemMain Students.txt Courses.txt
```

Part 1: Modifications to the Current System

Your task is to modify the current system to support new modifications described below. Include all of the following modifications in a single new system. Make sure that you use good programming practices, including comments.

A hint concerning your modifications:

Whenever possible, keep your changes to the system within the implicit invocation pattern. Modifications like adding new events, adding new components, and changing the events that a component listens to or generates will keep you within the pattern. Modifications to the infrastructure, adding new connector types, or modifications to existing components run the risk of your new system not being implicit invocation. Remember that you can implement one pattern with the infrastructure from another, such as implementing a call-return pattern using events. With each of your modifications, consider whether you are changing the fundamental system "within the spirit" of implicit invocation. If not, explain deviations, why you made them, and the consequences of the choices you made.

Required Modifications

A. Augment the system to support logging by adding a new component. All output that goes to the screen should also be written to a log file. (Note: There are many ways to implement this. Which way requires the least coding? Which way matches best with this pattern of implicit invocation architecture?)

B. Suppose we want to know when a course becomes overbooked. Add new capability to the system so that it announces when a class is overbooked. Note that it does not need to prohibit registrations for overbooked classes, but simply announce that fact. For the purposes of this assignment, consider that a class is overbooked when more than three students are registered. (“30” is more realistic, but “3” makes testing easier.)

C. Extract the course-conflict checking from the RegisterStudentHandler and put it into its own component. The observable system behavior should not change. (By "observable" we mean to the user of the system, not to someone viewing the architecture or the source code.)

D. Create a billing component that charges a student when registering for a course, subject to the following constraints. (1) If the student’s account doesn’t have enough money, they should not be allowed to register. (2) If the registration fails due to time conflicts, the account should not be charged. You may set the tuition to any positive amount and keep track of the account balances however you like. Assume that it is possible to cancel a registration or a charge if a problem occurs.

Acknowledgment: Garlan & Lattanze

Write-Up

After completing the modifications, provide a short write-up that includes the following:

1. Produce architectural descriptions (i.e., diagrams) of the modified system. Create a view of the system that depicts a runtime perspective of the system. Create another view that depicts the static code structures of the system. Describe what kind of reasoning each view supports. Include a textual explanation for the different elements in each view of the architecture. Include any other supporting views you think necessary to ensure that the description of the modified system is clear and unambiguous. Include whatever prose you believes is necessary to accompany the views.
2. For the new system, associate the architectural elements (i.e., components, connectors, ports, roles, bindings, hierarchies) with the implementation elements (i.e., variables, methods, classes, files). For the elements you cannot find clear mappings, does that mean something was wrong in the architecture or implementation? Discuss why such mismatches can happen.
3. Discuss any deviations from a pure implicit-invocation pattern in the original and modified systems, both in the architectural design and the implementation of it.
4. Based on your experience so far, what kinds of changes are (1) easy or (2) hard to make in implicit-invocation architectures relative to the other patterns/styles you have learned about this semester. Explain why; use examples to compare and contrast in your discussion.
5. Briefly describe the changes to the modified system that would be necessary to transform it into a distributed, implicit event system where the elements need not necessarily reside on the same platform. How difficult would this be? Would the implicit invocation pattern be violated?

Provide a description of how to set-up and execute your modified systems. If we are unable to follow your instructions (and/or they are incorrect) you will be penalized or receive a zero for the programming portion of the assignment.

Submission Guidelines

- Your implementation must be delivered in a zip file following the following format LAB2-*your last name*.zip.

- The zip must contain
    - the modified source code,
    - a README file describing how to install and run your implementation,

- **Submit the zip file above and the *write-up* on Blackboard by the due date.**

Grading Criteria

Your solutions and commentary will be graded using the following criteria:

- The quality and contents of your write-up, indicating your understanding through discussion of the implications of changes to the system architecture. Be clear, concise, and complete. Be sure to answer all parts of the questions. Note that grammar and spelling count!
- Consistency between the architectural descriptions and the design claimed in the source files (i.e., can we map your architectural design to code elements).
- The degree to which it respects the implicit invocation architectural pattern as much as possible and deviations are clearly explained.
- Professionalism: presentation, timely submission, etc.

Also, make sure that:

- The resulting system performs as required. We will test your solution on our own test data.
- The modified code is clean, clear, documented, and of high quality.

Each question will be weighted as follows (100 points maximum):

Part 1. Implementation

- modification A: 5 points
- modification B: 10 points
- modification C: 10 points
- modification D: 15 points

Part 2. Write-up

- Question 1: 20 points
- Question 2: 10 points
- Question 3: 10 points
- Question 4: 10 points
- Question 5: 10 points