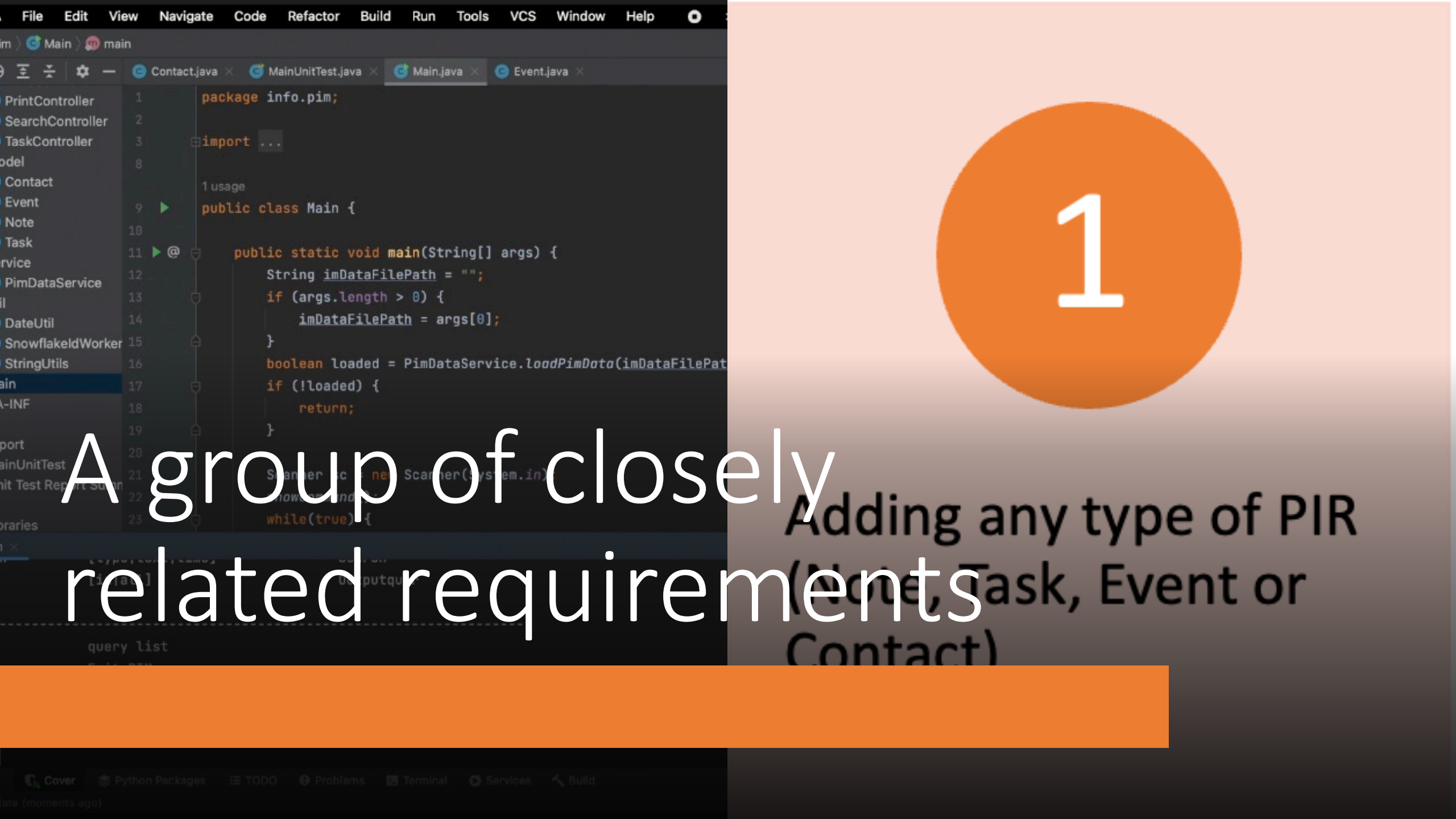
COMP 3211 Group 35

CHEN Ziyang, Rocky 21095751d

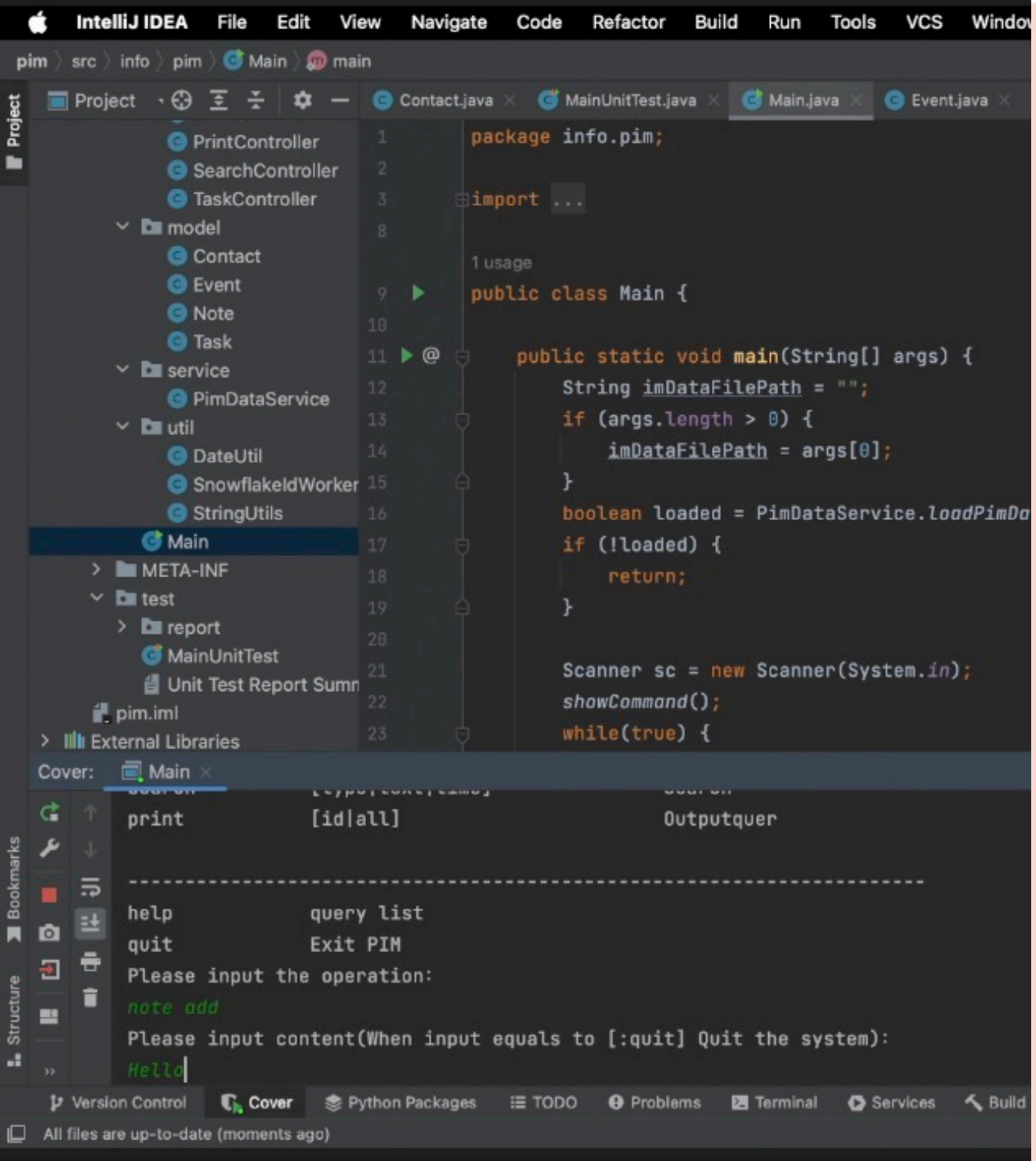
LI Shuhang, Hubery 21102658d

YE Chenwei, Jesse 21103853d

HE Rong, Shawn 21101622d



A group of closely related requirements

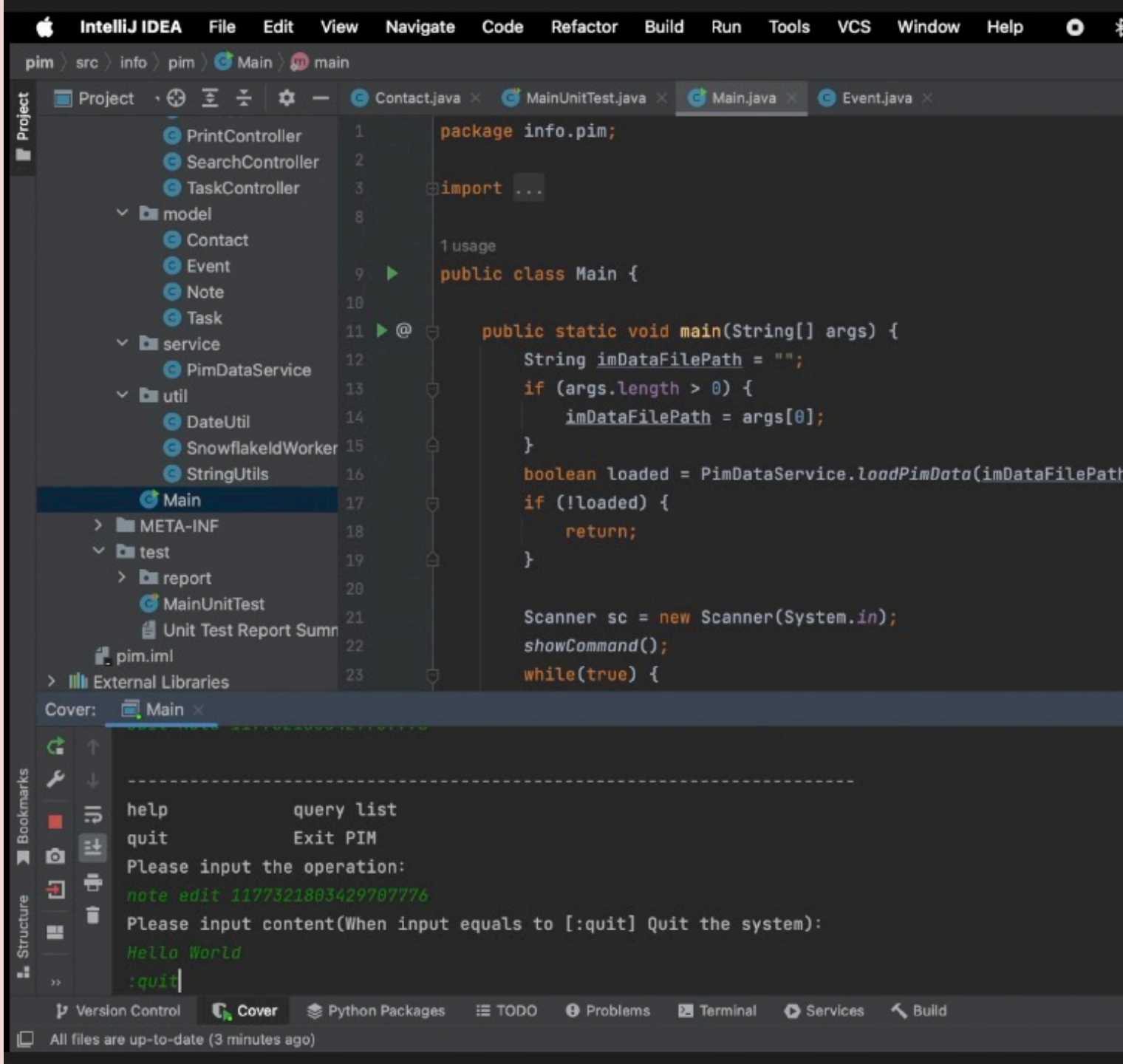


1

Adding any type of PIR (Note, Task, Event or Contact)

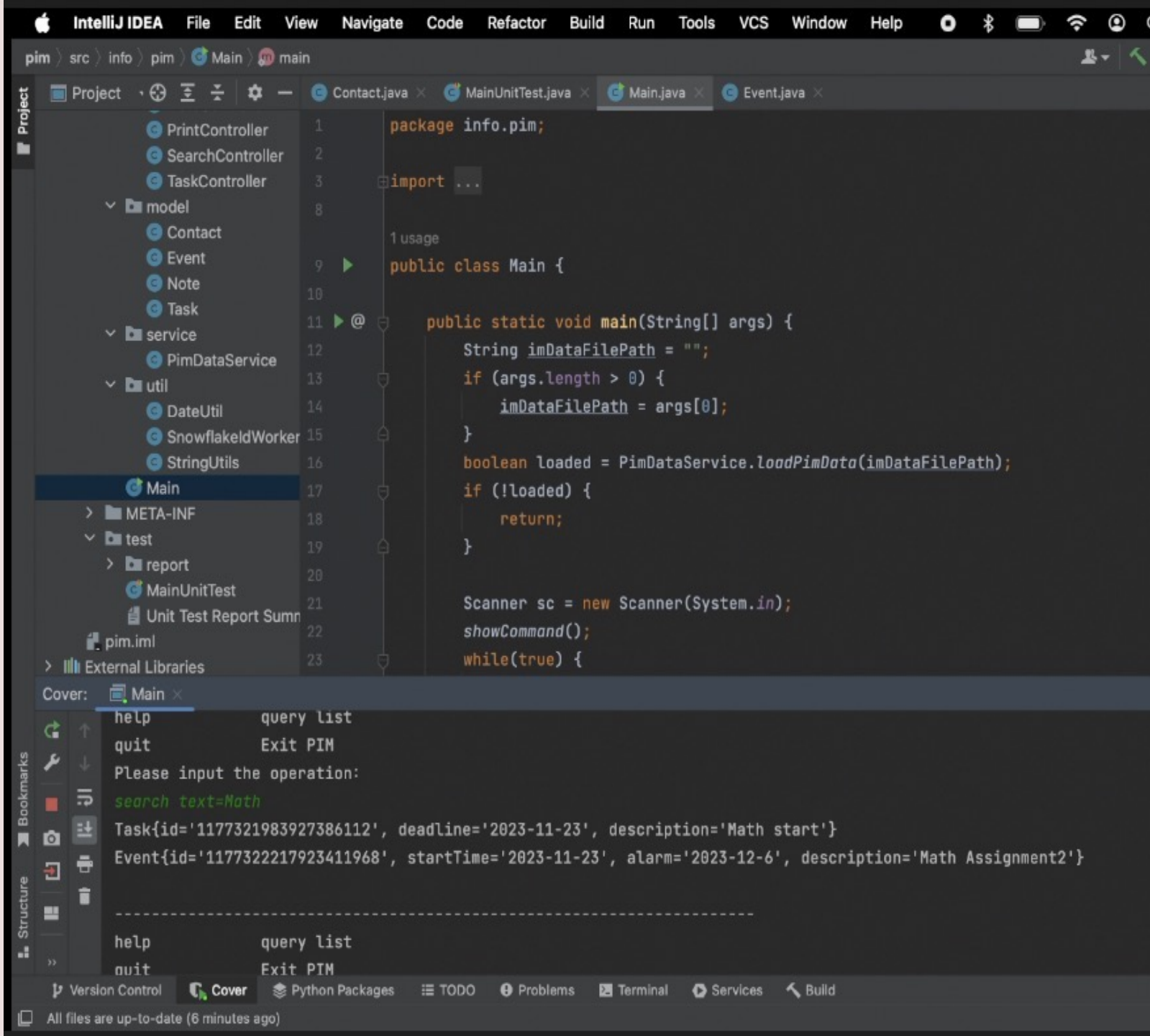
2

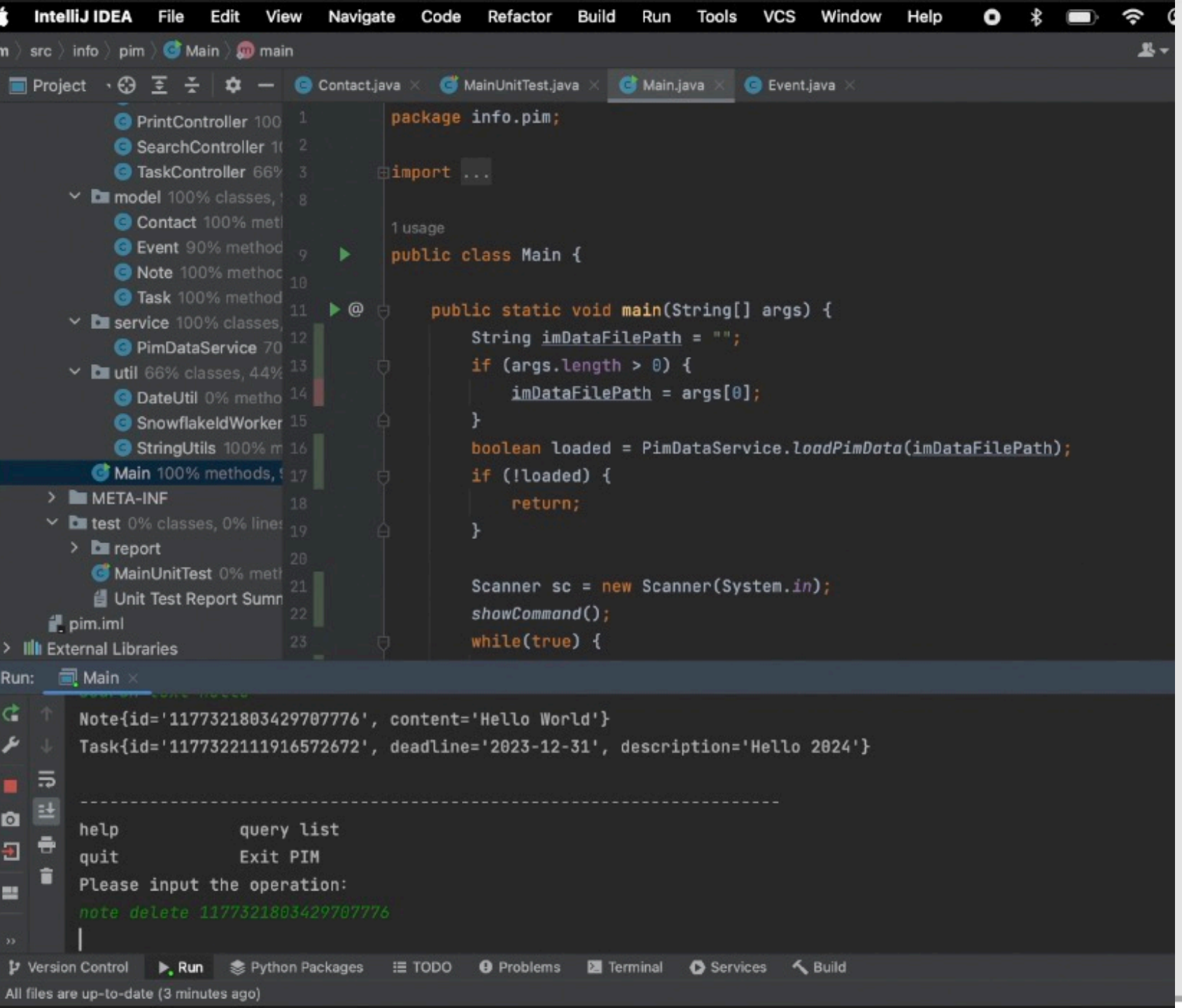
Modifying any existing PIR



3

Searching by type, text, and time(Allow users to create compound search criteria by using &&, || or !)



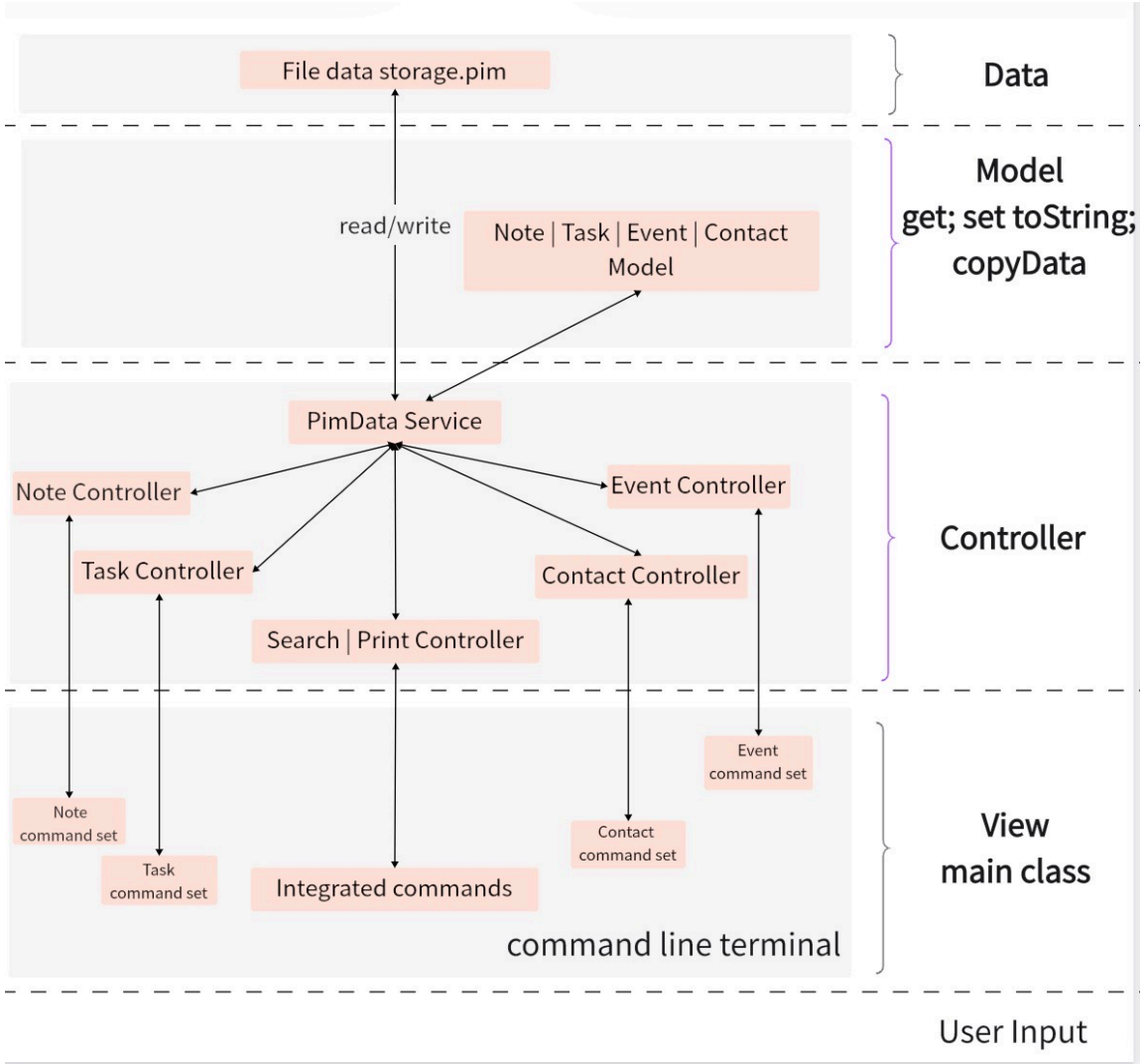


4

Deleting PIR from PIM

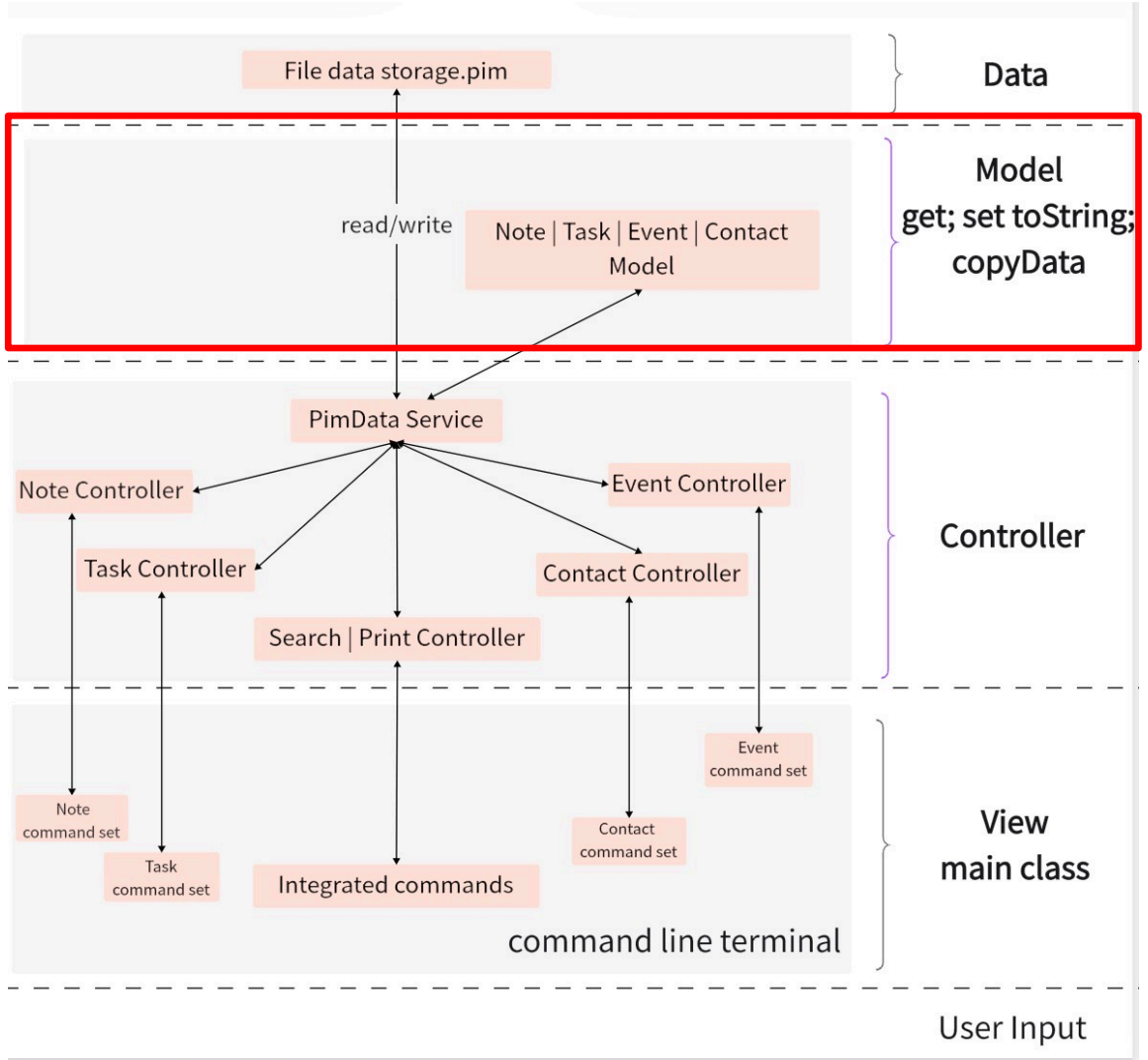
The design to support the group of requirements

- We adopt **the Model-View-Controller architecture pattern.**



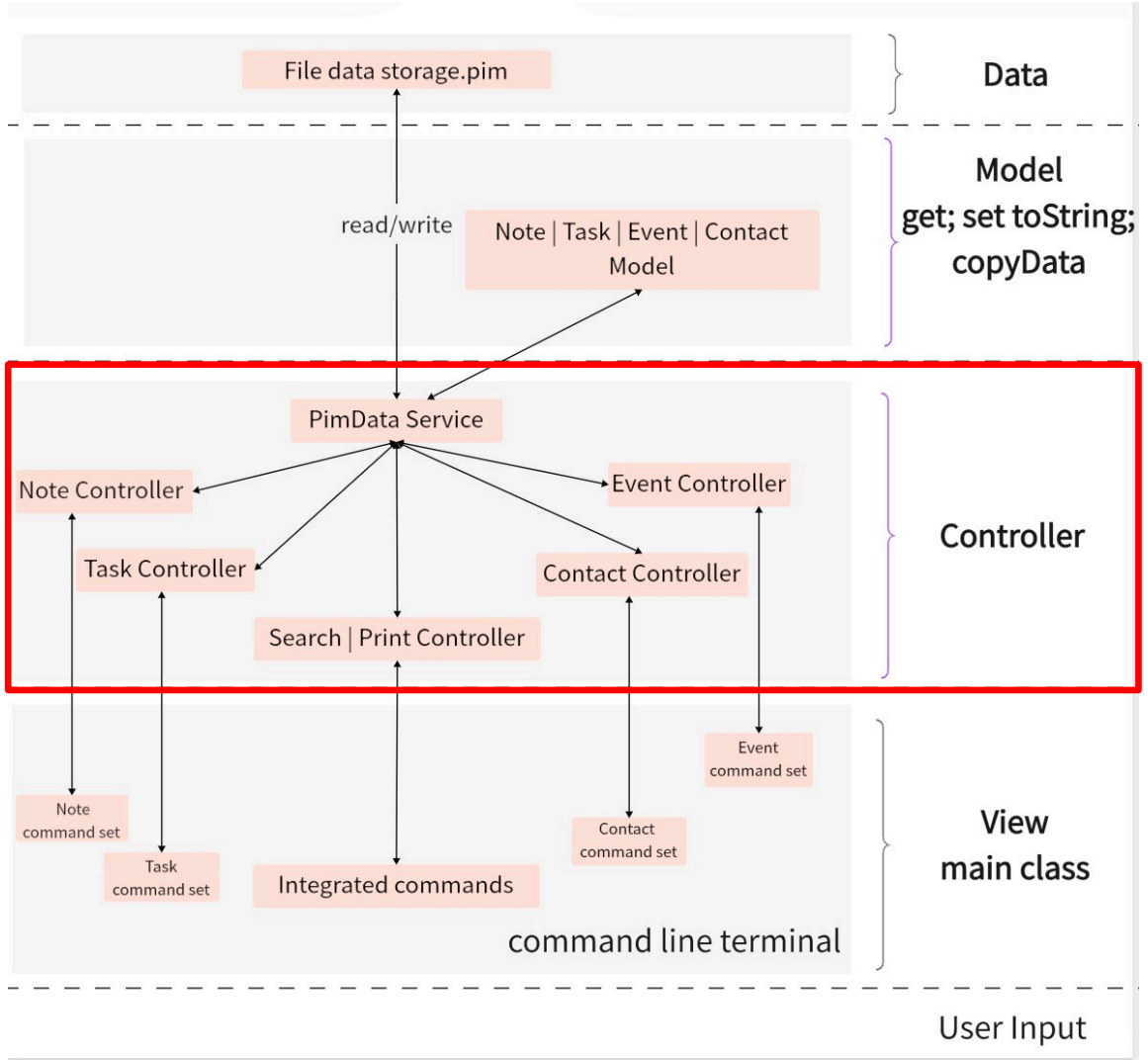
The design to support the group of requirements

- We adopt **the Model-View-Controller architecture pattern.**
- **Model:** Implements entity abstraction of data. The four model classes Note, Task, Event and Contact are defined in the info.pim.model package.



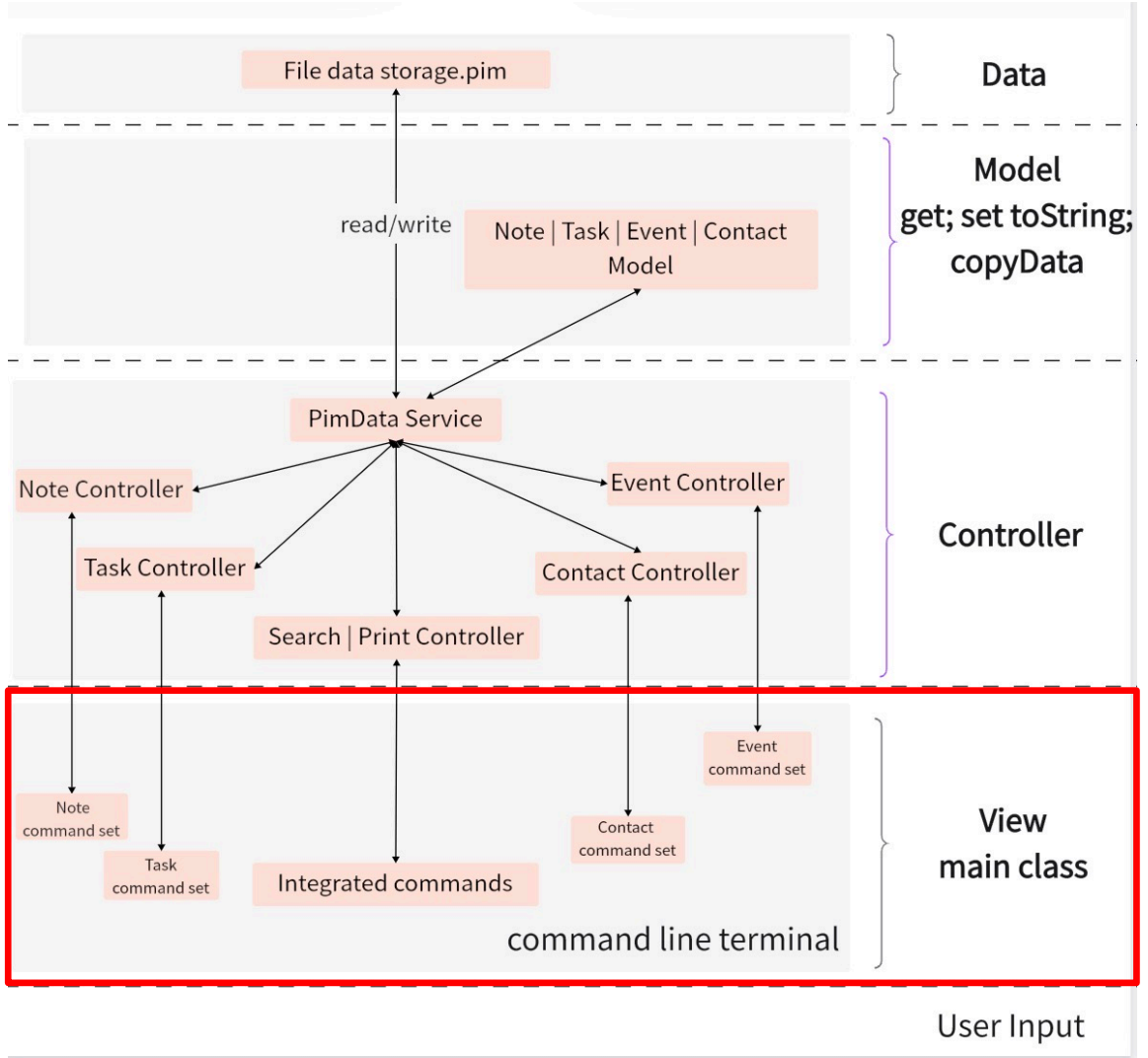
The design to support the group of requirements

- We adopt **the Model-View-Controller architecture pattern.**
- **Model:** Implements entity abstraction of data. The four model classes Note, Task, Event and Contact are defined in the info.pim.model package.
- **Controller:** includes control classes and service classes. The control class accepts instructions from the view layer, performs some logical processing, and completes data access and update operations through the service class. Four controller classes are defined in the info.pim.controller package: NoteController, TaskController, EventController and ContactController. They process commands entered from the command line and update the model accordingly.

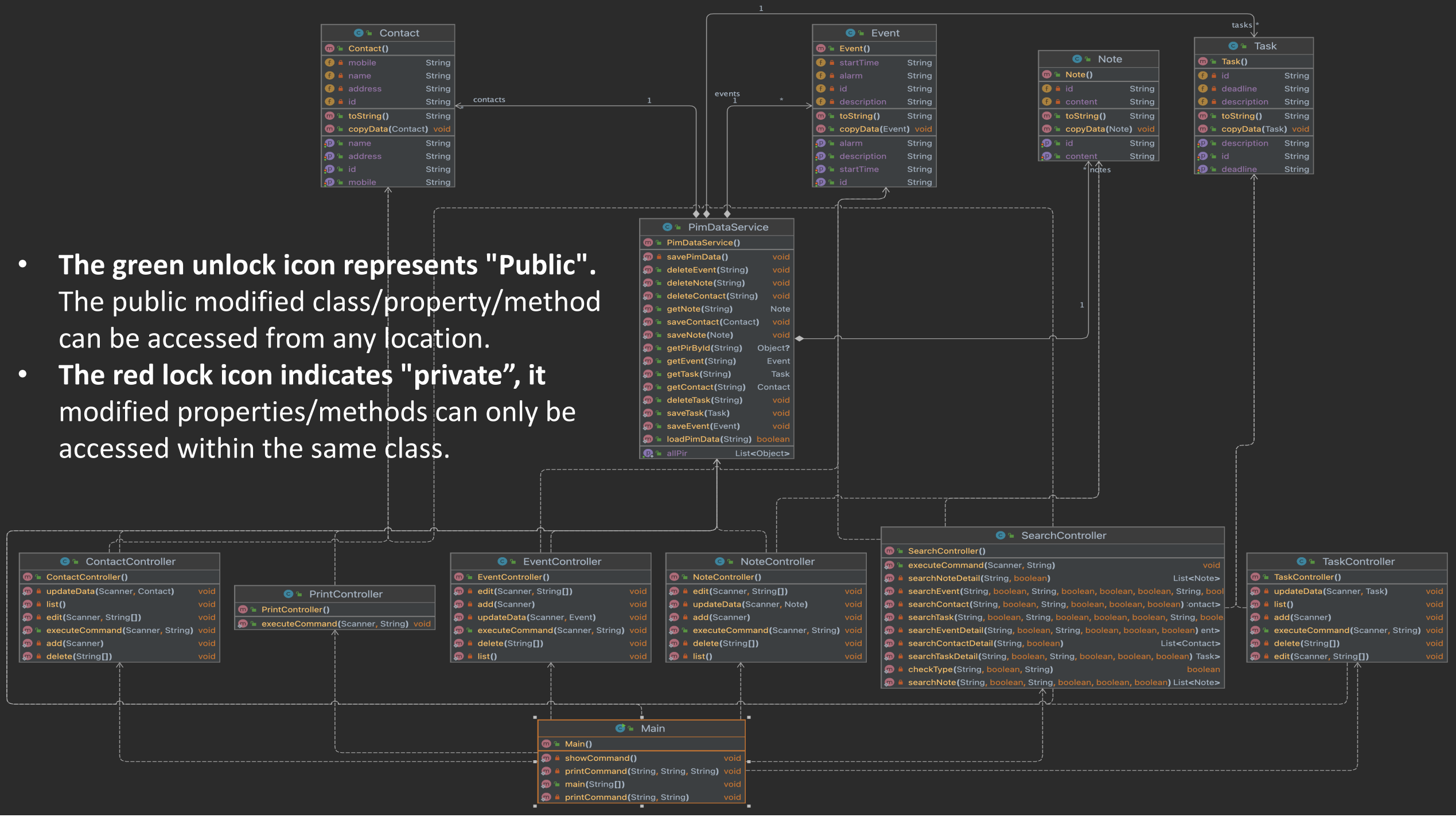


The design to support the group of requirements

- We adopt **the Model-View-Controller architecture pattern.**
- **Model:** Implements entity abstraction of data. The four model classes Note, Task, Event and Contact are defined in the info.pim.model package.
- **Controller:** includes control classes and service classes. The control class accepts instructions from the view layer, performs some logical processing, and completes data access and update operations through the service class. Four controller classes are defined in the info.pim.controller package: NoteController, TaskController, EventController and ContactController. They process commands entered from the command line and update the model accordingly.
- **View (main):** View is the command line interface that interacts with the user, receives system instructions input by the user, and returns the results of the instruction operations to the command line interface.

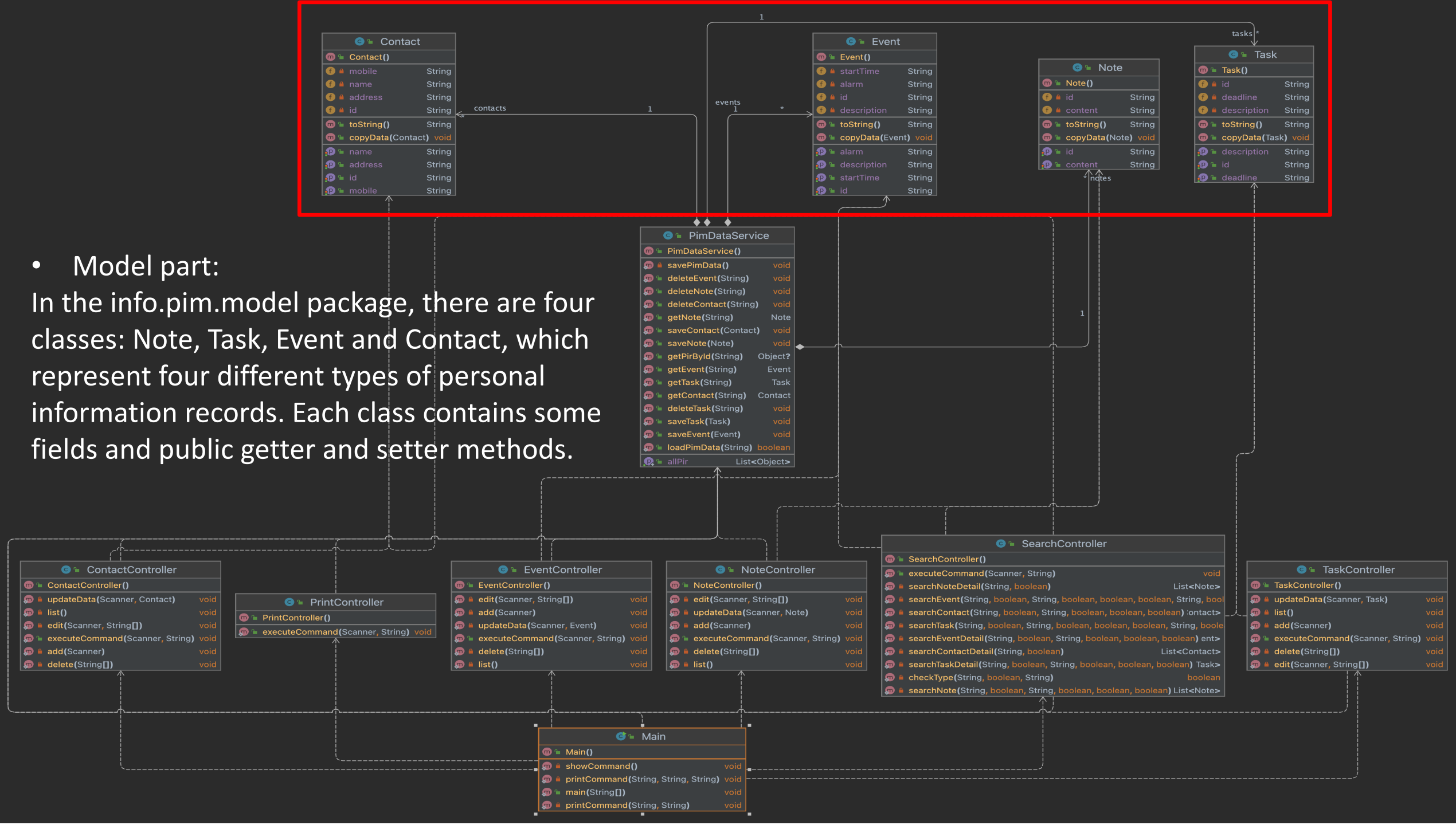


- **The green unlock icon represents "Public".** The public modified class/property/method can be accessed from any location.
- **The red lock icon indicates "private", it** modified properties/methods can only be accessed within the same class.



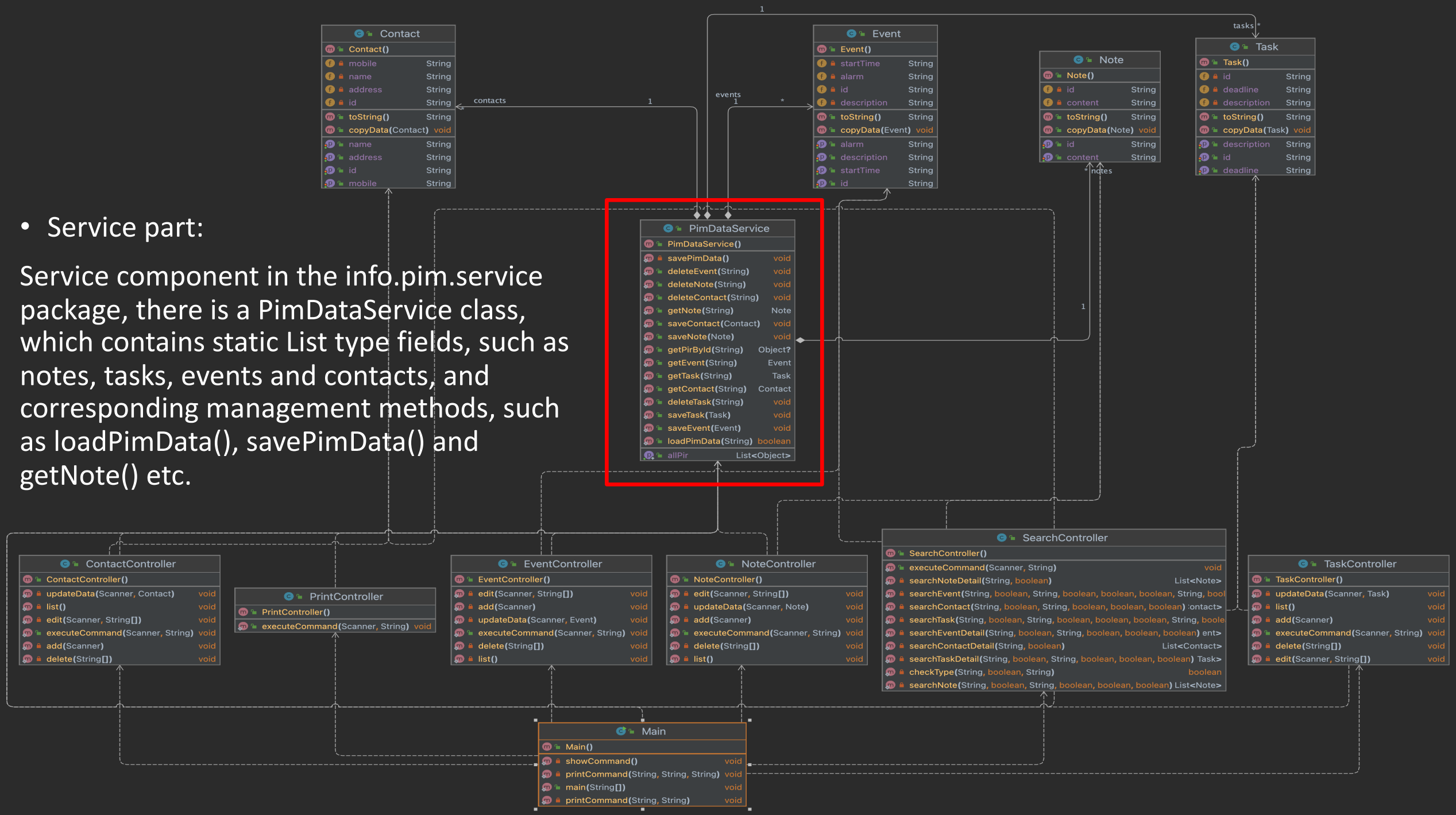
- Model part:

In the info.pim.model package, there are four classes: Note, Task, Event and Contact, which represent four different types of personal information records. Each class contains some fields and public getter and setter methods.



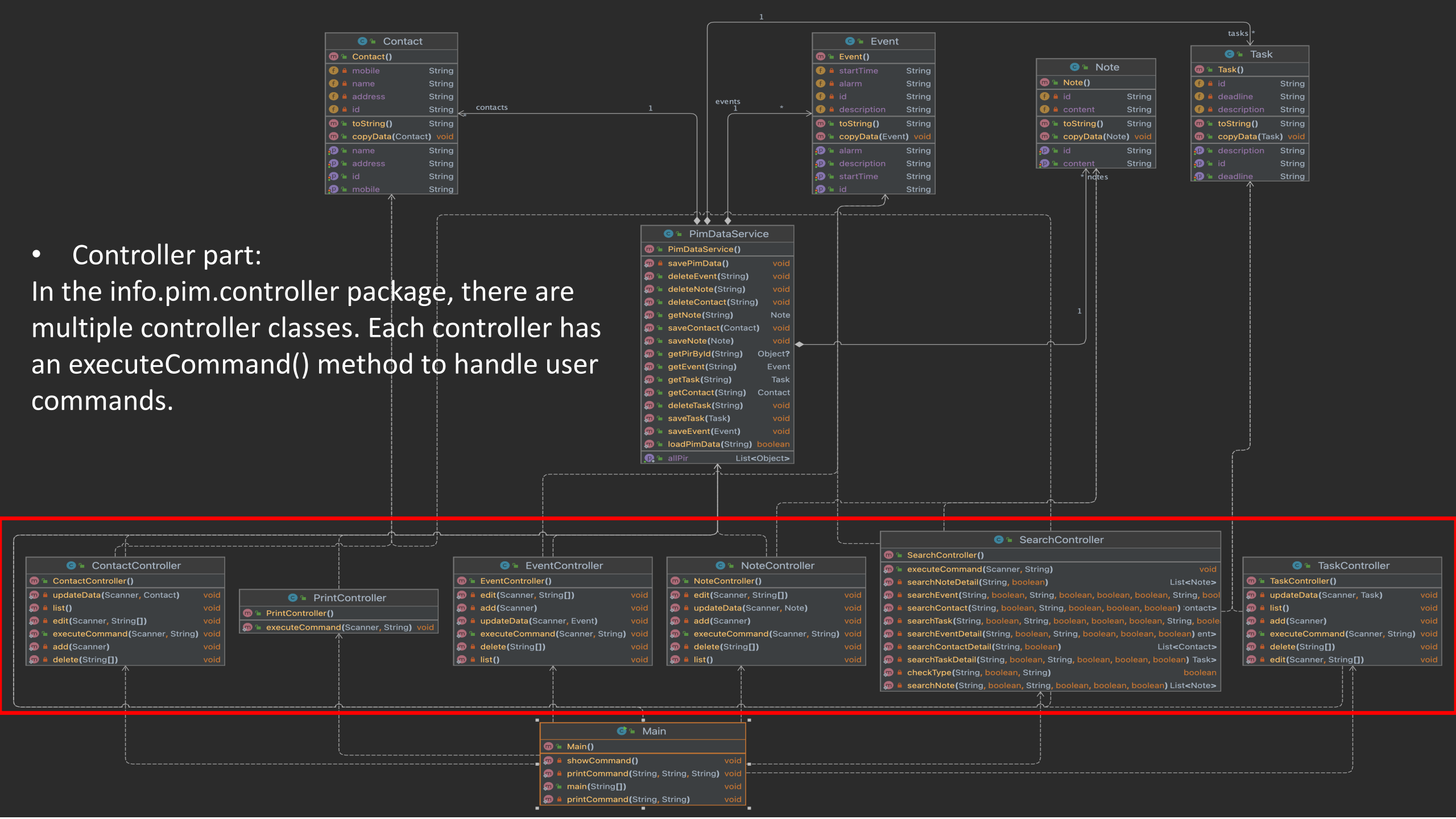
- Service part:

Service component in the info.pim.service package, there is a PimDataService class, which contains static List type fields, such as notes, tasks, events and contacts, and corresponding management methods, such as loadPimData(), savePimData() and getNote() etc.



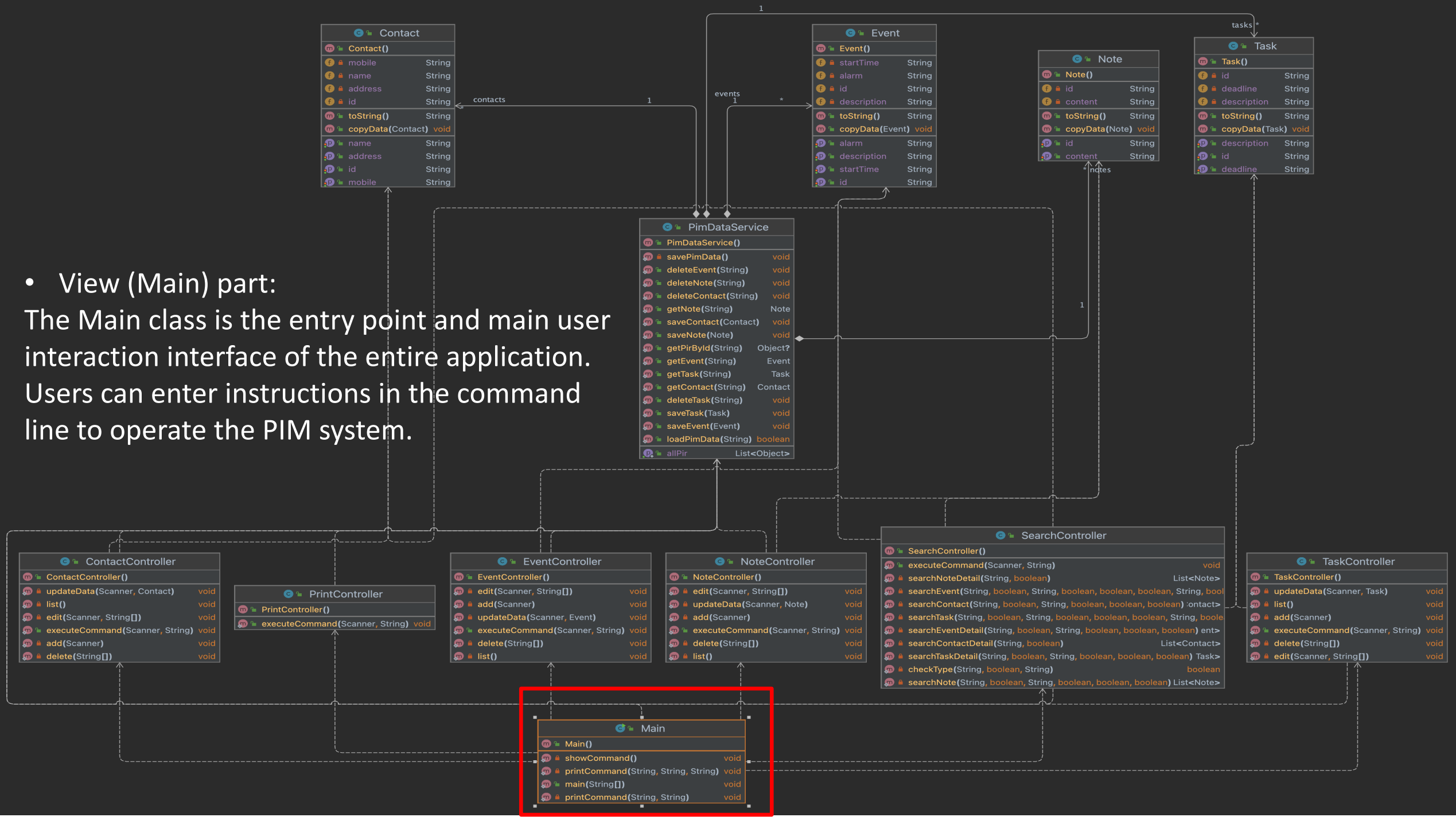
- Controller part:

In the info.pim.controller package, there are multiple controller classes. Each controller has an executeCommand() method to handle user commands.



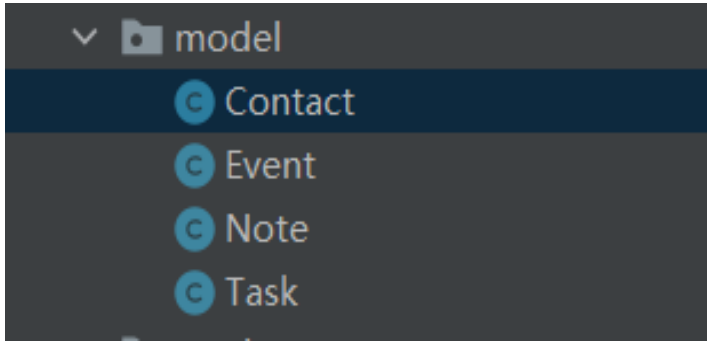
- View (Main) part:

The Main class is the entry point and main user interaction interface of the entire application. Users can enter instructions in the command line to operate the PIM system.



The strategy adopted when preparing high-quality unit tests for the relevant model code

The model contains four types of PIRs



The code for each type of PIR mainly consists of three parts: get(), set(), toString()

```
public String getId() { return id; }

public void setId(String id) { this.id = id; }

public String getName() { return name; }

public void setName(String name) { this.name = name; }

3 个用法
public String getAddress() { return address; }

3 个用法
public void setAddress(String address) { this.address = address; }

3 个用法
public String getMobile() { return mobile; }

3 个用法
public void setMobile(String mobile) { this.mobile = mobile; }
```

```
    @Override
    public String toString() {
        return "Contact{" +
                "id='" + id + '\'' +
                ", name='" + name + '\'' +
                ", address='" + address + '\'' +
                ", mobile='" + mobile + '\'' +
                '}';
    }
}
```

- Comprehensiveness: Tests should cover all major functionality.

- Isolation: Each unit test should be independent of other tests.

- optimize test execution time

```
public void testTaskAdd() {
    Task task = new Task();
    String id = PimDataService.idWorker.nextId();
    task.setId(id);
    task.setDeadline(DateUtil.date());
    task.setDescription("description add");
    // add
    PimDataService.saveTask(task);

    // get add data
    Task taskSaved = PimDataService.getTask(id);
    assertSame(task, taskSaved);

    // test print data
    String dataStr = "Task{" +
            "id='" + id + '\'' +
            ", deadline='" + taskSaved.getDeadline() + '\'
            ", description='" + taskSaved.getDescription()
            '}';
    assertEquals(task.toString(), dataStr);
}
```

Thanks for listening!