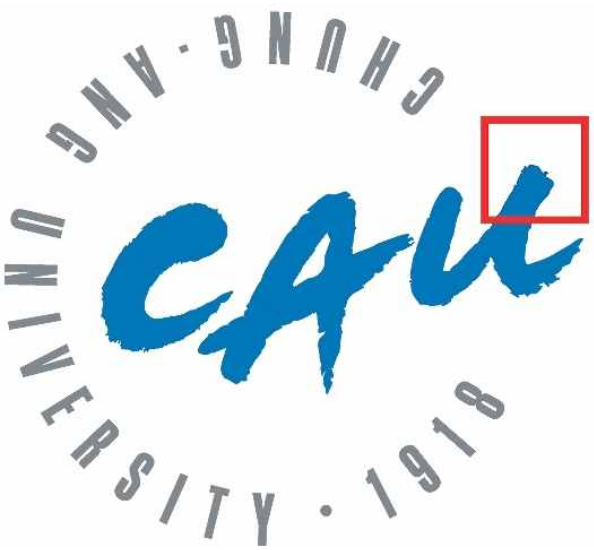
Book Transaction System Project

-Documentation-



컴퓨터공학과

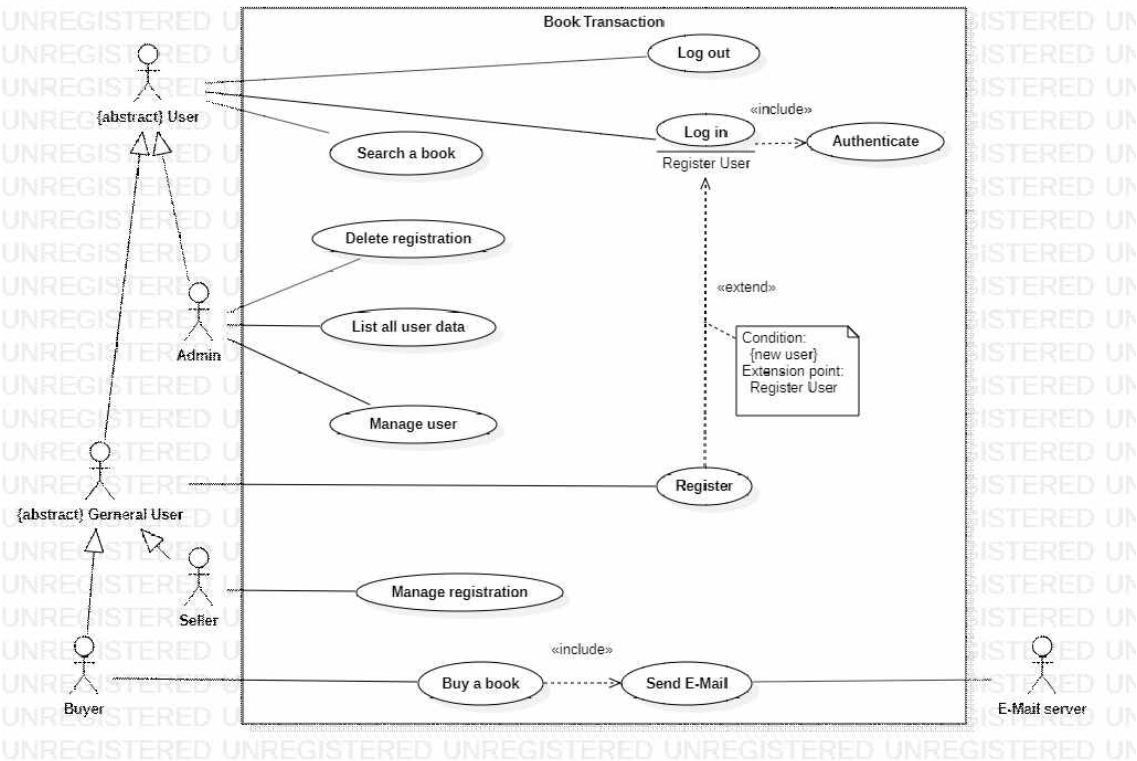
20160971 이정재

• Actor-Goal List

Actor	Goal
buyer	• register • login • logout • search a book • buy a book
seller	• register • login • logout • search book • register a book • modify registered book • delete registered book

Actor	Goal
admin	• login • logout • search a book • delete registered book • list all user data • deactivate user • activate user • delete user
E-Mail Server	• send an E-mail

• Use Case Diagram



• Actor

- User: Admin을 포함한 모든 System 사용자의 역할을 나타내는 Actor입니다.
- General User: Admin을 제외한 System을 이용해 헌책을 사고파는 사용자의 역할을 나타내는 Actor입니다.
- Admin: System 상에 올라온 거래와 General User를 관리하는 역할을 나타내는 Actor입니다.

- Buyer: System을 이용해 헌 책을 구입하는 역할을 나타내는 Actor입니다.
- Seller: System을 이용해 헌 책을 판매하는 역할을 나타내는 Actor입니다.
General User는 Buyer와 Seller의 역할을 동시에 수행할 수 있습니다.
- E-Mail server: 판매가 이루러졌을 때 Buyer와 Seller에게 각각 메일을 보내는 역할을 나타내는 Actor입니다.

• **Use Case**

- search a book은 모든 사용자가 이용할 수 있는 기능이므로 전체 사용자 역할인 User actor에 연결했습니다.
- login 할 때 새로운 user라면 register use case도 수행되도록 extend 했습니다. 이때 register use case는 Admin actor를 제외한 General user actor에서만 수행되도록 했습니다.
- login 시 deactivated user는 아닌지, 기존에 등록된 user인지 확인해야 하므로 authenticate use case를 include 했습니다.
- Admin actor에서 activate user, deactivate user, delete user는 manage user라는 하나의 공통된 목적을 갖고 있으므로 하나의 use case로 묶었습니다.
- Seller actor에서 register book, modify book, delete book은 manage registration라는 하나의 공통된 목적을 갖고 있으므로 하나의 use case로 묶었습니다.
- Buyer actor가 buy a book use case를 수행하면 Buyer actor와 Seller actor에게 email을 보내는 send email use case를 include했습니다.

• Use Case Text

UC1: Buy Book

• Scope: Book transaction application
• Level: User goal
• Primary Actor: Buyer
• Stakeholders and Interests:
  - Buyer: Wants to buy a book.
  - Seller: Wants to sell a book.
  - E-Mail server: Wants to receive purchasing log.

• Preconditions: Buyer has logged in to the system.
• Success Guarantee: Purchasing log is saved. Send purchasing log to both buyer and seller.

• Main Success Scenario
1. User searches a book to buy.
(can search book with the title, ISBN number, name of the author, id of the seller)
2. System displays a list of searched book.
3. User buys a book.

4. System logs purchase and sends it to E-mail server.
5. E-mail server sends E-mails to buyer and seller that contains purchasing log information.
6. User quits buying books.

• Extension
*a. At any time, user quits buying books.
*b. At any time, system fails.
*c. At any time, admin deactivates user.
  1. System notices user.
  2. User logs out from the system.

1a. Book not found.
  1. System displays that there's no such book.
  2. User probably search another book (Main Success Scenario #1).

3a. Book sold out or Admin/Seller deletes registration or Admin deactivates Seller.
  1. System notice user that the book is not sellable.
  2. User may try another one (Main Success Scenario #1 or #3).

UC2: Register Book

• Scope: Book transaction application
• Level: User goal
• Primary Actor: Seller
• Stakeholders and Interests:
  - Seller: Wants to register a book.

• Preconditions: Seller has logged in to the system.
• Success Guarantee: Update registered information on system.

• Main Success Scenario
1. User starts registration.
2. User register the book on system.
3. System records information of the book.
4. User quits registration.

• Extension
*a. At any time, user quits registration.
*b. At any time, system fails.
*c. At any time, admin deactivates user.
  1. System notices user.

2. User logs out from the system.

2a. Needed information missing or Wrong input.
1. System gives error message.
2. User may tries again (Main Success Scenario #2).

UC3: Delete User

- Scope: Book transaction application
- Level: User goal
- Primary Actor: Admin
- Stakeholders and Interests:
  - Admin: Wants to delete entire user data including registered book data.

- Preconditions: Admin has logged in to the system.
- Success Guarantee: Delete user data. Delete book data he/she registered.

- Main Success Scenario

1. Admin lists all user data.
2. Admin deletes user.
3. System deletes user data and user's registration data.
4. Admin quits deleting user data.

- Extension

1a. User not found
1. System displays that there's no user.

2a. User is in activated status.
1. System displays error.
2. Do UC: Deactivate User.

2-3a. System fails.
To support recovery and correct deletion of user's entire registration, ensure that all sensitive state be saved before system down.

1. Admin restarts System, logs in.
2. System recovers prior state and resumes deletion.

UC4: Delete Registered Book

- Scope: Book transaction application
- Level: User goal
- Primary Actor: Admin

• Stakeholders and Interests:
  - Admin: Wants to delete registration.

• Preconditions: Admin has logged in to the system.
• Success Guarantee: System updates registration data.

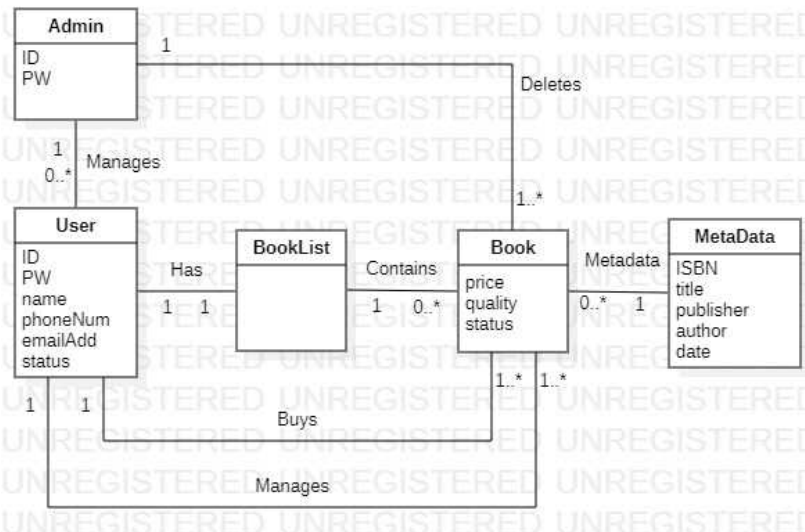
• Main Success Scenario
1. Admin searches a book.
2. Admin deletes a book.
3. System updates registration data.
4. Admin quits deleting registration.

• Extension
*a. Admin quits deleting.

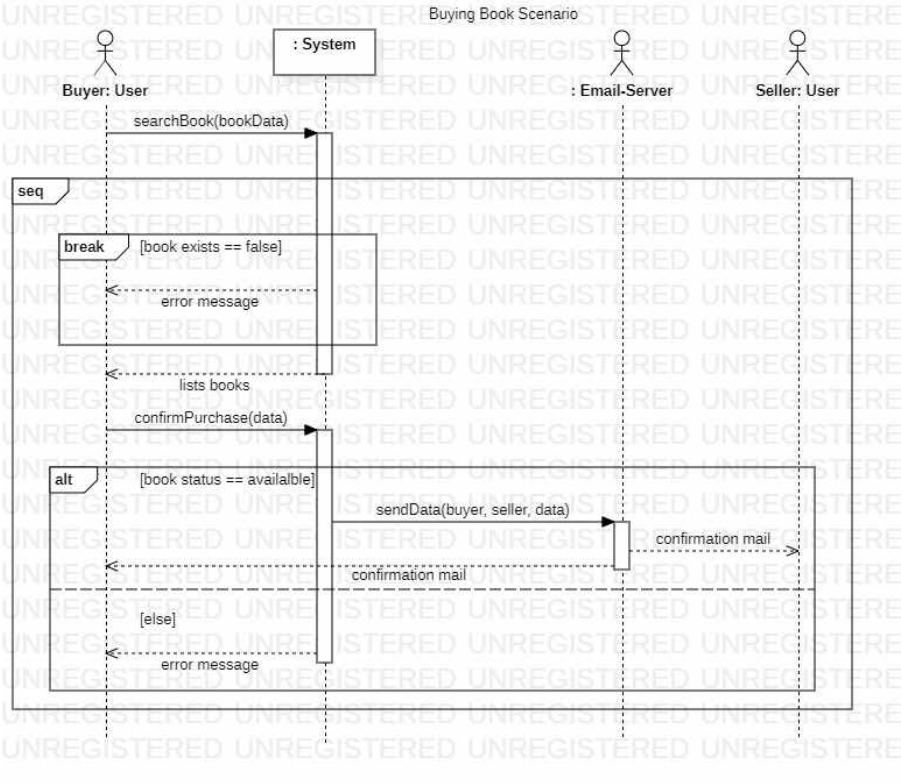
1a. Book not found
  1. System displays that there's no such book.
  2. Admin probably search another book (Main Success Scenario #1).

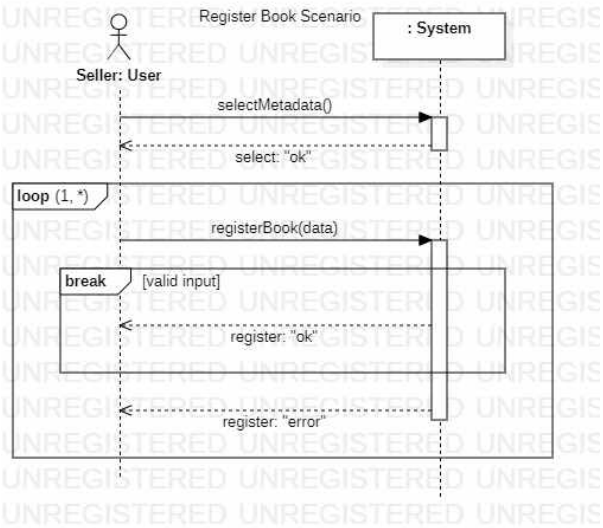
2a. Seller deletes the book.
  1. System notices admin the book doesn't exist.
  2. Admin may search another book (Main Success Scenario #1).

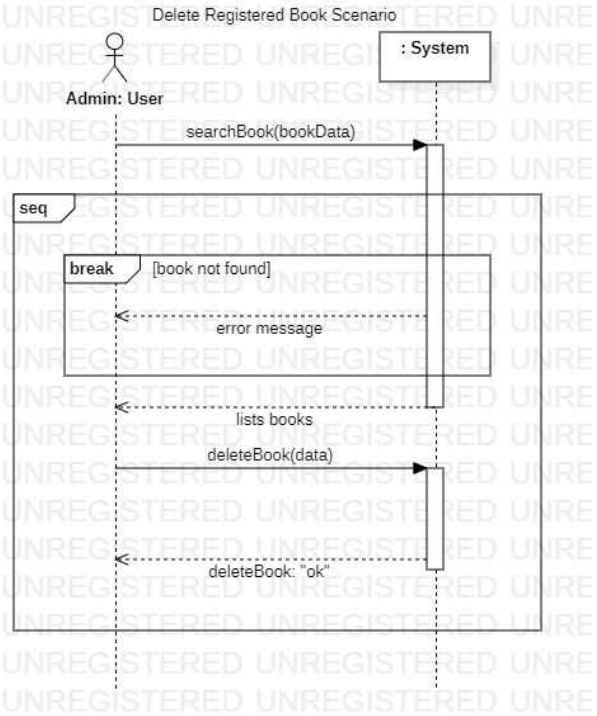
• Domain Model

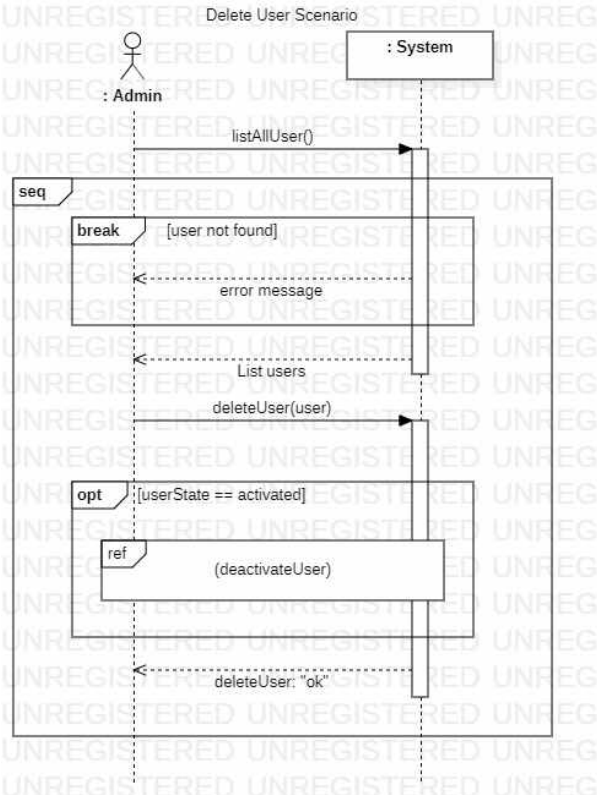


- System Sequence Diagram






Delete Registered Book Scenario
: System
Admin: User
searchBook(bookData)
seq
break
[book not found]
error message
lists books
deleteBook(data)
deleteBook: "ok"


Delete User Scenario
: System
: Admin
listAllUser()
seq
break
[user not found]
error message
List users
deleteUser(user)
opt
[userState == activated]
ref
(deactivateUser)
deleteUser: "ok"

• Operation Contracts

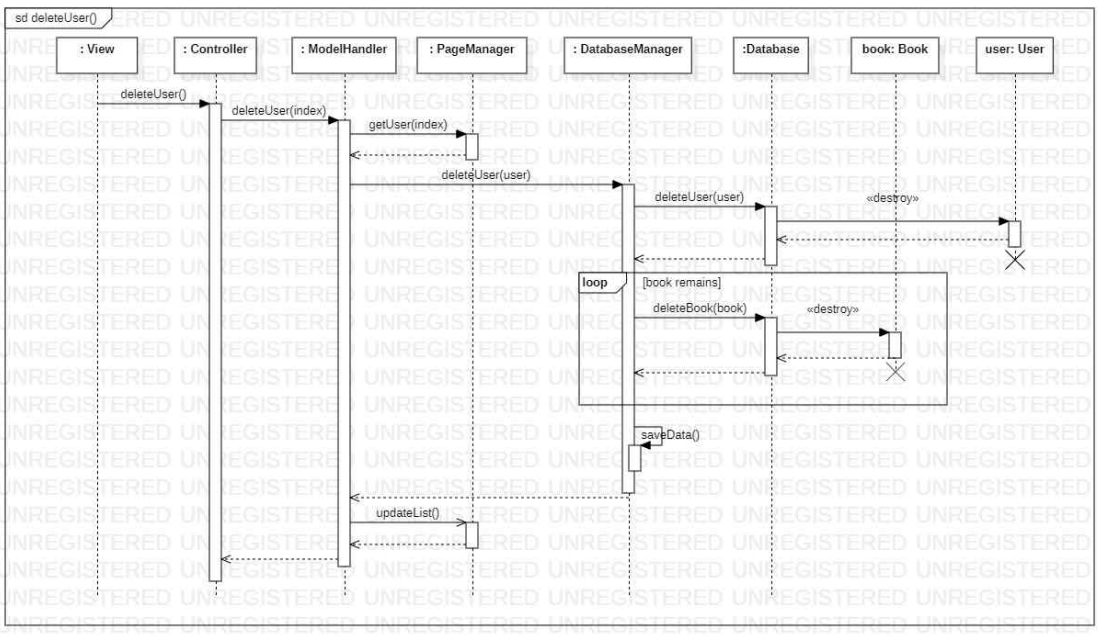
Operation:	registerBook(data)
Cross Reference:	Use Case: Register Book
Preconditions:	Valid input data (book data)
Postconditions:	• Attributes of newBook were initialized (attributes changed). • newBook was associated with the user's BookList (association formed).

Operation:	deleteUser()
Cross Reference:	Use Case: Delete User
Preconditions:	Deactivated user
Postconditions:	• User was deleted (instance deletion). • User's bookList was deleted (instance deletion).

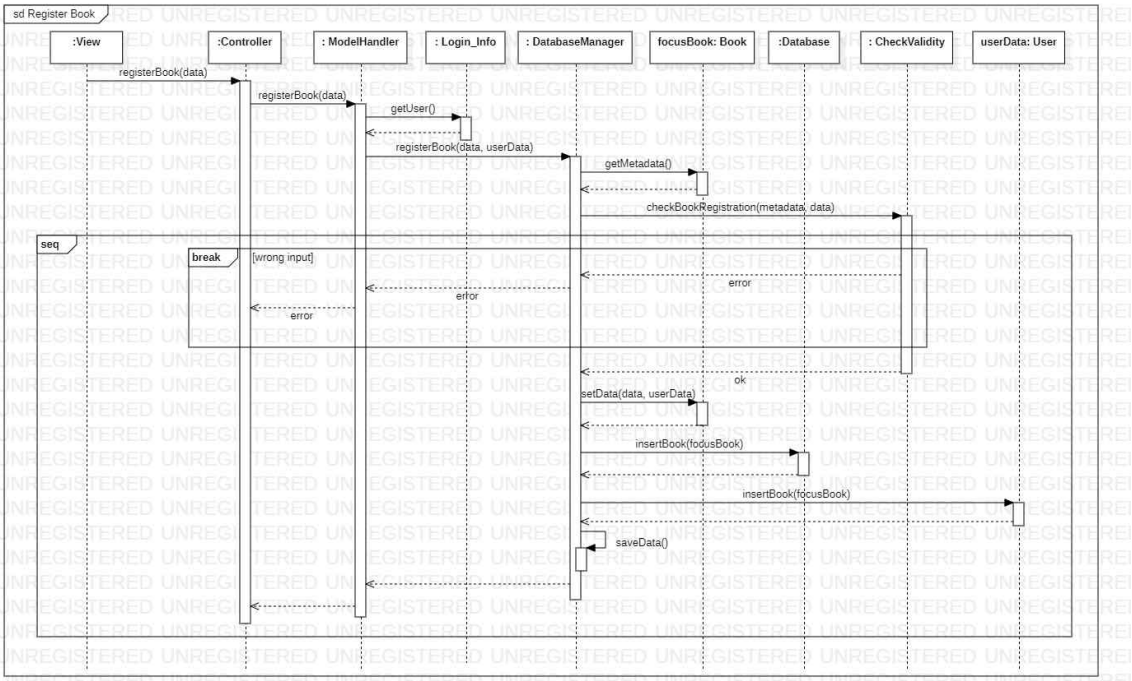
Operation:	selectMetadata()
Cross Reference:	Use Case: Register Book
Preconditions:	
Postconditions:	• newBook was associated with metadata (association formed).

• Sequence Diagram

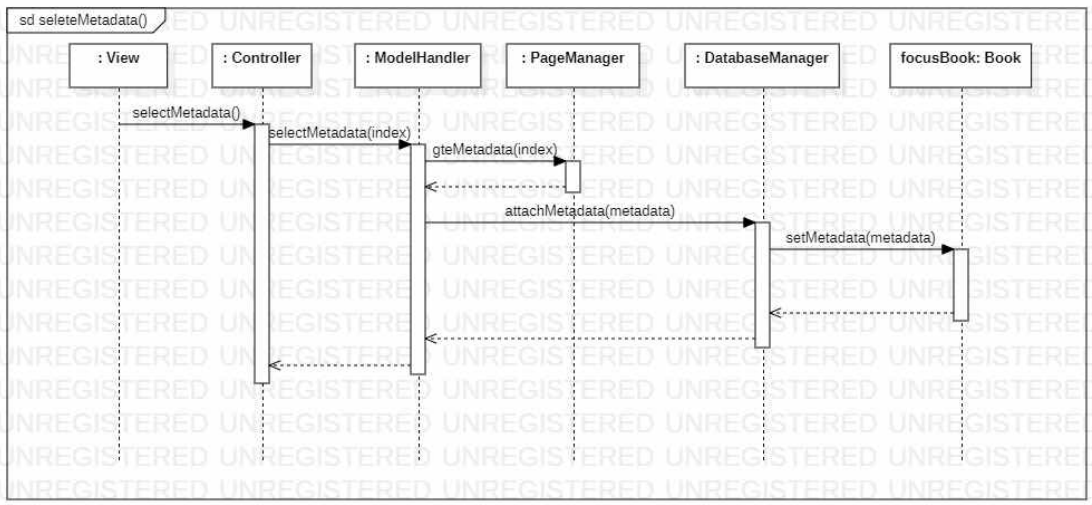
- deleteUser()



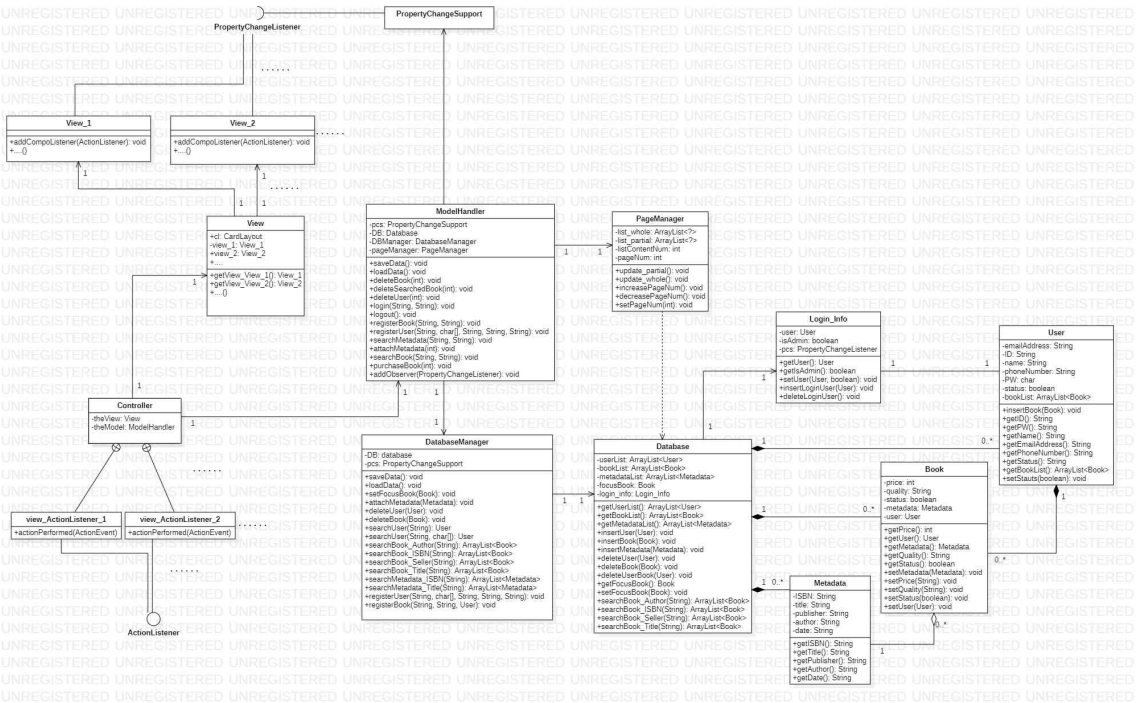
- registerBook()



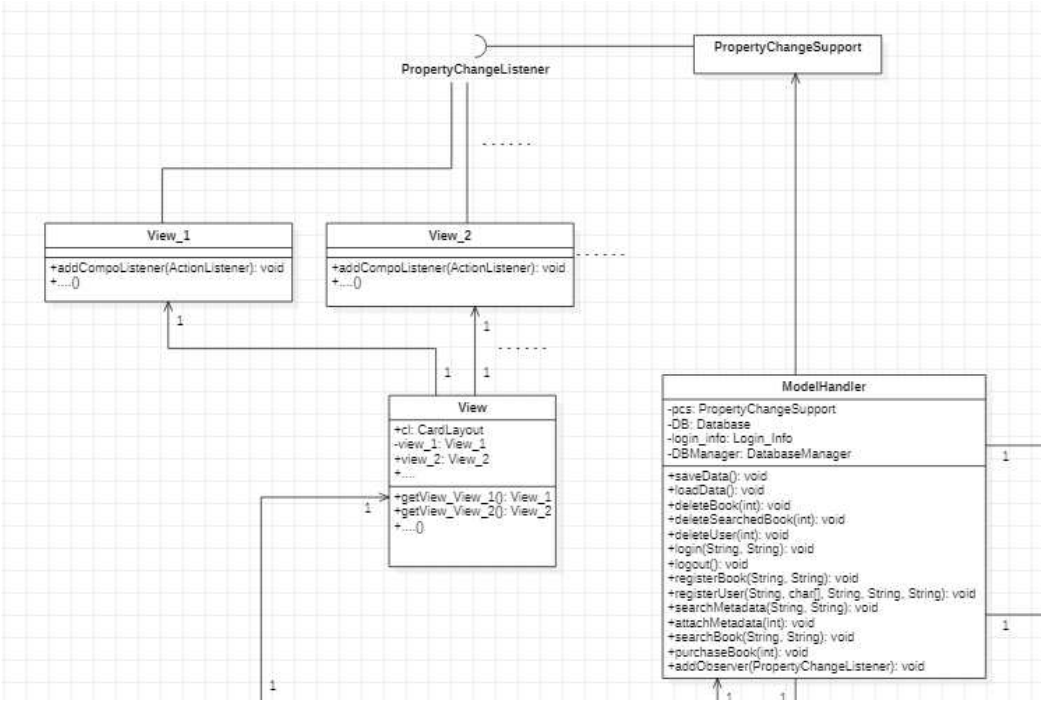
- selectMetadata()



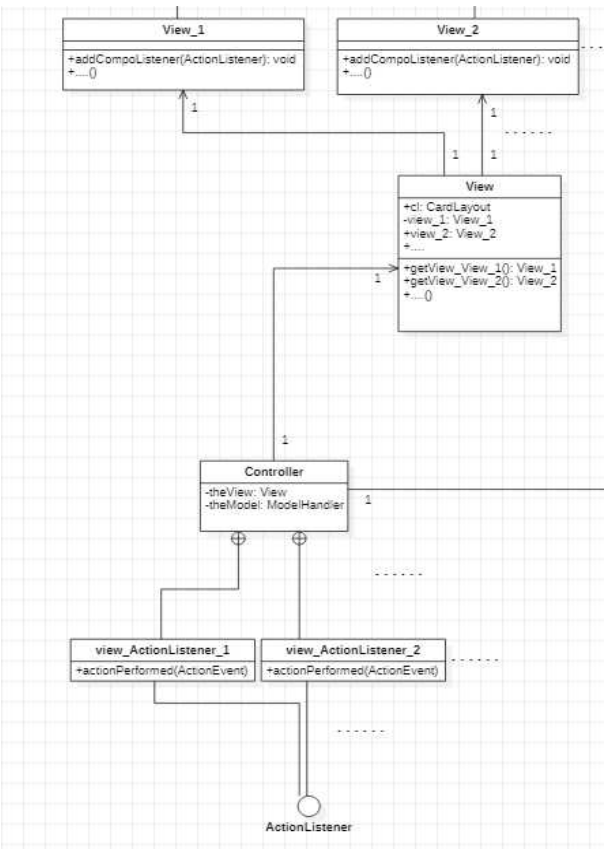
- Design Class Diagram



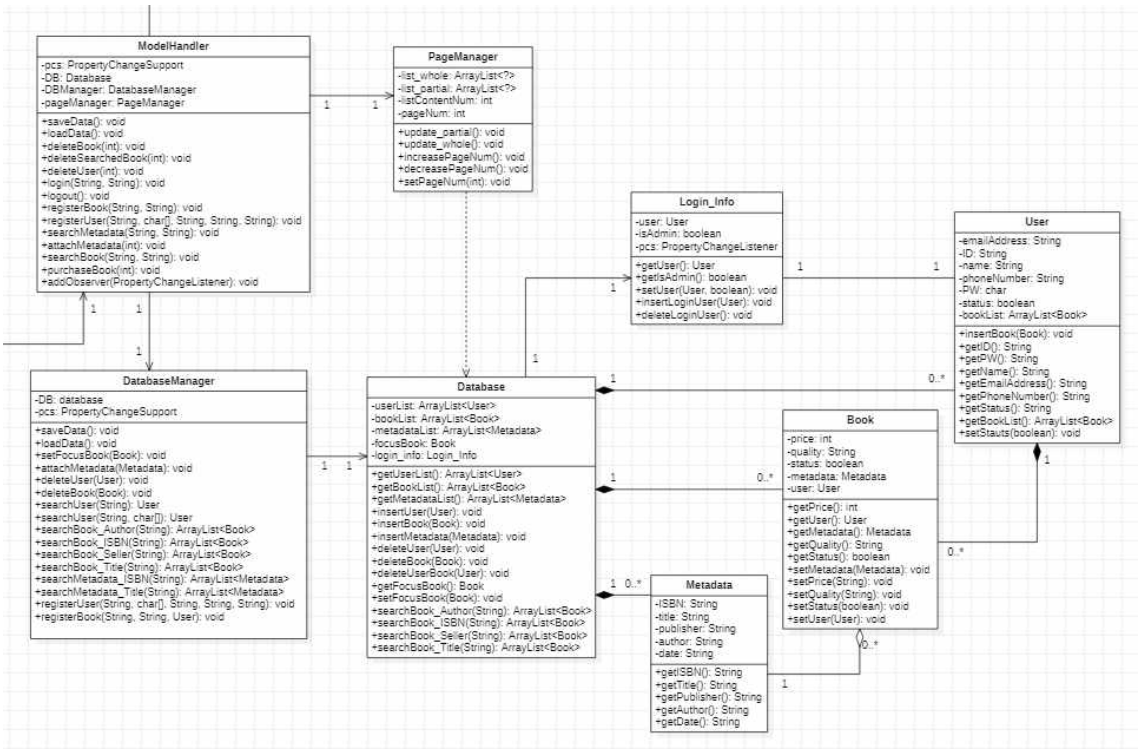
<Model & View>



<Controller & View>



<Model>



- Code

1.MVC 설계 코드

```
package bookTransactionSystem;

public class MVCmain {
    @SuppressWarnings("unused")
    public static void main(String[] args) {
        ModelHandler theModel = new ModelHandler();
        View theView = new View();
        Controller theController = new Controller(theView, theModel);
    }
}
```

- main 코드입니다. Model, View, Controller에 해당하는 클래스의 인스턴스를 만들고 있습니다. Model과 View는 서로에 대한 정보가 없고 Controller만이 Model과 View의 정보를 가지고 있습니다. View와 Model은 인터페이스를 통해 소통합니다.

1-1. View와 Model

```
public class View extends JFrame implements PropertyChangeListener{
```

- View에서 PropertyChangeListener 인터페이스를 implements한 모습입니다. 기존의 Observer 인터페이스가 없어져 PropertyChangeListener를 사용하게 되었습니다.

```
public class ModelHandler {
    private Database DB;
    private Login_Info login_info;
    private DatabaseManager DBManager;

    private PropertyChangeSupport pcs;
```

```
ModelHandler(){
    pcs = new  PropertyChangeSupport(this);
    DB = new Database(pcs);
    login_info = new Login_Info(pcs);
    DBManager = new DatabaseManager(DB, pcs);

    userPage = new UserPageManager(10, pcs);
    manageBookPage = new ManageBookPageManager(4, pcs);
    searchedBookPage = new SearchedBookPageManager(4, pcs);
    metadataPage = new MetadataPageManager(8, pcs);
```

- Model에서 PropertyChangeSupport 인스턴스를 만들고, constructor에서 각각의 Model들에게 이 인스턴스를 전달하는 코드입니다.

```
public void addErrorObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("error", l);
}
public void addConfirmObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("confirm", l);
}
public void addFocusBookObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("focusBook", l);
}
public void addLoginObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("login", l);
}
public void addManageBookListObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("manageBookList_partial", l);
}
public void addSearchBookListObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("searchedBookList_partial", l);
}
public void addSearchMetadataListObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("metadataList_partial", l);
}
public void addUserListObserver(PropertyChangeListener l) {
    pcs.addPropertyChangeListener("userList_partial", l);
}
```

```
public void setFocusBook(Book focusBook) {
    this.focusBook = focusBook;
    pcs.firePropertyChange("focusBook", null, focusBook);
}
```

```
public void insertLoginUser(User user) {
    if(user.getID().equals("admin"))
        isAdmin = true;
    this.user = user;
    pcs.firePropertyChange("login", null, this);
}
```

```
    pcs.firePropertyChange("manageBookList_partial", null, list_partial);
```

Model에서, 각각의 update value에 대하여 subscriber를 추가하는 함수를 만들고 update가 일어난 지점에서 notify하는 코드입니다. Old Value값은 사용하지 않았습니다.

```
Controller(View theView, ModelHandler theModel){
    this.theView = theView;
    this.theModel = theModel;

    this.theModel.addErrorObserver(this.theView);
    this.theModel.addConfirmObserver(this.theView);

    this.theModel.addLoginObserver(this.theView.getView_Main());
    this.theModel.addLoginObserver(this.theView.getView_ManageBook());
    this.theModel.addLoginObserver(this.theView.getView_ManageUser());
    this.theModel.addLoginObserver(this.theView.getView_SearchResult());

    this.theModel.addFocusBookObserver(this.theView.getView_RegisterBook());
    this.theModel.addFocusBookObserver(this.theView.getView_EditBook());

    this.theModel.addManageBookListObserver(this.theView.getView_ManageBook());
    this.theModel.addSearchBookListObserver(this.theView.getView_SearchResult());
    this.theModel.addSearchMetadataListObserver(this.theView.getView_RegisterBook());
    this.theModel.addUserListObserver(this.theView.getView_ManageUser());
```

- Controller의 constructor에서 View를, Model에서 정의한 method를 이용해 subscriber로 추가하는 모습입니다.

```
public void propertyChange(PropertyChangeEvent evt) {
    if(evt.getNewValue() != null){
        Book book = (Book)evt.getNewValue();
        Metadata metadata = book.getMetadata();

        lblISBNCont.setText(metadata.getISBN());
        lblTitleCont.setText(metadata.getTitle());
        lblAuthorCont.setText(metadata.getAuthor());
        lblPublisherCont.setText(metadata.getPublisher());
        lblDateCont.setText(metadata.getDate());
        txtFPrice.setText(String.valueOf(book.getPrice()));
        comboBox.getModel().setSelectedItem(book.getQuality());
    }
}
```

- View에서, Model에서 update된 값을 이용해 View component 값을 바꾸는 모습입니다.

이렇게 View와 Model은 서로를 직접 알지 못하고 인터페이스를 통해 필요한 정보만을 주고받습니다.

1-2. Controller와 View

```
/***View_RegisterBook***/
class registerBook_registerBtnListener implements ActionListener {
    public void actionPerformed(ActionEvent e)  {
```

```
class registerBook_backBtnListener implements ActionListener {
    public void actionPerformed(ActionEvent e)  {
```

- Controller에는 ActionListner를 implements한 inner class들이 있습니다.

```
public void addCheckBtnListener(ActionListener e){
    btnCheck.addActionListener(e);
}
public void addBackBtnListener(ActionListener e){
    btnBack.addActionListener(e);
}
public void addRegisterBtnListener(ActionListener e){
    btnRegister.addActionListener(e);
}
```

-View마다 각각의 view components에 ActionListener를 추가하는 함수가 있습니다.

```
    theView.getView_Login().addLoginBtnListener(new login_loginBtnListener());
    theView.getView_Login().addRegisterBtnListener(new login_registerBtnListener());

    theView.getView_Register().addBackBtnListener(new register_backBtnListener());
    theView.getView_Register().addCheckBtnListener(new register_checkBtnListener());
    theView.getView_Register().addRegisterBtnListener(new register_registerBtnListener());
```

- Controller의 constructor 부분에서 UI component에 맞는 inner class의 인스턴스를, View의 ActionList를 추가하는 method의 매개변수로 넘겨줍니다. View의 각 component가 사용자의 action을 받을 때 이 동작은 Controller의 inner class 들이 수행하게 됩니다.

1-3. Controller와 Model

```
/***View_RegisterBook***/
class registerBook_registerBtnListener implements ActionListener {
    public void actionPerformed(ActionEvent e)  {
        try {
            theModel.registerBook(theView.getView_RegisterBook().getPrice(), theView.getView_RegisterBook().getQuality());
            theView.getView_ManageBook().addEditBtnListener(new manageBook_editBtnListener());
            theView.getView_ManageBook().addDeleteBtnListener(new manageBook_deleteBtnListener());
            theView.getCardLayout().show(theView.getContentPane(), "manageBook");
        } catch (Exception ex) {
            JOptionPane.showMessageDialog(theView, ex.getMessage(), "Error", JOptionPane.ERROR_MESSAGE);
            ex.printStackTrace();
        }
    }
}

class registerBook_backBtnListener implements ActionListener {
    public void actionPerformed(ActionEvent e)  {
        theView.getCardLayout().show(theView.getContentPane(), "manageBook");
    }
}
```

- Controller는 많은 일을 하지 않고 Model의 method를 호출하고 View panel을 바꾸는 일을 합니다. View를 교체하는 일은 Controller에서 하지만 View 안의 값들은 위에서 언급했듯이 Model에서 firePropertyChange를 호출하여 update합니다.

MVC 패턴을 적용하여 Model을 View와 Controller에서 완전히 분리시켰고 이후에 View의 변화가 있더라도 Model에는 영향을 주지 않습니다.

<이후로 extra point입니다.>

2. Metadata Class

- 같은 종류의 책일 경우 가격과 품질정보를 제외한 나머지의 책 정보는 서로 같습니다. 이러한 책이 많을 경우 공간을 낭비할 수 있으므로 저는 metadata라는 책의 고유한 정보를 빼내어 클래스를 새로 생성했습니다.

```java
private static final long serialVersionUID = 1L;
private String ISBN;
private String title;
private String publisher;
private String author;
private String date;

Metadata(){

}

Metadata(String ISBN, String title, String publisher, String author, String date){
    this.ISBN = ISBN;
    this.title = title;
    this.publisher = publisher;
    this.author = author;
    this.date = date;
}

public String getISBN() {
    return ISBN;
}
public String getTitle() {
    return title;
}
public String getPublisher() {
    return publisher;
}
public String getAuthor() {
    return author;
}
public String getDate() {
    return date;
}
```

- metadata는 프로그램이 시작할 때, 미리 만들어 놓은 txt 파일(세상의 모든 책이라고 가정했습니다)을 읽어와 인스턴스를 생성합니다. 이렇게 만들어진 metadata 인스턴스는 사용자가 책을 등록할 때 ISBN 또는 Title을 기준으로 검색되며 이때 선택된 metadata는 책과 연결됩니다.

3. Database Class

- Database 클래스를 사용하여 시스템에서 관리 중인 자원을 모았습니다. GRASP으로 보면 Pure Fabrication Pattern을 적용하여 cohesion과 code reuse를 강화했습니다.

```java
public class Database {
    private ArrayList<User> userList;
    private ArrayList<Book> bookList;
    private ArrayList<Metadata> metadataList;

    Database(PropertyChangeSupport pcs){
        userList = new ArrayList<User>();
        bookList = new ArrayList<Book>();
        metadataList = new ArrayList<Metadata>();
    }

    public ArrayList<User> getUserList() {
        return userList;
    }
    public ArrayList<Book> getBookList() {
        return bookList;
    }
    public ArrayList<Metadata> getMetadataList(){
        return metadataList;
    }
```

```java
    public void insertUser(User user) {
        userList.add(user);
    }
    public void insertBook(Book book) {
        bookList.add(book);
    }
    public void insertMetadata(Metadata metadata) {
        metadataList.add(metadata);
    }

    public void deleteBook(Book book) {
        bookList.remove(book);
        book.getUser().getBookList().remove(book);
    }
    public void deleteUser(User user) {
        userList.remove(user);
    }

    public void deleteUserBook(User user) {
```

4. PageManager & DatabaseManager Class

- ModelHandler의 크기가 너무 커지지 않도록 역할을 분담하는 클래스를 만들었습니다. database를 관리하는 DatabaseManager 클래스와, view에서 페이지 형태로 나타나는 data들(검색된 책 목록, 관리 중인 책 목록, 관리 중인 사용자 목록, metadata 목록)을 관리하는 PageManager를 만들어 ModelHandler 클래스의 responsibility를 줄이고

cohesion을 높였습니다.

또한 Creator pattern을 적용하여 Database의 새 인스턴스는 모두 DatabaseManager 클래스에서 만들도록 설계했습니다.

```
public class SearchedBookPageManager {
    private ArrayList<Book> list_whole;
    private ArrayList<Book> list_partial;

    private int listContentNum;
    private int pageNum;
    private PropertyChangeSupport pcs;

    SearchedBookPageManager(int listContentNum, PropertyChangeSupport pcs){
        pageNum = 1;
        this.pcs = pcs;
        this.listContentNum = listContentNum;
        list_partial = new ArrayList<Book>();
        list_whole = new ArrayList<Book>();
    }
    public void updatePartial() {
        list_partial.clear();
        for(int i = (pageNum - 1) * listContentNum;
                i <= list_whole.size() - 1 && i < pageNum * listContentNum; i++)
            list_partial.add(list_whole.get(i));
        if(list_partial.size() == 0 && list_whole.size() != 0) {
            decreasePageNum();
            updatePartial();
        }
        pcs.firePropertyChange("searchedBookList_partial", null, list_partial);
    }

    public void decreasePageNum() {
        if(pageNum != 1)
            pageNum--;
    }
    public void increasePageNum() {
        if(pageNum <= (list_whole.size() - 1) / listContentNum)
            pageNum++;
    }
    public void updateWhole(ArrayList<Book> list_whole) {
        this.list_whole = list_whole;
    }
```

- PageManager의 전체 코드입니다. 검색된 책 목록을 예로 들면, list_whole은 검색된 모든 책의 ArrayList이고 list_partial은 그중 화면에 보이는 책의 ArrayList입니다. listContentNum은 목록에 출력되는 컨텐츠의 수이고 pageNum은 현재 페이지가 몇 페이지인지 나타내는 변수입니다.
- 아쉬운 점이 있다면 PageManager를 자바의 wildcard를 사용하여 polymorphism 기법으로 나타내려고 했지만 끝내 실패하고 네 개의 분리된 클래스로 만들었다는 점입니다.

```java
public void deleteBook(int index) {
    Book book = manageBookPage.getList_partial().get(index);
    DBManager.deleteBook(book);
    manageBookPage.updatePartial();
}
public void deleteSearchedBook(int index) {
    Book book = searchedBookPage.getList_partial().get(index);
    DBManager.deleteBook(book);
    searchedBookPage.delete(index);
    searchedBookPage.updatePartial();
}
public void deleteUser(int index) {
    User user = userPage.getList_partial().get(index);
    DBManager.deleteUser(user);
    userPage.deleteUser(user);
    userPage.updatePartial();
}
public void editBook(String price, String quality) throws Exception {
    DBManager.editBook(price, quality);
    manageBookPage.updatePartial();
}
```

- 간소화된 ModelHandler 클래스의 코드입니다. Controller에게 받은 명령은 DBManager와 PageManager에게 위임합니다.

5. CheckValidity Class

- CheckValidity 클래스를 만들고 안에 public static method를 만들어 사용자의 input의 유효성은 이 클래스 안에서 해결할 수 있도록 했습니다. cohesion을 높이고 추가로 exception handling도 편하게 할 수 있었습니다.

```java
public abstract class CheckValidity {
    /***Login***/
    public static void checkLogin_Input(String ID, char[] PW) throws Exception {
        if(ID.length() == 0)
            throw new Exception("Enter your ID.");
        if(PW.length == 0)
            throw new Exception("Enter your PW.");
    }

    public static void checkValidUser(boolean status) throws Exception {
        if(!status)
            throw new Exception("Deactivated User");
    }
    /***Login***/

    /***User Registration***/
    public static void checkRegistration_Input(ArrayList<User> userList, String ID, char[] PW, String name, String phoneNumber,String emailAddress) throws Exception {
        checkRegistration_ID(ID, userList);
        checkRegistration_PW(PW);
        checkRegistration_Name(name);
        checkRegistration_PhoneNumber(phoneNumber);
        checkRegistration_Email(emailAddress);
    }

    public static boolean checkRegistration_ID(String ID, ArrayList<User> userList) throws Exception {
        for(User user:userList) {
            if(ID.equals(user.getID()))
                throw new Exception("Current ID has already been registered.");
        }
        if(ID.length() < 1 || ID.length() > 10)
            throw new Exception("Length of ID: 1 to 10");
        return true;
    }

    private static void checkRegistration_PW(char[] PW) throws Exception {
        if(PW.length < 5 || PW.length > 12)
            throw new Exception("Length of PW: 5 to 12");
    }
    private static void checkRegistration_Name(String name) throws Exception {
        if(name.length() < 1)
```

- checkValidity 클래스의 코드입니다. 각각의 input에 대해 검사하는 함수를 나눠 exception을 발생시키는 모습입니다.

6. 전체적인 Coupling의 최소화

- coupling을 최소화하여 코드를 설계했습니다.

```
public class ModelHandler {
    private PropertyChangeSupport pcs;

    private DatabaseManager DBManager;
    private UserPageManager userPage;
    private ManageBookPageManager manageBookPage;
    private SearchedBookPageManager searchedBookPage;
    private MetadataPageManager metadataPage;

    ModelHandler(){
        pcs = new  PropertyChangeSupport(this);
        DBManager = new DatabaseManager(pcs);

        userPage = new UserPageManager(10, pcs);
        manageBookPage = new ManageBookPageManager(4, pcs);
        searchedBookPage = new SearchedBookPageManager(4, pcs);
        metadataPage = new MetadataPageManager(8, pcs);

        metadataPage.updateWhole(DBManager.getMetadataList());
        userPage.updateWhole(DBManager.getUserList());
    }
```

- ModelHandler 클래스 내부의 변수들

```
public class DatabaseManager {
    private Database DB;
    private PropertyChangeSupport pcs;

    DatabaseManager(PropertyChangeSupport pcs){
        this.pcs = pcs;
        DB = new Database(pcs);
        loadData();
    }
```

- DatabaseManager 클래스 내부의 변수들

```
public class Database {
    private Login_Info login_info;
    private Book focusBook;
    private ArrayList<User> userList;
    private ArrayList<Book> bookList;
    private ArrayList<Metadata> metadataList;

    Database(PropertyChangeSupport pcs){
        userList = new ArrayList<User>();
        bookList = new ArrayList<Book>();
        metadataList = new ArrayList<Metadata>();
        login_info = new Login_Info(pcs);
    }
```

- Database 클래스 내부의 변수들
- 불필요한 관계가 생기지 않도록 Manager들에게 이를 위임하여 Coupling을 낮췄습니다.

7. 프로그램 추가 기능

- 데이터의 save와 load가 가능합니다.
- input string의 exception handling이 가능합니다.
- 책을 등록할 때 사용자는 책의 ISBN 혹은 Title을 검색하여 책의 정보를 자동으로 입력할 수 있습니다. 사용자는 가격정보와 품질정보만 추가하여 등록합니다. 모든 책은 ISBN이 등록되어 있다고 가정했습니다.
- 등록한 책을 수정할 때도 마찬가지로 가격정보와 품질정보만 수정할 수 있습니다. 다른 종류의 책을 올린 경우 삭제하고 다시 올리는 것을 가정했습니다.

- 구매가 완료된 책은 수정하거나 검색될 수 없습니다. 삭제만 가능합니다.
- Admin에 의해 deactivate된 사용자의 책은 일반 사용자가 검색할 수 없고 admin 만이 검색, 삭제할 수 있습니다.

• Test Case Class

```
public class DatabaseTest extends TestCase{
    private Database database;
    protected void setUp() {
        database = new Database(new PropertyChangeSupport(this));
    }

    @Test public void testInsertBook() {
        assertEquals(0, database.getBookList().size());
        database.insertBook(new Book());
        assertEquals(1, database.getBookList().size());
    }
    @Test public void testSearchBook_ISBN() {
        Book book = new Book();
        Metadata metadata = new Metadata("111", null, null, null, null);
        book.setMetadata(metadata);
        book.setUser(new User(null, null, null, null, null));

        assertEquals(0, database.searchBook_ISBN("1").size());
        database.insertBook(book);
        assertEquals(1, database.searchBook_ISBN("1").size());

        book = new Book();
        metadata = new Metadata("211", null, null, null, null);
        book.setMetadata(metadata);
        book.setUser(new User(null, null, null, null, null));

        database.insertBook(book);
        assertEquals(2, database.searchBook_ISBN("1").size());
        assertEquals(1, database.searchBook_ISBN("2").size());

        book = new Book();
        metadata = new Metadata("331", null, null, null, null);
        book.setMetadata(metadata);
        book.setUser(new User(null, null, null, null, null));

        database.insertBook(book);
        assertEquals(3, database.searchBook_ISBN("1").size());
        assertEquals(1, database.searchBook_ISBN("2").size());
    }
```

```
    @Test public void testSearchMetadata_Title() {
        Metadata metadata = new Metadata("1", "Title", "Publihser", "Author", "2010-10-02");
        Metadata metadata2 = new Metadata("3", "Turtle", "Pub", "Auth", "2010-01-02");

        database.insertMetadata(metadata);
        database.insertMetadata(metadata2);

        assertEquals(1, database.searchMetadata_Title("Title").size());
        assertEquals(2, database.searchMetadata_Title("tle").size());
        assertEquals(1, database.searchMetadata_Title("Tit").size());
    }
```

```java
public class PageManagerTest extends TestCase{
    UserPageManager userPage;
    ArrayList<User> userList;

    protected void setUp() {
        userPage = new UserPageManager(4, new PropertyChangeSupport(this));
        userList = new ArrayList<User>();
        userList.add(new User("admin", null, null, null, null));
        for(int i = 0; i < 6; i++)
            userList.add(new User(null, null, null, null, null));
    }

    @Test public void testUpdateWhole() {
        userPage.updateWhole(userList);
        assertEquals(6, userPage.getList_whole().size());
    }

    @Test public void testUpdatePartial() {
        userPage.updateWhole(userList);
        userPage.updatePartial();
        assertEquals(4, userPage.getList_partial().size());
    }
```

```java
  @Test public void testIncrease() {
      userPage.updateWhole(userList);
      userPage.updatePartial();

      assertEquals(1, userPage.getPageNum());
      userPage.increasePageNum();
      assertEquals(2, userPage.getPageNum());
      assertEquals(2, userPage.getList_partial().size());

      //마지막 페이지에서 increase 호출했을 때
      userPage.increasePageNum();
      assertEquals(2, userPage.getPageNum());
      assertEquals(2, userPage.getList_partial().size());
  }

  @Test public void testDecrease() {
      userPage.updateWhole(userList);
      userPage.updatePartial();

      userPage.increasePageNum();

      userPage.decreasePageNum();
      assertEquals(1, userPage.getPageNum());
      assertEquals(4, userPage.getList_partial().size());

      //첫 페이지에서 decrease 호출했을 때
      userPage.decreasePageNum();
      assertEquals(1, userPage.getPageNum());
      assertEquals(4, userPage.getList_partial().size());
  }
```