

# SOFTENG 251 Object Oriented Software Construction

## Assignment 2: Personal Movie Database (PMDB)

Due: 24th April (end of lab)

April 1st, 2008

### Introduction

Let's say you are writing a simple desktop application for maintaining information about movies and movie-related people, i.e. cast and crew, in a manner similar to the movie databases on the web such as [www.imdb.com](http://www.imdb.com) and [www.allmovies.com](http://www.allmovies.com). The application, which you decide to call *Personal Movie Database (PMDB)*, manages a collection of items, each containing information about either films or people, and supports operations for listing, adding, removing, sorting and searching these items. Of course, being a "database" (although not in the strict sense), the program also supports persisting the collection of items onto a file and loading from it afterwards. For this assignment, you are provided with a basic skeleton code for the program, downloadable from the *Assignments* section of the course page, and your job is to complete the necessary bits of the program as indicated by the tasks explained later.

You are provided with two additional components to the Movie Database:

- A set of *JUnit tests* for ensuring the code for your Movie Database is completed correctly. The tests are complete, so you are not required to add further tests of your own, although you are most welcome to.
- A *Movie Browser*, which is a graphical user interface (GUI) to your Movie Database allowing users to interactively look up and search the information, as shown in figure 1. You do not need to modify its code. The Browser not only adds a "coolness" value to your Movie Database but also acts as a secondary mechanism for visually "testing" your Database.

This assignment will get you to utilise of a variety of classes from the java Collections framework and also to create reusable, flexible methods and classes with the help of Generics.

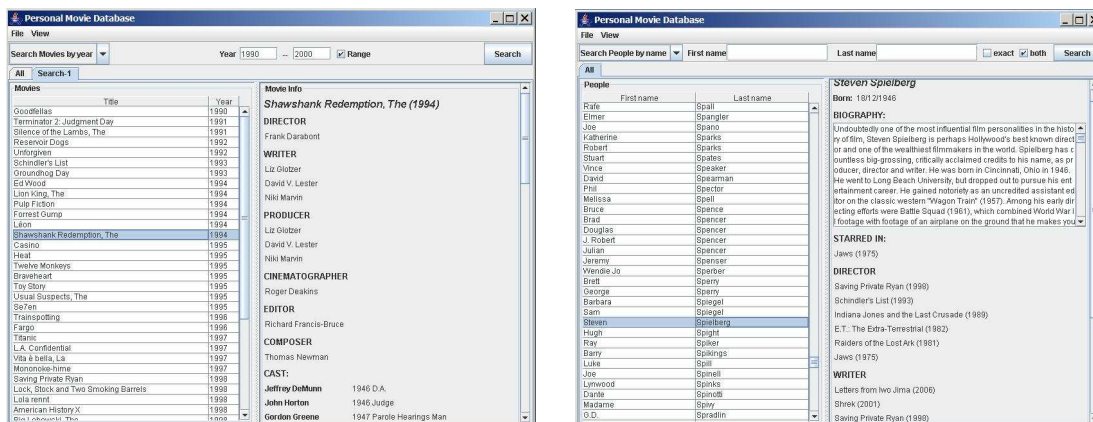


Figure 1: The Browser after you successfully complete your Movie Database

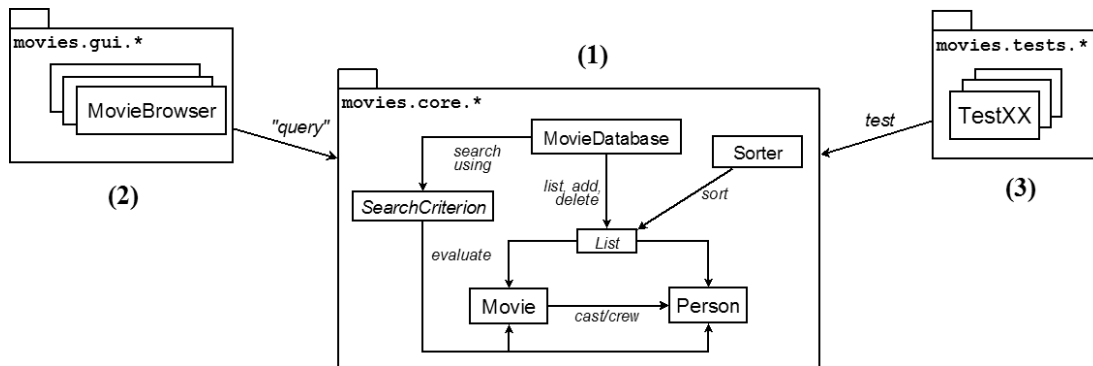


Figure 2: An overview of the elements of Movie Database

## Details of PMDB

The program you will be working on is of a considerably larger scale than in the previous assignment. However, *do not* feel overwhelmed by this, as the actual number of classes you need to work on is reasonably small. In fact a significant portion of the program is dedicated to the Browser, the details of which you *do not* have to worry about, and the unit tests, which you are encouraged to study but not required to modify.

Figure 2 shows the structure of your Personal Movie Database (PMDb) at a glance. You can see that it has three main packages: `movies.core` (1), `movies.gui` (2) and `movies.tests` (3).

1. Classes in the `movies.core` package make up the core Movie Database. Class `Person` contains basic attributes of a movie-related person including name, date of birth and biography. Class `Movie` has basic attributes for a film, including title and year of release. It should also store information about cast and crew by making references to `Person` objects: you have to implement this in Task 4. Class `MovieDatabase` stores all `Movie` and `Person` objects in their respective `List`s, providing operations (i.e. methods) for listing, adding and removing movies and people, which you are to complete through Task 1. Class `Sorter` defines methods for sorting any given `List` of movies or people by different attributes: this you are to implement through Task 2 and Task 3. `MovieDatabase` defines further operations for searching movies and people according to various kinds of customisable criteria, represented by the interface `SearchCriterion`: this will become clearer as you tackle Task 5.
2. The `movies.gui` package, along with its many subpackages, contains all classes comprising the graphical Movie Browser. Again, you *do not* need to understand the classes in this package. All you need to know is that the Browser can be launched by running the class `MovieBrowser`.
3. The package `movies.tests` contains all unit tests divided effectively into two test suites: the *essential* tests (`movies.tests.essential` subpackage) and the *optional* tests (`movies.tests.optional` subpackage). The essential tests are the ones that **must** be passed when you have completed all of the basic tasks (1 to 5). The optional tests on the other hand are designed to validate the optional tasks (6 and 7), should you choose to do them. Thus if you're not doing them, then it does **not** matter whether any of these tests passes.

For more details on the individual classes contained in the above packages, consult the Javadoc (Java documentation) pages available on the *Assignments* section of the course web page. You can alternatively read the comments within the Java source files for these classes — you get the same information either way (the Javadoc webpages were automatically generated from these comments through the Javadoc tool. Neat huh?).

## Assessment Criteria

1. Functional correctness — your program passes all provided unit tests, and the Browser functions properly to reflect this.
2. Strong evidence of effort to 1) apply good object-oriented programming techniques, such as using inheritance and polymorphism when appropriate; and to 2) reuse as much of the available code as possible — i.e. avoid reinventing the wheel — both from the Java standard library and from within your own program.
3. Demonstration of a thorough understanding of your own work by being able to answer questions from the lecturer or the tutors.

## Tasks

It is strongly recommended that you complete the following tasks in order. Associated with each task is a test case that must pass on completion. In the end, if all of the *essential* test cases (i.e. from the `movies.tests.essential` package) pass, then you can be fairly confident your implementation is correct. You are encouraged to look at the test cases and study what exactly is being tested to better understand the requirements of your program.

Obviously, initially most of the tests will fail. However, as you successfully complete each task, you will find your test cases passing one by one. Similarly, the Movie Browser will initially not function properly, but as you tackle each task you will see the appropriate changes being reflected in the Browser.

### Task 1 Add/Delete Movies and People

Under the class `MovieDatabase`, complete the methods `addPerson()`, `deletePerson()`, `addMovie()` and `deleteMovie()`, which should be self-explanatory. Read the Javadocs for each of these methods for details on parameters and required return values.

After you successfully complete this task:

- `TestAddDeleteListMoviesAndPeople` should pass.
- The Browser should correctly display a list of movies and people as you can see in the tables on the left-hand side of the windows in figure 1.

### Task 2 Define natural ordering for Person

As discussed in lectures, one way to allow objects of a particular type to be sorted is to define a *natural ordering* of the type, by making the type implement the interface `java.lang.Comparable`. You need to define the natural ordering for `Person`, which has been made to implement `Comparable`, where the method `compareTo()` is left for you to complete. Basically, the natural ordering between any two people is determined by comparing their last names first, followed by their first names and lastly date of birth. Read the Javadoc for this method for the exact details on how the ordering is determined.

After you successfully complete this task:

- `TestPersonBasics` should pass.
- In the Browser, you should be able to sort people according to their natural ordering. To do this, switch to “People View” (refer to window on the right in figure 1), then click on the table header called “Last name”.

### Task 3 Sort Movies and People

Another way to allow objects to be sorted is to define a `Comparator` (from `java.util`) for the objects' type, as discussed in lectures. The advantage to using Comparators is that the same type can be ordered in many different ways. Thus, for example, to be able to sort `Person` objects by first name instead of last name, you can write your own `Comparator` implementation to compare people by their first names. To perform the actual sorting, the class `Collections` (also from `java.util`) provides the method `sort()` that takes in a `List` and `Comparator` as argument.

You need to flesh out the `Sorter` class by completing the methods `sortMoviesByTitle()`, `sortMoviesByYear()`, `sortPeopleByFirstName()` and `sortPeopleByLastName()` in order to sort movies by title or by year, and sort people by first name or by last name, respectively. Each method takes in two parameters: the first being the actual list to sort, and the second being a boolean indicating whether the sorting should be done in ascending or descending order. By default, the `Collections.sort()` method sorts in ascending order, so you need a way to reverse the operation as well. The methods `reverse()` and `reverseOrder()` may prove useful.

After you successfully complete this task:

- `TestSort` should pass.
- In the Browser, you should now be able to sort by all four attributes “Title”, “Year”, “First name” and “Last name” — by clicking on the corresponding table headers. Each successive click will toggle between ascending and descending sort.

### Task 4 Associate cast and crew with a Movie

Modify `Movie` so that it stores cast and crew information.

- A movie has a set of characters, each played by a single actor (`Person`). It is possible for several characters to be played by the same actor.
- A movie has a set of crew roles (e.g. director, writer, producer), each filled by a single crew member (`Person`). Again, it is possible for several roles to be filled by the same crew member, but only one person can fill each role. Notice this is a slight simplification of the real world, as it makes more sense for multiple people to be involved in a single role: you can do this by completing the Optional Task 6.

Complete the following methods: `addCharacter()`, `getActorPlaying()`, `removeCharacter()`, `getActors()`, `getCharacters()`, `addCrewRole()`, `getCrewMemberFor()` and `removeCrewRole()`. You can find the details for what each of these methods should do along with explanations of parameters and return value in the methods' Javadoc comments. You may consider using a `HashMap` or a `TreeMap` to store the mappings from characters to actors, and likewise from roles to crew members.

Note that the Collection of `Person` and `String` objects returned from methods `getCharacters()` and `getActors()` must be sorted according to the natural ordering of the respective objects. You may consider returning a `TreeSet`, which automatically ensures all of its items are sorted.

After you successfully complete this task:

- `TestCastCrew` should pass.
- The Browser should display, on the right hand side, the cast and crew list for each selected movie while you're in “Movies View” (see figure 1). Note however that for each crew role there would only be at most one crew member listed.

## Task 5 Search Movies and People

Complete `MovieDatabase` to allow a list of movies or people to be searched according to a given search criterion, for example “movies released in the year 2000” or “people with the last name Coppola”. The search criterion can literally be anything, so you wouldn’t want to have to duplicate the searching logic for every criterion you want to specify. Instead what you do is separate the basic searching logic from the individual search criteria. For any criterion, the basic searching logic is always:

```
go through each item of a given list {
    if item satisfies a given criterion then add it to the results
}
return results
```

Thus the idea is to have a single generic method for handling the searching, to which you can “plug in” any kind of search criterion. The method `genericSearch()` has been defined for this purpose, which you have to complete. This method should be able to take in a list of items (of any type) and a search criterion, and return a new list containing only the items satisfying the given criterion. The search criterion can be anything as long as it implements the generic interface `SearchCriterion<T>`, which defines a single method `accept()` that should return true if the given item matches the criterion. The type parameter `T` refers to the type of the item to evaluate, for example `Person` or `Movie`. You will see that three implementations of `SearchCriterion` are provided:

- `PersonNameCriterion` evaluates whether a person has a certain first name and/or last name.
- `MovieTitleCriterion` evaluates whether a movie has a certain title.
- `MovieYearCriterion` evaluates whether a movie was released during a certain year range.

You need to flesh out each of the above classes. Read their Javadocs for a full specification on what they are supposed to match. You are invited to come up with your own search criteria in addition to these three — see Optional Task 8.

After you successfully complete this task:

- `TestSearch` should pass.
- You should be able to search movies by title and year and search people by name through the Browser using the “Search” Button. Each time a search is made, a new tab is created under which the search results are displayed, as you see on the left window in figure 1. The tab can be closed afterwards by double-clicking on it.

## Optional Tasks

The following tasks are optional. The Movie Browser will become fully operational when you complete them.

### Optional Task 6 Many crew members to many roles

Modify `Movie` such that it allows multiple crew members to be associated with a single role. In other words enable a many-to-many relationship between roles and crew members. The methods `addCrewMember()`, `removeCrewMember()` and `getAllCrewMembersFor()` are meant to accommodate this extended relationship. Initially, they simply delegate to the methods you completed in Task 4, so you need to change them.

There are several ways in which you can implement a many-to-many relationship. For instance:

- You can define a `Map` that maps from a role to a `Set` of people. To associate a role to a new crew member you would first get the set of people associated with the role and then add the new crew member to the set.
- You can define a `Set` that holds a collection of `<role,Person>` pairs. This way, you would have to create a new pair for each role-to-crew-member association while avoiding the possible addition of duplicate pairs.

You are encouraged to think about the trade-offs between the different choices.

After you successfully complete this task:

- `TestMultiCrew` should pass.
- The Browser should now list variable numbers of crew members per role.

### Optional Task 7 Movie ↔ Person relationship

Complete the methods `getMoviesStarring()` and `getMoviesCrewedBy()` in `MovieDatabase`. Doing this will automatically allow the Movie Browser to display all `Movies` associated with the `Person`, in addition to its basic information.

Also enforce the integrity constraints between movies and people: when a person is deleted through `Movie.deletePerson()`, make sure all associations made to this person are also removed from all movies for which the person was a cast or crew member. The template methods `Movie.removeActor()` and `Movie.removeCrewMember()` have been provided to you.

After you successfully complete this task:

- `TestMoviesOfPeople` should pass.
- In the Browser's "People View", information about each person should now include all movies he/she has acted and crewed in.

### Optional Task 8 Define additional search criteria

Define your own search criteria by implementing the `SearchCriterion` interface. Be as creative with defining your criteria as you want, for example: "people who have directed more than X movies between years Y and Z", or "movies that have at least one actor playing more than X characters".

You can also relatively easily configure Movie Browser to allow interactive searching using your own search criteria. Refer to the javadoc documentation for `SearchInputPane` from package `movies.gui.search` for details on how to do this.

There are no designated tests for this task. Pat yourself on the back for making it all the way to the end.