

This project turns your DVD library program into a Web application. The application provides read-only behavior. Users may view, search and sort the database, but they may not add, delete or edit movies. You have already written most of the code for this assignment. The challenge for this project is to re-factor your code to conform to a Model-View-Controller (MVC) architecture. You will also need to write some code to validate user input.

Section I: Application Design

You will be given a skeleton program organized as an MVC¹. The view is complete; you must complete the model and controller. This section gives an overview of the MVC design and briefly describes *which* modifications you must make to complete the application. The remaining sections provide more details on *how* to make those modifications.

The Model

The model stores and manipulates the DVD library information. It consists of four basic things:

Movies. A movie is implemented as a class that contains **attributes** (e.g., title, genres, etc.) for a particular film. The movie class provides get and set methods for each movie attribute. Each set method must validate the corresponding attribute's data. It is your job to implement this class.

A database. A database contains information about lots of movies. It is implemented as a dictionary-like class that maps UPCs to movies. The class provides the ability to search and sort movies. The class also lists which attributes a movie may store. You will be provided with a basic `Database` class. It will be your job to inherit from this class to provide search and sort behaviors.

A storage manager. The storage manager shuffles data between disk and memory. You will be provided with an abstract storage manager class and a specific storage manager that pickles and unpickles movies. Your job will be to inherit from the abstract storage manager to define a text-based manager.

Errors. The model contains a custom exception that is raised whenever the model encounters an error.

The model exists on its own and is not coupled to either the view or the controller.

The View

The view converts database information to HTML strings. You will be provided with a fully-working view. Your job is to modify the controller to manipulate the view. The view exists on its own and is not coupled to either the model or the controller.

The Controller

The controller is the glue that binds the model and the view together. It has direct access to and knowledge of both the model and the view. The controller uses this knowledge to display the model. The controller is responsible for populating the model, creating a view, getting and validating user input, detecting errors, and using the view to display the data. You will be provided a basic controller that does everything *except* validate the user input. Your job will be to extend the controller to provide validation.

The application's basic framework consists of a module called `movie1ib`. This module contains three submodules: `model`, `view`, and `controller`. Each submodule contains its own submodules that provide the basic functionality described above. You should not modify any files in `movie1ib`. You should instead extend this functionality, as described in Section III.

Section II: Getting Started

You must get your application to execute on the UTCS Web server. Here are the steps you need to follow to get started:

1. Pick the team member who will host the application on their UTCS Web space. We'll assume that person's login is `partner`.
2. Log in to UTCS as `partner`. In `partner`'s home directory create the directory `public_html/cs105` by executing the command `mkdir -p public_html/cs105`.

¹The design of this Web application is more heavy-handed than is strictly necessary. You could write this entire thing in a single file with about 200 lines of code. However, the design is indicative of more complex, real-world Web applications, and it provides good practice for object-oriented programming in Python.

3. Change to this directory with the command `cd public_html/cs105`.
4. Unpack the project files into this directory with the command `unzip /u/ben/public_html/teaching/cs105_sp08/hw/pa3.zip`.
5. Now you can use the application by browsing to `http://www.cs.utexas.edu/~partner/cs105/moviedb.cgi`. You'll need to enter the user name and password that I will post on the course discussion board. The empty application prints the column names but not the movies. You can sort the database with a URL like `http://www.cs.utexas.edu/~partner/cs105/moviedb.cgi?sort=upc` and search the database with a URL like `http://www.cs.utexas.edu/~partner/cs105/moviedb.cgi?stars=Clooney`. The empty application prints errors for both of these operations.

Section III: Completing the Application

This section describes the basic design in more detail and lists the steps you must take to complete the application. The project consists of the following files:

Movie.py. You will modify this file to create a `Movie` class.

MovieText.py. You will modify this file to implement the text-based storage manager.

MyDatabase.py. You will modify this file to implement database searching and sorting.

MyMovieController.py. You will modify this file to implement Web input validation.

moviedb.cgi. This is the main program. You will not need to modify this file.

movielib. This directory contains the skeleton application. You will not need to modify these files, but you may want to take a look at them.

movies.txt. The text data for the movie database. You will not need to modify it. The program loads data from this file using your `MovieText` implementation.

The following sections describe how to read / modify these files to complete your application.

The Model

The model consists of the files in module `movielib.model`, plus files you must complete: `MyDatabase.py`, `Movie.py`, and `MovieText.py`.

The file `MyDatabase.py` contains a class `MyDatabase` that inherits from `movielib.model.Database.Database`. The `Database` class provides dictionary-like behavior for your library. The class has a data member called `ATTRIBUTES` which contains a name for each movie attribute. You will not need to redefine this data member in `MyDatabase`, unless you implement one of the extras (like loaning). The class also defines the following methods:

populate. This method takes a list of `Movie` instances and maps each movie's `upc` attribute to the instance. You will not need to override it in `MyDatabase`.

search. This method takes an attribute name (as a string) and a search string. The method returns a list of movies whose given attribute contains the search string. You must override this method in `MyDatabase` to provide the correct functionality. You need not do a fancy search. It's sufficient to return a match if the search string is a substring of the given attribute's value. Of course, if you've already written something fancier, that's OK.

sort. This method returns a list of movies in the database, sorted by UPC. You must override this method in `MyDatabase` to provide the correct functionality. If you chose to do the sort-by-attribute extra, then the method call can specify the attribute name by which to sort and return a list of movies sorted by that attribute name.

The file `Movie.py` contains an empty class `Movie`. You must fill in this class definition. You should be able to re-use most of the code you wrote for the previous assignment. The requirements for the `Movie` class are as follows:

The class must have a data member for each name in `Database.ATTRIBUTES` (e.g., `upc`, `title`, etc.).

Each data member must store the type described in the previous assignment (e.g., `year` is an `int`, etc.).

The class must have a `get` method for each attribute. The method must be named according to this template: `getAttribute` (so, it'll have methods `getUpc`, `getTitle`, etc.). The `get` method simply returns the value of the corresponding data member.

The class must have a `set` method for each attribute. The method must be named according to this template: `setAttribute` (so, it'll have methods `setUpc`, `setTitle`, etc.). This method takes a value for the data member. The method must validate this value using a regular expression as described in the previous assignment. If the value is valid, then the method assigns that value to the data member. If the value is invalid, the method must raise a `MovieException`.

The file `MovieText.py` contains a class `MovieText` that inherits from `movielib.model.MoviePersistence.MoviePersistence`. The `MoviePersistence` class loads movie data from and saves movie data to disk. The class constructor takes a file name from which to load and/or save movies. The class defines the following methods:

loadMovies. This method loads the movie data from disk. The method must convert the disk format to a list of `Movie` instances and return this list. You must override this method in `MovieText` to convert the text file to a list of `Movie` instances. You should be able to reuse almost all of your code from the previous assignment.

saveMovies. This method takes a list of `Movie` instances and saves them to disk. You must override this method in `MovieText` to convert a list of `Movie` instances to file format and save the file to disk. You should be able to reuse almost all of your code from the previous assignment.

The file `movielib.model.MoviePersistence.py` also contains a class `MoviePickle` that inherits from `MoviePersistence`. The Web application does not use this class, but you may examine it as an example of how to write a storage manager.

The file `movielib.model.MovieException.py` contains a class `MovieException` that implements a custom exception. Your program must raise this exception any time it discovers an error. The controller can catch this exception to print error messages. To raise this exception, you must import it using:

```
from movielib.model.MovieException import MovieException
```

then raise it using:

```
raise MovieException('error')
```

where `'error'` is the error message.

The View

The view is complete--you do not need to modify it. However, you do need to write a controller that uses the view, so it can be helpful to know what behaviors the view implements. The file `movielib.view.HTMLMovieView.py` contains a class `HTMLMovieView` that converts movie data to HTML. The class provides the following methods:

error. This method takes an error message string and returns an HTML encoding of that string. This method is probably the only view method you will need to call.

header. This method returns an HTML string for the beginning of an HTML document. You will not need to call it unless you are doing one of the extras.

footer. This method returns an HTML string for the end of an HTML document. You will not need to call it unless you are doing one of the extras.

displayAll. This method takes a list of movies whose data are in turn encoded as a list of attribute values. The method returns an HTML table encoding of the data. The method optionally takes a list of attribute names that correspond with the attribute values. You will not need to call this method unless you are doing one of the extras.

The Controller

The controller consists of the files in module `movielib.controller`, plus the file you must complete: `MyMovieController.py`. It also includes the file `moviedb.cgi`, which you do not need to modify.

The file `MyMovieController.py` contains a class `MyMovieController` that inherits from `movieLib.controller.MovieController.MovieController`. The `MovieController` orchestrates the program control flow between the model and the view. Its constructor takes a model and a view and stores them as data members. A `MovieController` instance also contains a data member `form` that corresponds to Web form data and a member `validated` that is `True` if the Web data have been validated. The class provides the following methods, only one of which you must override:

loadForm. This method stores Web form data into data member `form`.

validate. This method validates the user form. You must override its definition when you implement `MyMovieController`. The `validate` method accesses the controller's `form` data member and must implement the following behavior:

If the form is `None`, set the `validated` member to `False`.

If the form is empty, set the `validated` member to `True`.

If the form is not empty, then it may have a key `"sort"`, or it may have a key whose name matches any name in `Database.ATTRIBUTES`. If it has any other keys, set the `validated` member to `False` and exit the method. If the form has a key `"sort"`, then the key's value must be any string that is a member of `Database.ATTRIBUTES`. If the value is not a member of `Database.ATTRIBUTES`, then set the `validated` member to `False` and exit. If the form has a key that is a member of `Database.ATTRIBUTES`, then its value must be valid for the given attribute. In other words, if you were to pass the value to the appropriate `set` method, the `set` method wouldn't raise an exception. An invalid value causes the method to set the `validated` member to `False` and exit.

You must implement all this behavior for the `validate` method. The goal of `validate` is to allow URLs like these:

`http://.../moviedb.cgi`, `http://.../moviedb.cgi?sort=upc&year=1999`, `http://.../moviedb.cgi?stars=Clooney`

but to disallow URLs like these:

`http://.../moviedb.cgi?unknown=1`, `http://.../moviedb.cgi?year=abc`

display. This method prints HTML-encoded movie data. The user will be allowed to interact with the data by sorting or searching it. The controller automatically sorts and/or searches according to the user-supplied data. You do not need to modify this method. However, the method relies on your database sort/search methods, and on your validation method.

The file `moviedb.cgi` is the main program, and is the file that users will visit to use your Web application. This file contains a `main` function that creates an instance of `MyDatabase` and populates the database using an instance of `MovieText`. The function then creates an instance of `MyMovieController`, passing the database and a view instance. Finally, the program causes the controller to display the movie data.

Summary

The last page of this document contains a diagram that summarizes the application's module, file, and class relationships.

Advice

One partner can unpack the application and work on making sure it runs out of the box on the UTCS Web server. The other partner can work on the model. Once the application is running on the UTCS Web server, work together to finish the model. Test the model independent of the controller or view. When the model is done, test it with the rest of the Web application. If the model works, the application should display all the movies. You can use the `cgibt` module to debug your application by uncommenting the appropriate lines in `moviedb.cgi`. Finish by implementing and testing the validation on the Web server. Start early: there's not a lot of code to write but it takes a long time to figure out what must be written. When in doubt always ask questions on the discussion board or during office hours.

What To Turn In

Turn in all the files you modified. If you don't do any extras these files are: `MyDatabase.py`, `Movie.py`, `MovieText.py`, and `MyMovieController.py`. You must also turn in a project report in the file `report.txt`. Each submitted file should have both team

members' names and EIDs in a comment at the top of the file. Be sure to read the **Programming Assignment Overview** for instructions on how to prepare your report and submit your work. The *homework-name* for this assignment is: pa3.

Extras

- Add better error messages. Try to give the user information about why input validation failed.
- Add an HTML form that lets the user search and sort the movies. To do this, you'll probably want to extend class `HTMLMovieView` to create the form, then override method `display` in `MyMovieController` to output the form.

