



ABERDEEN COLLEGE

D76V 35

**Software Development :
Object Oriented
Programming**

Workbook 7 of 8

**Outcome 2 : Use encapsulation,
polymorphism and inheritance**

**Outcome 3 (part): Implement a solution
from an OO Specification**



Awarded for excellence

Higher Computing (TN3)
Engineering, Computing and Business Studies

The Penultimate Challenge!

In the last workbook we began to study the relationships between objects in an OO system, and saw how we could build small classes from bigger ones using the concept of inheritance. In this book we will revise and expand upon that knowledge. We will also see how to design simple OO systems from textual specifications, with a view to implementing the model using Java. In the next book, we will look at more complex systems, and revise how to produce the necessary documentation.

The Basics Of Design.

Case Study 1. The Fleapit Cinema.

The Fleapit is a small but popular cinema in town. It is, in fact, the **only** cinema in town, which is why it is so popular. Because it only holds 30 people and only shows one movie, twice a day, there is a large demand for seats. The seats are all the same and it costs £1 to see the film. There have been some ugly scenes in the foyer recently, so the management has decided that all seats have to be booked in advance. They can only be booked for that same day, however – bookings cannot yet be accepted for different days of the week. The theatre owners have approached us to design an OO system to cope with the bookings.

The first things we must ask ourselves are:- who are the users of this program, and what exactly does this program have to do?!

Remember that the program **user**, i.e. the human element of this system, is referred to as an **actor**.

The actor would have to be able to:-

- ❖ **Book** seats, specifying the performance time and number;
- ❖ **Enquire** as to the availability of seats;
- ❖ **Cancel** seats that were no longer required.

In all of these cases, the user will have to interact with the system, which will display, confirm or change state information.



Because we are creating the system as a series of classes, we must first identify the classes that we are going to define. We can do this by identifying **key nouns and noun phrases** and / or **key verbs and verb phrases**, preferably in that order. Let's look at the use case again.

The Fleapit is a small but popular **cinema** in town. It is, in fact, the **only** cinema in town, which is why it is so popular. Because it only holds 30 people and only shows the **one movie, twice a day**, there is a large demand for **seats**. The **seats** are all the same and it costs £1 to see the film. There have been some ugly scenes in the foyer recently, so the **management** has decided that all **seats** have to be booked in advance. They can only be booked for that same day, however – bookings cannot yet be accepted for **different days** of the week. The **theatre** owners have approached us to design an OO system to cope with the bookings.

The above could therefore be considered as classes – the Cinema, the Movie, the Twice A Day (i.e. performance) and the Seats. We are **not** interested in the foyer, the fights, the rest of the town or any combination thereof, because these are not likely to be of relevance to this program! Remember that this is firstly an exercise in **data abstraction** —we are trying to isolate the **essence** of the system and the classes it is likely to contain.

We could ask ourselves the following questions:-

Is it **Relevant**? Does it refer to an actor / user, or some real life object?

Is it a **Duplicate**? Has the object been listed twice, in different forms?

Is it **General**? Is it really an object in its own right, or just a different type of another object?

Is it too **Simple**? Could it be incorporated in another class?

Let us look at our possible classes in turn:-

Cinema. There is only one cinema object involved, so we don't really need a separate class for this. However, we *could* make this the top-level, or *Application* class, and call it *Cinema* as this is what the system is about! Question – would this same situation hold if we were dealing with a chain of cinemas?

Movie. This is not really relevant for this particular application – it *would* be if the Fleapit were a Multiplex, showing several films at once! So we can probably discount this on this occasion.

Twice A Day. In a sense, “x times per day” could be termed as “x number of *performances* per day,” which would give us a *Performance* class. There are three separate performances – each of which happen on a certain day of the week, and at a certain time. Do we need separate Day and Time classes?

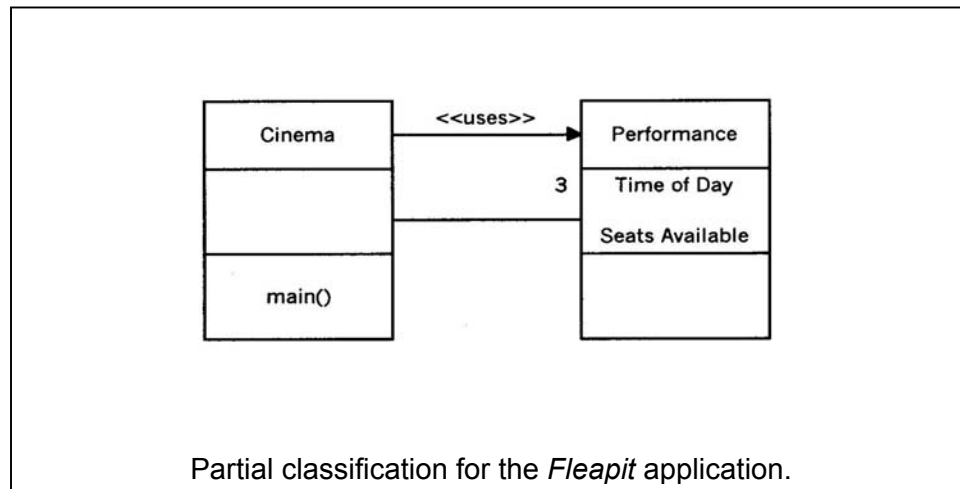
Management. The Management does not really form part of the program, but will *use* it – i.e. this is our *actor*. So this does not need a class of its own.

Seats. Seat could be listed as a simple integer, and so does not really merit a class of its own; we could just set up an array of seats in the *Performance* class. Would this still be the case, however, if there were different types of seats at different prices?

Days of the Week. Our system is not being written to cope with advance bookings, so we don't have to worry about this just yet. Would we need a separate *Day* class in an extended application?

Theatre. See Cinema.

Once we have defined any possible classes, we test for completeness – does the list of classes we have created, cover all the possible program requirements?



Use Cases.

Let's look again at what we expect of the program; the user should be able to :-

Book seats, specifying the performance time and number. The user specifies which performance and how many seats are required. The system checks that the seats are available – if so, it confirms the booking. If not, an appropriate message will be displayed.

Enquire as to the availability of seats. The user specifies which performance is required, and the system will display the number of seats available.

Cancel seats that were no longer required. The user specifies the performance and number of seats to be cancelled. The system then confirms the cancellation, and makes the seats available for re-booking.

The *Cinema* Class.

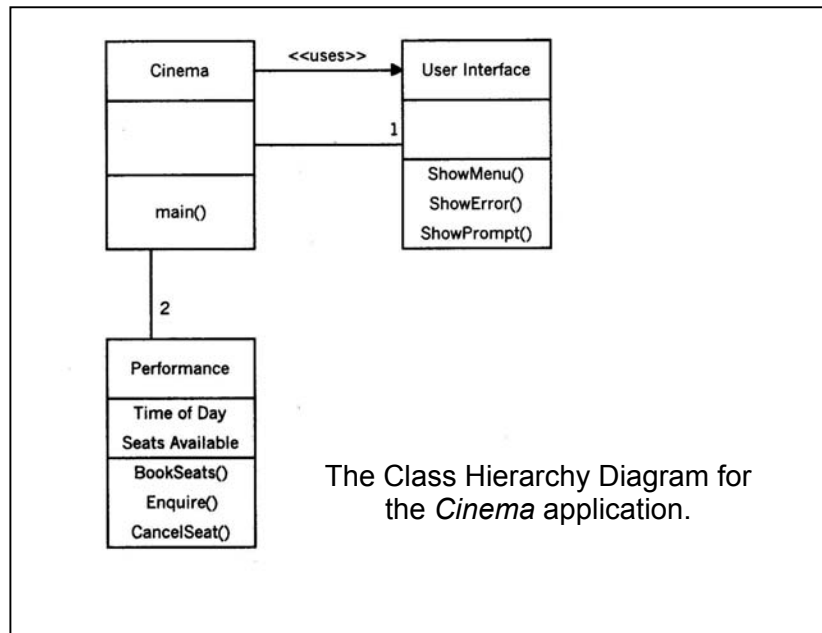
We already decided that the *Cinema* class was going to be the Top level Application, and this one will have the *main()* method in it. The *Cinema* class can also be used to create *performance* objects of the *Performance* class (in the same manner as we created all these *clocks* from the *Time* class!). The top level class is often used to create objects of major importance and “pull” all the other classes together, but doesn't really do much else, and for this reason it is sometimes called a *Class of Convenience*.

Like many programs, however, there is likely to be a lot of interaction between the actor / user and the electronic system. This will involve writing sizeable pieces of code in order to facilitate the input and output of data. Strictly speaking, this interaction should be between our user and the *Performance* class. It is not a good idea, however, to clutter up our class with a lot of irrelevant dialogue – especially as the user interface is something that is likely to be changed a lot during the development of the software. So it makes sense to have all the interface dialogues in a class of their own.

Remember, also, that Java classes are primarily designed for re-use. We may, eventually, want to re-sell the Fleapit Cinema program to Richard Branson, for use with Virgin Cinemas, in which case the fundamental program can remain the same but only the interface needs to be changed. Far better, then, to amend one very small class than completely re-write a large one!

Our final program will therefore consist of the following:-

- ❖ A Top level *Cinema* class
- ❖ A *Performance* class
- ❖ A *UserInterface* class – contains all the prompts, error messages and confirmations
- ❖ and any Java library classes, including the *TerminalInput* if you are using it.



Exercise 7.1 Implement the above classes with a view to reusing the *Cinema* and *Performance* classes.

Exercise 7.2 Write a different *UserInterface* to reuse the program. This time we are allocating 25 seats at the Auchterturra Teuchter Wifies Bingo Hall, for one Bingo sitting three times daily.

Case Study 2 - Thomas the Tank Engine.

This is a slightly more complex example than the one shown above.

Thomas the Tank Engine works on a branch line with a number of stations along it. The Fat Controller has decided that he needs a new OO system to maintain a timetable of arrivals and departures at Thomas' main station; the same timetable is followed every day. He would like to be able to query arrival times at the main station; query departure times for the journey to any other station; query the duration of the journey to any other station, and quote all times in hours and minutes. No journey will take more than a day to complete, and Thomas only runs during the day.

Exercise 7.3 Using the same technique as for the *Cinema* application, make a list of potential classes and examine each one on the basis of relevance, equivalence, generality and simplicity.



It should be a safe assumption that the program is going to interact with a human, and that the actor will be accessing the *Timetable* object. Like the *Fleapit* example, however, we may later wish to re-sell our program to another party, who may also use a *Timetable* class. Sodor International Airlines, for example, would not be too impressed with a program that began with a "Sodor Steam Railway Company" splash screen! So again, we are going to introduce a *UserInterface* class, which will act as a "helper class" to link *Timetable* and the actor.

For this system example, we are going to use *Timetable*, *Journey*, *Time* and *UserInterface* classes. Finally, we need a Top-Level class, which we are going to call *Railway*, to pull the whole thing together and start it off.

We also need to establish relationships between our classes. Some of these relationships are obvious, for example *Journey* will have variables showing the beginning and end points, and times of arrival and departure. *Journey* will also have a one-to-one relationship with the point of departure. *Timetable*, on the other hand, has a whole-part, one-to-many relationship with *Journey*. Finally, the top-level class *Railway* will have one interface and one timetable, so both of these will be one-to-one association relationships.

Exercise 7.4. Draw the UML classification diagram for the *Railway* application.

Designing the Classes.

Our requirements tell us something about the functionality, i.e. the methods needed, for the classes that reside at the top of the class hierarchy. We have to figure out for ourselves the services which the lower level classes must provide in order to enable the top level classes to perform their specified functions. Our *UserInterface*, for example, has to display a menu of choices, prompt the actor for a reply, then retrieve and display the results of the query.

We know that the program begins with the *Railway* class (*how* do we know this?), that sends the first message to the *UserInterface* to display the opening menu. From then on, the actor and program engage in meaningful transactions to request, supply, retrieve and display the appropriate data. The *UserInterface* will therefore need methods to :-

- ❖ *doQuery()*
- ❖ *getDepartureDetails()*
- ❖ *getArrivalTimes()*

How will this program work? Firstly, the actor supplies a question - e.g. the arrival time of a certain train. The *Timetable* object then sends a message to each *Journey* object in turn, checking whether this specific *Journey* matches the specified point of departure. (If the *Journey* objects are set up in an array or vector, the program will simply loop through it until it finds the data that matches the query.) Once we have established which *Journey* objects match the specified departure station, we can then send messages to the specified objects to get the arrival times associated with these *Journeys*.

We can now concentrate on the design of the particular method *doQuery()*, used by the class *Timetable* to send the message to *Journey*. Here's the pseudocode:-

```
doQuery(departureStation):
    find all journeys which match departure station
    if the number of matches is greater than 0 then
        construct an array of Strings of size equal to number of
        matches
        fill array with arrival times
        return array of arrival times
    else return empty array
```

Exercise 7.5. Complete the partial *Railway* application on the source diskette.

Case Study 3 – Radicol Radio.



DJ Dan works in a small radio station, and has been put in charge of a large archive of CDs. The radio station broadcasts mainly mainstream music (pop and rock), but there is also a Classical programme, an Indie programme (plays house and indie music) and the Golden Oldies show. Different DJs present the different shows, using the archives CDs, but nobody knows exactly what is in the archive as the inventory paperwork is somewhat shambolic! The boss of the radio station has therefore commissioned a study to find out a) what CDs the station has, and b) how they are actually used. This study may also be used at a later date to construct audience figure surveys, etc. that can be used by advertisers for specific targets.

Let us now consider the users of the archive. DJ Wolfgang presents Couthy Classics on Sundays from 3 pm until 6. His listeners tend to be older persons who write in requesting certain pieces of Classical music, and consequently his advertisers tend to be Life Assurance Companies and walk-in bath manufacturers. DJ Skunk, on the other hand, presents Flashpoint, the Indie show, between 7 – 8pm on Mondays to Fridays. His main advertisers are SupaSchnapps and Repeat Boutique, and naturally his choice of music is very different from Wolfgang's. He does not accept listeners' requests but makes up his own playlists.

Our third user is DJ Tony, who does Goalden Nuggets. This is broadcast from 9.00 am to 5.00 pm on Saturdays and includes the football results, so he cannot really accept listeners requests. His playlists are all tracks from the 60's and 70's. All the other shows comprise mainstream music and presented by DJ Dan, who does accept listeners' requests. The advertising base for both Tony and Dan can be anything from baked beans to dot coms.

Exercise 7.6. Define the candidate classes from the above textual model. Does your list agree with the sample solution?

Exercise 7.7. What should our users be able to do with this program? Who are the actors in the system? Think about how the menu of options would appear, and the use cases that are likely to be involved.

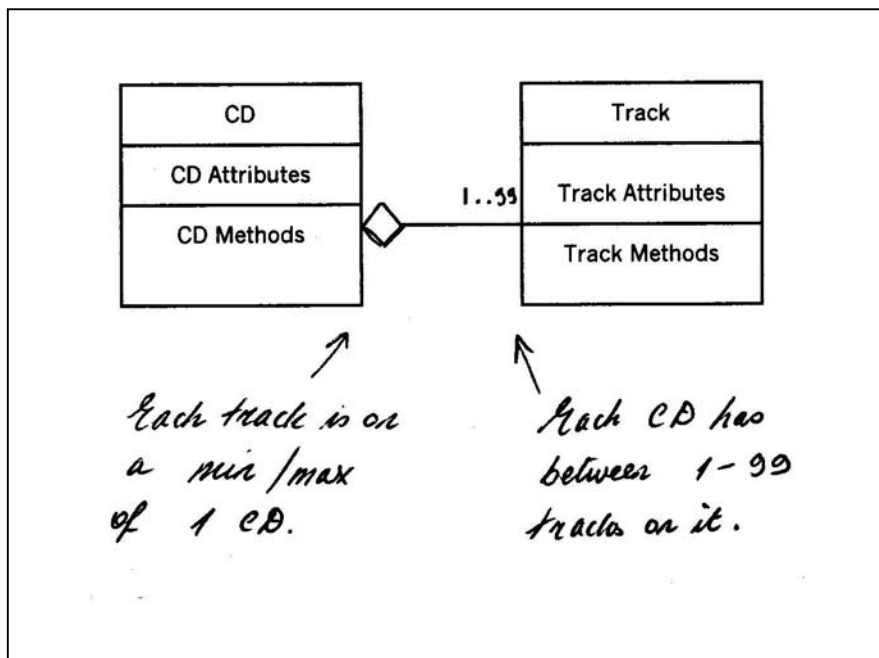
Exercise 7.8 We have already established that the station has a large collection of CDs, so it should be safe to assume that CD is going to be a class. What type of data, or **attributes**, would each CD object have?

Class: CD
Attributes: Name of CD Type of CD Artist Record Label Catalogue Number
Methods: CD methods go

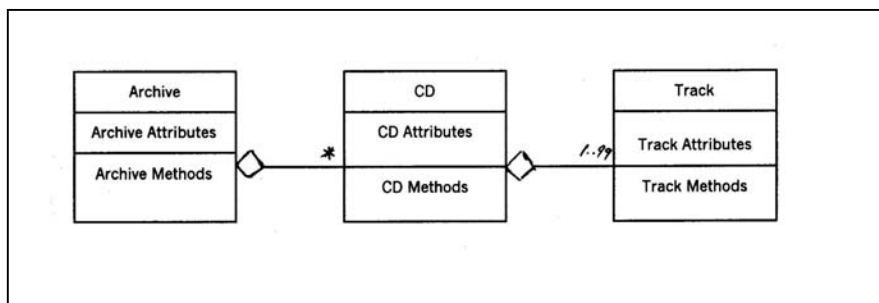
Firstly, let us look at the ATTRIBUTES of the CD Class. We should be able to set these up using Java equivalents – Name (String), Type (Integer) Artist (String), Record Label (String) and Catalogue Number .

Q - What is missing from this list of attributes?

A – Our CD class is too simple. Each CD will also have a number of tracks on it, and this is what DJ Dan (and the listeners) are interested in! So we can set up Track as another object. The CD class will have a whole-part, one-to-many relationship with Track.



By extrapolation (or more of the same, if you like!) we know that all the CDs are part of a larger **Archive** object, so we can put that in as well.



Exercise 7.9. What can we infer about the relationship between the Archive and CD classes, from the diagram above? Is *Archive* a suitable candidate for the *main()* method? If so, why?

Exercise 7.10. Study the Java source code for all three classes plus the top level (on the disk as version 1.0). Complete the above diagrams to include all the relevant attributes and methods. Compile and run the *Cdtest0* program and watch what it does - ensure that you are comfortable with what is actually happening, before proceeding to the next section.

TIP! If you don't quite follow how the methods work, print out the source code for all four classes. Get some coloured markers, and highlight what is being called from where. This will help you understand how the target, method and parameters all combine to form a message which is sent by one object and received by another.

Adding the Interface Class.

Like the earlier programs, we require our human user to interact with it in some meaningful way. Again, we will have a sizeable amount of screen prompts, error messages, directions etc. so to keep things tidy we will again introduce an Interface class, leaving any error handling to the *TerminalInput* class. Our actor can now enter, store, retrieve or amend the appropriate information.

A sample Interface Class - study the source code below, then for **Exercise 7.11** compile and run all of the amended classes in Version 1.1.

```
//the Interface class for the Radio program
//Christine Nyssen 07.10.2000

public class Interface
{
    public Interface()
    {
        //start with the default constructor for the class.
        TerminalInput input = new TerminalInput(System.in);
        //declaring the terminal input object and calling it input
    }

    public void mainMenu()
    //the opening menu for the program
    {
        System.out.println() ;
        System.out.println("          -----") ;
    };
    System.out.println("          *** RADICOL RADIO MAIN MENU ***") ;
    };
    System.out.println("          -----") ;
    };
    System.out.println() ;
    System.out.println("          1: Add a CD to the database"); ;
    System.out.println() ;
    System.out.println("          2: Quit"); ;
    System.out.println() ; ;
};
```

```
    }

    public int getMainChoice()
    //gets the choice from main menu
    {
        TerminalInput input = new TerminalInput(System.in);
        System.out.println() ;
        System.out.println() ;
        System.out.print("Enter Choice Here.....");
        return (input.readInt());
    }

    public char inputError()
    //standard error message for out of range input
    {
        TerminalInput input = new TerminalInput(System.in);

        System.out.println();
        System.out.print("!Input Error!  Press any key to continue...
");
        return (input.readChar());
    }
}
```

You will see that we have now added a menu (which so far only has one option in it) but our OO system is beginning to take shape!

Further Exercises. Add these to the basic Radicol Radio program.

Exercise 7.12. When creating a CD, ask how many tracks are on it. Set up an array of that size, then associate the track object array with the CD object. Add a menu item that displays the tracklist of any named CD, by simply scrolling through the array.

Exercise 7.13. Add a menu item to search for a specific track by name. Return the name of the CD that the track is on.

Exercise 7.14. Add a menu item to record listeners' requests, incorporating a "Request" class, containing attributes for the DJ, show and record requested. Note that if the listener asks for a request from a DJ who does not accept them, an appropriate message should be displayed and the request rejected. Add a counter to the Track class to note how many times a specific record has been requested. Finally, add a menu item to display the number of times any particular track has been played.

Publishing Your Classes.

As programmers, we “own” the code we write, which is in turn a valuable item. We do not normally wish to publish it for free (the term for this is “open source”) – all we want to do is sell the program, without the buyer being able to “hack” into it and see how it actually works. Java classes also work a bit like this, in that we want other parties to be able to use the class, but not necessarily to be able to see the internal workings of it! This brings us back to the OO principle of *encapsulation*.

You can see this principle in action for yourself, by downloading any of the free Java applets that are available on the Web. Usually all that you can obtain are the `.class` bytecode files; you rarely get the `.java` ones which contain the actual source code!

In order to be able to use other designers’ Java classes, however, we do need to have certain items of information, so that we know what the class can do. This is known as the **external** or **published interface** of the class.

This is the published Interface of the Java **Clipboard** class:-

Class Clipboard

[java.lang.Object](#)

|
+--[java.awt.datatransfer.Clipboard](#)

public class **Clipboard**

extends [Object](#)

A class which implements a mechanism to transfer data using cut/copy/paste operations.

Field Summary

Protected Transferable	contents
Protected ClipboardOwner	owner

Constructor Summary

Clipboard (String name)	
Creates a clipboard object.	

Method Summary

Transferable	GetContents (Object requestor) Returns a transferable object representing the current contents of the clipboard.
String	getName () Returns the name of this clipboard object.
void	SetContents (Transferable contents, ClipboardOwner owner) Sets the current contents of the clipboard to the specified transferable object and registers the specified clipboard owner as the owner of the new contents.

Methods inherited from class [java.lang.Object](#)

[clone](#), [equals](#), [finalize](#), [getClass](#), [hashCode](#), [notify](#), [notifyAll](#), [toString](#), [wait](#), [wait](#), [wait](#)

Field Detail

owner

protected [ClipboardOwner](#) owner

contents

protected [Transferable](#) contents

Constructor Detail**Clipboard**

public **Clipboard**([String](#) name)

Creates a clipboard object.

Method Detail**getName**

public [String](#) **getName**()

Returns the name of this clipboard object.

setContentts

public void **setContentts**([Transferable](#) contents,
[ClipboardOwner](#) owner)

Sets the current contents of the clipboard to the specified transferable object and registers the specified clipboard owner as the owner of the new contents. If there is an existing owner registered, that owner is notified that it no longer holds ownership of the clipboard contents.

Parameters:

content - the transferable object representing the clipboard content

owner - the object which owns the clipboard content

getContents

public [Transferable](#) **getContents**([Object](#) requestor)

Returns a transferable object representing the current contents of the clipboard. If the clipboard currently has no contents, it returns null. The parameter Object requestor is not currently used.

Parameters:

requestor - the object requesting the clip data (not used)

Returns:

the current transferable object on the clipboard

The published details of a class usually include the name of the class, its **public data members** and its **public methods**. Any data member or method that has been written with **private** visibility, is only accessible from within the class, and so never appears as published information.

Exercise 7.15. Log on to the Java Documentation website and study the published interfaces of some of the classes you are already familiar with, for example *String*. Now write the published interface of the *TerminalInput* class in a similar fashion.

Exercise 7.16. Publish the interfaces of the CD and Track classes.

Exercise 7.17. FTP the *Advertiser.class* file from the disk into a new directory. Using the published documents, set up an array of three *Advertisers* using different constructors. Display the information.

Exercise 7.18. Do this with a friend. Obtain any compiled *.class* file and the relevant published interface from one of your friends. Now implement a test driver for the class, using *only* the compiled bytecode and published interface documents.



Congratulations! You have now finished Workbook 7!