

G52OBJ

Assessed Coursework

2009/2010

Revision: 1.00

This coursework is the only coursework element and the total marks are out of 100. However, your mark on this coursework will be scaled to a weighting of 25% of the total mark for the entire module. When completed, the coursework should be handed in at the School Office. Read carefully the instructions below about submitting online versions of your solutions. The *final* date for submission is **Friday 14th May at 4pm**.

Introduction

This coursework is split into two questions. The first question concerns creating an object-oriented implementation of the classic game of *Top-Trumps*. Below, you will find an analysis of the game and how it is played. For this coursework, you are to break this analysis into a set of classes and the methods those classes provide. You should then implement those classes in Java so that users can play a game of *Top-Trumps*. The code to initialise the deck of cards from a CSV file is provided for you.

The second question concerns using *Reference Counting* to manage the memory of a simple program, in the coursework support files you will find an implementation of a simple program that presents the user with a simple Maze which they can walk around by giving commands at the command line. This program is feature-complete, but does not attempt to manage its memory at all. Your task for the first question is to take the existing C++ programme and modify it so that it uses Reference Counting to ensure that objects are freed correctly.

Question One

Top-Trumps^{*} is a classic British card game from the late-1970s. The game is played with a deck of cards related to a theme (such as a TV programme characters, cars, military hardware etc.) and each card in the deck contains a list of data about the item. A typical Top-Trumps style card is shown below — taken from a set based around different fonts[†].

Helvetica	
Price	£464
Year	1957
Legibility	8
Weights	51
Cuts	06
Rank	01

A typical Top-Trumps card

The deck is dealt to each player in the game who then has a pile of cards which are kept in this order for the duration of the game. There must be at least two players. The game begins with the

* See http://en.wikipedia.org/wiki/Top_Trumps for more details...

† Available from <http://www.face37.com/work/24/type-trumps...>

player sitting to the dealer's left. He or she turns over the *top* card from their pile and selects a category from the card and reads out its value. The other players then turn over their *topmost* card and read out the value for the *same* category from their cards. This round is run by whoever has the highest value for that category. The winner takes all the cards played in that round and adds them to the bottom of their pile. The winner of that round then becomes the current player. In the case of a draw, the cards are put to one side and another round is played. When that round is won, the winner takes all the cards put aside, as well as the winning hand. The game continues until one person is left holding all the cards (or, in other words, all the other players have zero cards). If a player has zero cards, they leave the game.

1.1. Computer version

Using the description of the game above, plus the details that follow you should create an Object-Oriented version of this game. You will need to break the analysis of the Top-Trumps game down into a set of classes, objects and methods and then go and implement it in Java.

We have provided a starting point of the game for you. This is the **Director** part of the **Builder** pattern. It will read in a CSV file passed on the command line to initialise the game. Since we do not know how you will implement your game (or your deck), we have defined an interface `IDeckBuilder` which your deck builder must implement. This interface contains three methods:

```
public interface IDeckBuilder
{
    public void newCard(String title);
    public void addProperty(String name, int value);
    public Deck getDeck();
}
```

These will be called in the following fashion. The method `newCard` will be called each time a new card is to be created. This card will then become the *current card* and is added to the deck. Any following calls to `addProperty` will add that property to the current card. This method, then, is used to create all the categories on the cards.

Once the Deck has been created, the `getDeck` method will be called to get access to the Deck object and this will be then passed to a new Game object, on which the method `playGame` will be called. You are responsible for providing the implementation of this class and method, alongside any others you require.

1.1.1. Playing the Game

For this game, we shall assume there are only two players, a human player and a computer player. The human player will always start first. Once `playGame` has been called the program should enter a loop which will work as follows. The top card will be turned over and its contents printed on the screen (assuming the example card above) e.g.:

```
Top Card is Helvetica:
```

```
Categories:
1. Price: £464
2. Year: 1957
3. Legibility: 8
4. Weights: 51
5. Cuts: 06
6. Rank: 01
```

```
Please select a category to compete with:
```

The user can then enter the number of the category they wish to play with. The program will then compare the value of this category against the value on the top card of the computer player. For the moment, the winner is the person with the highest value, although it is possible that this

will change in a later version and we should always write software to be amenable to change. Assuming that the round is won, the cards are then added to the bottom of the winner's pile and that person starts the next round. If it is a draw, the cards are put to one side, and the same player starts the next round. The program will continue until the game is won at which point the program will exit.

When a round is played, the computer will output a message on screen, like the following, to inform all players of the result:

```
<Winner> has won this round with card <Card> \
which scored <points> in category <Category>.
```

Where <Winner> is either Computer, or Human (depending on the player who won), <Card> is the title of the card, <Points> the amount of points on the card for category <Category>.

In the case of a draw, the computer will output the message:

```
Players have drawn on <Category>, playing next card.
```

Note: These messages should appear all on one line—it is split in the example above because it is too long to fit on the page otherwise!

1.1.2. Computer Player

The computer player is very simple and only uses a very simple strategy (although again this could change in the future). It always picks the category on the card with the highest value to play. When it is the computer's turn, the program will output:

```
Top Card is Helvetica:
```

```
Categories:
1. Price: £464
2. Year: 1957
3. Legibility: 8
4. Weights: 51
5. Cuts: 06
6. Rank: 01
```

```
Computer has selected category <Category>
```

Where <Category> is whatever category was selected by the simple strategy described above. The program will then work out who has won in exactly the same fashion as for a human player.

1.2. Allocation of Marks

The marks for this piece of coursework will be allocated as follows: 70% of the mark for this question is for the quality of your design, how well it follows the design principles we have seen in the course, and makes use of appropriate design patterns, 30% of the mark will be based on how well you have implemented your design.

Remember the majority of the marks are awarded for a good OO-design, not a feature-complete implementation of *Top-Trumps*. Therefore, you should prioritize your time towards the design and rather than trying to code up the most efficient and perfect implementation.

[75 marks]

Question Two

Within `cswk.zip`, there is a folder called `Maze`. This contains a fully-working C++ implementation of a simple *Maze* that the user can move around by entering commands (such as `north`, `south`, `east`, and `west`) in the fashion of an old-style text-based adventure game. More details about the design of this program can be found in the Appendix.

For this question, you are expected to modify this implementation of *Maze* so that it uses *reference counting* to manage the objects' lifetimes. All the objects already inherit reference counting functionality from a base-class of `CRCObject` however no calls are made to `Retain()` and `Release()`. This means the reference counts never get updated and so the objects never get destroyed. Your task then is to add these calls to the program at the appropriate points.

The implementation of `CRCObject` provided also maintains a global count of references which is printed out when the program ends. You can use this to check whether you have got the reference count calls balanced. If you have, then the global reference count should be zero when the program quits.

Note: The code provided is designed to be compiled and run on the school's UNIX machines and your answer will be marked on these machines. You may wish to use other compilers while working on this problem but you should ensure that it compiles correctly on the UNIX machines before submission.

[25 marks]

2. Deliverables (or what you have to do)

The deliverables for this coursework are as follows:

- Source code for your implementation of the Top-Trumps program
- A short description (no more than 1000 words) of the classes you have created describing their function, what their methods do and how they relate to each other.
- The source code to your modified *Maze* programme that correctly uses reference counting to ensure objects are destroyed correctly.

You should print out a copy of your report, along with listings of your classes source code and hand them into the school office. In addition, we will scoop up electronic copies of your source code, so please ensure that all files are in a directory called `g52obj` within your `Private` directory.

Note: As you should by now be fully aware, UNIX is case-sensitive and so please ensure your directories are named exactly as specified.

If you have any questions about the coursework, then feel free to email me at `srb@cs.nott.ac.uk`, or drop into the lab sessions (we'll run labs right up until the closing date where possible).

APPENDIX

The Maze is represented as series of Rooms (we assume that there are no more than 256 rooms in total). Each Room has a room identifier (an integer between 0 and 255) and a description (a simple string) the values of which are set when the room is created. A Room is made up of four walls, a north wall, a south wall, an east wall and a west wall. These walls can either be solid brick walls, or a wall with a door in it. Doors link two rooms together and can be passed through to get from one room to the other (in other words, you can enquire of a door what room is on the other side of it). Doors can only link the south wall of one room to the north wall of another, or the east and west wall of each room. By default, a room has four solid walls in each of the positions. If a door is created linking two rooms together than the respective walls in each room are replaced by a wall with a door.

A player starts off in the room with the identifier zero. The player can be told to move north, south, east or west and if the room contains a door in the wall in that direction (i.e. if the player is told to go east, the east wall must contain a door) then the player moves through the door and into the room on the other side. The player cannot walk through solid walls.

Common Interfaces

To simplify the implementation, the user interaction code (it can be found in the supplied `main.cpp`) is separated from the rest of the implementation by a two interfaces. Note that the interfaces derive from `CRCObject` so that the implementing objects gain reference counting functionality.

CMazeBuilder class

The `CMazeBuilder` class is used to construct a maze. It defines three methods:

```
void BuildRoom(int rid, string desc);
void BuildDoor(int direction, int r1, int r2);

IMaze *GetMaze();
```

The first of these is called whenever we add a new Room into the maze. The integer `rid` is the room's identifier and is used by `BuildDoor` to describe which rooms are linked together by this door.

`BuildDoor` builds a door between two rooms, specified by the integer values `r1` and `r2`. The `direction` parameter specifies whether the door connects the north of room `r1` with the south of room `r2`, or the east of room `r1` with the west of room `r2`. A value of 0 implies a north-south connection, while any other value implies an east-west connection.

The final method `GetMaze` returns an `IMaze` pointer and is called once the maze has finished being created.

IMaze interface

The `IMaze` interface is the simplest of the interfaces and supports only one method:

```
virtual IPlayer *GetPlayer() = 0;
```

The purpose of this method is simple, it is used by the user interface code provided to get access to the player object (or at least the player object's `IPlayer` interface) so that it can move the player object around the maze under the control of commands from the user.

IPlayer interface

The `IPlayer` interface handles movement around the maze and provides the following methods:

```
virtual string GetEnvironment() = 0;  
virtual bool GoNorth() = 0;  
virtual bool GoSouth() = 0;  
virtual bool GoWest() = 0;  
virtual bool GoEast() = 0;
```

The first method `GetEnvironment` returns a string containing a description of the room that the player is currently in. This description is just the description that was passed to the room object when it was created.

The rest of the methods move around the maze in the various directions. They cause the player to move into the next room in that direction (assuming there is a door for the player to walk through!). If there is a room for the player to move into, the player's position is updated to be in the room on the other side of the door and the method returns `true`. Otherwise, the player stays in the same room and the method returns `false`. The return value from these methods is used by the provided UI code to print out the message saying the player cannot walk through walls.

Loose coupling

As you should have noticed, once the maze has been created (and the `IMaze` pointer obtained) all communication between your code and my code is via `IPlayer` interface. This is a classic example of good OO-design, the code handling the user interaction is loosely couple from the implementation of the logic. This means we can vary the implementation of the maze (as is happening here, with each student providing their own implementation) without having to change any of the UI code at all.