Software Engineering — Lab 1

Uppsala University

Spring 2014

Introduction

In this first block well refresh our object oriented skills and dive into concepts such as constructors, public and private membership, interfaces and inheritance as well as a couple of more technical features such as threads and GUIs.

1 A Simple Ticket Machine

The purpose of this first exercise is to get you up to speed. Familiarizing ourselves with the C# language, Visual studio environment and hopefully refresh your object oriented memory. In this exercise were building a very simple ticket machine. Think of this machine as an automated ticket clerk - or a parking meter where you pay upfront. We put money in to the meter, and when weve put enough money, then we press buy ticket. The machine swallow the money and spits out a ticket.

We will implement this ticket machine as a class, that expose four simple methods to the outside world: (1) InsertMoney takes a number as a parameter and you can think of this action as putting coins into the machine. The balance in the machine is updated. However we can at any time call (2) RefundMoney (think - press the return money/undo button) and the machine will return our money and reset the balance. The (3) PrintTicket method prints us a ticket given we have put enough money into the machine. Lastly (4) GetTicketPrice simply displays the price of a ticket.

1.1 An interface for the ticket machine

The interface for the TicketMachine is provided below. Either open the Visual Studio solution for this lab (which already contain the file for this interface) or create a new Console Application project and create a new file. In either case, make sure your interface contain the following code:

```
"ITicketMachine.cs"

public interface ITicketMachine {
  void InsertMoney(int amount);
  void RefundMoney();
  void PrintTicket();
  int GetTicketPrice();
}
```

1.2 Creating the class

Lets now create the TicketMachine class. Create a new class called TicketMachine.cs and tell it to implement the interface we just created:

```
"TicketMachine.cs"

public class TicketMachine : ITicketMachine
{
    ...
}
```

Try building the application at this moment and notice why it fails. We have yet not implemented all the members of the interface we created earlier.

For now, lets not concern ourselves with what the methods must do, but lets just add them so we can get an application that actually builds.

```
"TicketMachine.cs"

public class TicketMachine : ITicketMachine
{
  public int GetTicketPrice(){
    throw new NotImplementedException ();
  }

  public int GetBalance(){
    throw new NotImplementedException ();
  }

  public void InsertMoney(int amount){
    throw new NotImplementedException ();
  }

  public void PrintTicket(){
    throw new NotImplementedException ();
  }

  public void RefundMoney(){
    throw new NotImplementedException ();
  }
}
```

1.3 Command line interface

Ok, lets now create a CLI so that we can interact with our TicketMachine class. You should already have a file called Program.cs. Copy and paste the following code into this file. Have a look at the contents of this code and discuss it with a lab partner.

```
"Program.cs"

class Program
{
  private static ITicketMachine TM;

  public static void Main ()
  {
    TM = new TicketMachine (200);
```

```
10
11      // Display menu once
12      DisplayMenu ();
13
14      // Start the CLI
15      string userInput = WaitForUserInput ();
16      while (userInput != "0"){
17        HandleUserInput(userInput);
18        userInput = WaitForUserInput();
19      }
20    }
21
22    private static string WaitForUserInput(){
23      Console.Write("Enter action (1-5) and press RETURN: ");
24      return Console.ReadLine();
25    }
26
27    private static void DisplayMenu(){
28      Console.WriteLine("- - - - MENU - - - -");
29      Console.WriteLine("0: Exit");
30      Console.WriteLine("1: Show this menu");
31      Console.WriteLine("2: Insert money");
32      Console.WriteLine("3: Refund money");
33      Console.WriteLine("4: Print ticket");
34      Console.WriteLine("5: Show ticket price");
35      Console.WriteLine("- - - - - - - - - - -");
36    }
37
38    private static void HandleUserInput(string userInput){
39      switch(userInput)
40      {
41      case "1":
42        DisplayMenu();
43        break;
44      case "2":
45        Console.Write("Enter the amount you wish to put in (SEK): ");
46        int amount = Int32.Parse (Console.ReadLine());
47        TM.InsertMoney (amount);
48        break;
49      case "3":
50        TM.RefundMoney();
51        break;
52      case "4":
53        TM.PrintTicket ();
54        break;
55      case "5":
```

```
      int price = TM.GetTicketPrice ();
      Console.WriteLine ("# The price of a ticket is: " + price + " SEK");
      break;
    default:
      Console.WriteLine ("# Invalid input");
      break;
    }
  }
}
```

You should now be able to run the project and get the following output in the console.

```
- - - - MENU - - - -
0: Exit
1: Show this menu
2: Insert money
3: Refund money
4: Print ticket
5: Show ticket price
- - - - - - - - - - -
Enter action (1-5) and press ENTER:
```

However if you do make one of the choices above, a NotImplementedException is thrown. So lets implement the actual body of the TicketMachine class.

1.4 Constructor and properties of the TicketMachine

So, lets first make sure that the ticket machine knows how much a ticket is. Inject the price as an integer through the TicketMachines constructor and add a private class variable that can hold the price. While were at it. Lets also add the balance so that we can keep track of how much a user has inserted into the ticket machine at any given time. Like so:

```
"TicketMachine.cs"

public class TicketMachine : ITicketMachine
{
  private int price, balance;

  public TicketMachine (int ticketCost)
  {
    this.price = ticketCost;
    this.balance = 0;
  }
  ...
```

1.5 A runtime example - your turn!

So, now its your turn to implement the bodies of the methods of the ticket machine class. Below is a sample of what the application execution should look like when you are done.

```
- - - - MENU - - - -
0: Exit program
1: Show this menu
2: Insert money
3: Refund money
4: Print ticket
5: Show ticket price
- - - - - - - - - - -
Enter action (1-5) and press RETURN: 5
# The price of a ticket is: 200 SEK
Enter action (1-5) and press RETURN: 2
Enter the amount you wish to put in (SEK): 150
# Current amount: 150 SEK
Enter action (1-5) and press RETURN: 4
# The price of a ticket is 200 SEK, but you've only ins
Enter action (1-5) and press RETURN: 2
Enter the amount you wish to put in (SEK): 25
# Current amount: 175 SEK
Enter action (1-5) and press RETURN: 4
# The price of a ticket is 200 SEK, but you've only ins
Enter action (1-5) and press RETURN: 3
# Refunding: 175 SEK
# Current amount: 0 SEK
Enter action (1-5) and press RETURN: 2
Enter the amount you wish to put in (SEK): 300
# Current amount: 300 SEK
Enter action (1-5) and press RETURN: 4
* * * * * *
* TICKET
* 200 SEK
* * * * * *
# Your change is 100 SEK
Enter action (1-5) and press RETURN:
```

Firstly, dont sweat it! We've already built a functioning command line interface so all you need to do is to instantiate the TicketMachine correctly and then implement the members of the TicketMachine class. Below is some pseudo-code to get you started.

```
1 "TicketMachine.cs"
2
3 public int GetTicketPrice(){
```

```
4     // return the price of a ticket
5   }
6
7   public int GetBalance(){
8     // return the amount of currently inserted money;
9   }
10
11  public void InsertMoney(int amount){
12    // update the amount of inserted money
13    // print the current balance using Console.WriteLine
14  }
15
16  public void PrintTicket(){
17    // if enough money is inserted,
18    //    print the ticket
19    //    and print the change
20    // if not enough money is inserted,
21    //  print an error message
22    //    displaying the price and currently inserted money
23  }
24
25  public void RefundMoney(){
26    // print the amount of money refunded
27    // reset the current amount and print it
28  }
```

2 More Ticket Machines

So your ticket machine is fully functional. Yay, celebrate! However, lets now look at how much abstraction weve achieved with our current implementation of the ticket machine. Does our ticket machine allow different prices, or other payment schemes? Lets try it out.

2.1 Altering the price

Try instantiating the ticket machine with a different price. Such as:

```
"Program.cs"

...
public static void Main ()
{
  TM = new TicketMachine (1020);
  // or maybe even
  TM = new TicketMachine(40500);
...
```

We could of course even alter the price at **runtime**. Try this out:

```
"Program.cs"

...
public static void Main ()
{
  Console.Write("How much is a ticket (SEK): ");
  int price = Console.ReadLine();
  TM = new TicketMachine (price);
...
```

2.2 The power of programming towards interfaces

Now lets look at the mighty power of programming towards interfaces. We will be creating a donation machine with slightly different behavior. Create a new file called DonationMachine.cs and paste in the following class.

```
"DonationMachine.cs"

public class DonationMachine : ITicketMachine
{
  int price, balance, total;

  public DonationMachine (int minimumDonation)
  {
    this.price = minimumDonation;
```

```
    this.balance = 0;
    this.total = 0;
  }

  public int GetTicketPrice(){
    return this.price;
  }

  public int GetBalance(){
    return this.balance;
  }

  public void InsertMoney(int amount){
    if (amount >= this.price) {
      this.balance += amount;
      Console.WriteLine ("* * * * * * * * * * * * * * * *");
      Console.WriteLine ("* THANK YOU FOR YOUR DONATION :)");
      Console.WriteLine ("* " + this.balance + " SEK");
      Console.WriteLine ("* * * * * * * * * * * * * * * *");
    }  else {
      Console.WriteLine("# The minimum donation is: " + this.price);
    }
  }

  public void PrintTicket(){
    Console.WriteLine ("* * * * * * * * * * * * * * *");
    Console.WriteLine ("* THIS MACHINE HAS COLLECTED");
    Console.WriteLine ("* Total: " + this.total + " SEK");
    Console.WriteLine ("* Thank you everyone :)");
    Console.WriteLine ("* * * * * * * * * * * * * * *");
  }

  public void RefundMoney(){
    Console.WriteLine ("# Sorry, no refunds");
  }
}
```

Make sure you now instantiate the DonationMachine instead of the TicketMachine from the main class. As below:

```
"Program.cs"

...
public static void Main ()
{
  TM = new DonationMachine (1020);
...
```

Run the application! Think about how little modification we had to do to the command line interface (Program.cs). The only thing we changed was the hard-coded instantiation of the class. Furthermore, this problem of instantiation is something that we will discuss further when looking at design patterns.

2.3 A parking machine - your turn!

Now lets try to build a parking meter machine without changing anything, except which class were instantiating in Program.cs like so:

```
"Program.cs"

...
public static void Main ()
{
  TM = new ParkingMachine (20);
...
```

And also creating the class, as follows:

```
"ParkingMachine.cs"

public class ParkingMachine : ITicketMachine
{
  ... write your definition here ...
}
```

While the parking meter machines have different behaviors in different places this particular machine works just like the TicketMachine in the first assignment. Except for the following few things.

- The price is the value of 10 minutes of parking
- So when the user asks for a ticket the ticket must show how many minutes of parking the user receives
- Parking time can only be in full multiplications of 10 minutes, the rest the user will receive as change.
- So, if the price of a ticket is 5 SEK, and the user inserts 13 SEK, the parking time would be 20 minutes, and the change 3 SEK.

2.4 When you are done...

Consider again the power of interfaces. While the above examples might be slightly contrived, take a minute to consider the implications of adding a class such as the one below.

```
public class ExternalTicketMachine : ITicketMachine
{
  private int price, balance;

  public ExternalTicketMachine ()
  {
    this.price = // make a web request to get the price of a ticket
    this.balance = 0;
  }

  public int GetTicketPrice(){
    // make a web request to get the price of a ticket
  }

  public int GetBalance(){
    // return current balance
  }

  public void InsertMoney(int amount){
    // increase current balance
  }

  public void PrintTicket(){
    // make a web request to an external service
    // sending some user details and the current balance
    // returning the ticket recieved by the web request call
  }

  public void RefundMoney(){
    // refund money
  }
}
```

When we are programming to interfaces we are more prepared for change.

3 Basic GUI coding and Event Handling

In this assignment, you should build a basic desktop application using Microsoft Visual Studio. The assignment should be solved using the programming language C#.NET. The course literature McConnell (2009) primarily shows code examples in VB.NET and C++. It is up to you to and good C# tutorials on Internet.

a) The application should be a windows application consisting of one form. On that form, place one multiple-row textbox component, one listbox with dummy content (e.g. `"line1"`, `"line2"` et cetera) and one single-row textbox component. When something is written in the single-line textbox and `ENTER` is pressed, you should take the selected value from the listbox (e.g. `"line 1"`, 14 merge it with `" : "` and the content of the single-row textbox. Then you should add the result string to the multiple-row textbox.

b) Add a button on the form with the text `"Store message"`. When the button is clicked, the content of both text boxes should be cleared.

c) Add another button on the form with the text `"Start simulation"`. When the button is pressed, you should start a new thread. We will discuss the concept of threads further in the lab sessions. The thread should pick a message from a random list of messages, and add it to the multiple-row textbox. The idea is that the thread simulates 'real' chat messages.

By solving (a) to (c), we build a 'dummy prototype' for the client side of a chat system. In assignment 2, we will elaborate on the application to provide it with real chat functionality. However, in this assignment, we concentrate on building a GUI and some test functionality to simulate chat messages. In the lab sessions, we will discuss how to structure the code to make it as easy as possible to make the transfer from dummy messages to 'real' chat functionality.

Work in pairs. You are completely free to discuss with other groups, as long as you are able to understand and motivate your solution in the examination seminar.

3.1 Chat system

In this exercise, you have been given a partially completed solution for a full chat system. Both the client project and the server project are incomplete and contain errors. jointly build a chat server in the lab room. Your mission is to complete this solution and make it a working chat system. If you wish to use your chat client GUI created in the last exercise you are most welcome to do so.

4 Iterator

This exercise consist of the following three steps.

1. Implement the phone book examples from the notes.
2. Refactor the find method to throw an exception instead of returning -1
3. Develop a class diagram for the iterator.
4. Implement the iterator pattern for the phone book, populate the phone book and iterate over it, printing its contents.