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.

Deliverables

This is a summary of the deliverables for this block.

Exercise 1 – Ticket machines

The completed `Visual Studio` solution. One project with full implementation of the following classes/interfaces:

1. `ITicketMachine`
2. `TicketMachine`
3. `DonationMachine`
4. `ParkingMachine`

Exercise 2 – Chat system

A working chat system consting of two (potentially three depending on if you put the helper methods in a separate project) projects. All in one `Visual Studio` solution.

Exercise 3 – Iterator

1. The completed `Visual Studio` phonebook solution.
2. The iterator diagram.

1 Ticket Machines

The purpose of this first exercise is simply to get up to speed. Familiarize ourselves with the C# language, Visual studio environment and refresh our knowledge in object oriented programming.

We're building ticket machines to illustrate the power of programming towards interfaces rather than towards implementations. We'll start off with a very simple ticket machine, and later move on to more complex ones.

1.1 Simple Ticket Machine

In this exercise were building a very simple ticket machine. You can think of the ticket machine as an automated ticket machine at the cinemas. You insert money, press ok, and get a ticket.

1.1.1 An interface for the ticket machine

Naturally we'll build thins ticket machine as a class. The class will be called `TicketMachine` and it will contain a number of public methods. Let's describe the class using a `C# interface`.

```
interface ITicketMachine {
  string InsertMoney(int amount);
  string ShowPrice();
  string ShowBalance();
  string RefundMoney();
  string PrintTicket();
}
```

Step 1) Open up `Visual Studio` and create a new `C#` solution. Make sure you choose `C#` as the language, and not `Visual Basic` or any other language.

Step 2) Add a new interface file to your project. Call it `ITicketMachine` and save it as `ITicketMachine.cs`.

Step 3) Make sure your interface looks like the code example above.

1.1.2 Implementing the interface

Lets create the actual class. The ticket machine that will implement the interace.

Step 4) Create a new class called `TicketMachine.cs` and tell it to implement the interface we just created. Like so:

```
class TicketMachine : ITicketMachine
{
}
```

Step 5) Build the project (by pressing `F5`) and contemplate why it won't build.

To make sure the application will build. Let's actually implement the members of the interface in the concrete ticket machine class.

Step 6) Open the concrete ticket machine class and right-click the name of the interface in the class definition. Then select `Implement Interface > Implement Interface`. Or simply implement all of the members of the interface manually.

```
class TestMachine : ITicketMachine
{
  public string InsertMoney(int amount){
    throw new NotImplementedException();
  }
  public string ShowBalance(){
    throw new NotImplementedException();
  }
  public string ShowPrice(){
    throw new NotImplementedException();
  }
  public string RefundMoney(){
    throw new NotImplementedException();
  }
  public string PrintTicket(){
    throw new NotImplementedException();
  }
}
```

Superb! Now we, at least, have an application that builds.

1.1.3 Command line interface

So, now we have the "shell" for a class that will act as the ticket machine. But we need a way to interact with the ticket machine. I.e. we need a class that will use the ticket machine class.

For the sake of simplicity you've been provided with a command line interface (CLI) class that you can download here.

In your main `Program.cs` class you can then interact with the `TicketMachineInteractor` like so:

```
class Program{
  public static void Main(){
    // Instantiate a ticket machine
    ITicketMachine tm = new TicketMachine();

    // Instantiate an interactor, and pass in the ticket machine
    TicketMachineInteractor interactor = new TicketMachineInteractor(tm);

    // Start the interactor
    interactor.Start();
  }
}
```

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.

Step 7) Try running your application!

The output should be similar to this:

```
(q) Quit
(i) Insert money
(b) Show balance
(p) Show price
```

```
(r) Refund money
(t) Print ticket
================================================
Choose action (h for help)
>
```

If you, when running your application, do make one of the choices above – of course a `NotImplementedException` will be thrown because that's all our concerete Ticket Machine does for now.

1.1.4 Ticket Machine ticket costs

Before we get into implementing the body of each method in the concrete ticket machine class – let's declare a constructor. The constructor will take an integer as an argument. The integer represents the price of a single ticket. Consequently, our constructor needs to save the value passed in, as an instance variable so that we can use it later on. We need it to Let's do it:

Step 8) Implement a constructor in the ticket machine class, that takes an integer as an argument, and saves that value to an instance variable.

```
class TicketMachine : ITicketMachine{
  private int price;

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

1.1.5 Full implementation

Now it's time to actually implement the members of the concrete ticket machine class. Below you'll find a runtime example of what a session with the ticket machine could look like.

```
Choose action (h for help)
>   i
>>> Enter the amount you wish to insert (SEK):
>   50
>>> Inserted 50 kr
================================================
Choose action (h for help)
>   t
>>> Insufficient funds.. Missing 150 kr
================================================
Choose action (h for help)
>   p
>>> The price of one ticket is 200 kr
================================================
Choose action (h for help)
>   i
>>> Enter the amount you wish to insert (SEK):
>   175
>>> Inserted 175 kr
================================================
Choose action (h for help)
```

```
>   t
>>> Printing ticket....
================================================
Choose action (h for help)
>   b
>>> Your balance is currently: 25 kr
================================================
Choose action (h for help)
>   i
>>> Enter the amount you wish to insert (SEK):
>   75
>>> Inserted 75 kr
================================================
Choose action (h for help)
>   r
>>> Refunding 100 kr
================================================
Choose action (h for help)
>
```

If you find the runtime example above too confusing – there's some comments below explaining what each method should do. Please note: all the methods return strings that will be printed by the interactor.

```
public string InsertMoney(int amount){
  // update the current balance as defined by amount
  // only if amount is greater than 0
  // return string that explains what happened
}
public string ShowBalance(){
  // return a string that describes the current balance
}
public string ShowPrice(){
  // return a string that describes the price of a ticket
}
public string PrintTicket(){
  // if balance is enough
  //    reduce balance accordingly, and
  //    return string describing ticket
  // if balance is not enough
  //    return error message and amount missing
}
public string RefundMoney(){
  // print the amount of money refunded
  // reset the current amount and print it
}
```

Consequently the last task of this exercise is to...

Step 9) Implement the intended functionality of all the `ITicketMachine` members in the concrete `TicketMachine`.

1.2 More Ticket Machines

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

Naturally we can change the ticket price at **compile time**:

```
public static void Main ()
{
  ITicketMachine cheap = new TicketMachine (1020);
  ITicketMachine expensive = new TicketMachine(40500);
...
```

And naturally we can of course alter the price at **runtime**, by for example allowing the user to specifiy the price of a ticket:

```
public static void Main ()
{
  Console.Write("Specify how much a ticket is in SEK and press enter");
  int price = Int32.Parse(Console.ReadLine());
  ITicketMachine tm = new TicketMachine (price);
...
```

But there's one thing we haven't really talked about yet. The power of programming towards interfaces. Our ticket machine is actually way more abstract than one might first imagine. Take note of the fact that the `TicketMachineInteractor` class does **not** accept a `TicketMachine`, but rather an `ITicketMachine`. This means that it can an instance of any class as long as that class implements the `ITicketMachine` interface.

1.2.1 Donation Machine

Let's build a new ticket machine that also implements the `ITicketMachine` interface, but exhibits different behaviour.

Step 10) Create the file `DonationMachine.cs` and let it contain the class definition that implements the interface `ITicketMachine`.

Step 11) Implement the body of the members, according to the specification below.

The pseudo-code below explains what the donation machine does.

```
public class DonationMachine : ITicketMachine{
  public string InsertMoney(int amount){
    // update the current balance as defined by amount
    // only if amount is greater than 0
    // return string that explains what happened
  }
  public string ShowBalance (){
    // return a string that describes the current balance
  }
  public string ShowPrice (){
    // return a string that describes the minimum donation
  }
  public string PrintTicket (){
    // if balance is greater than minimum donation
    //    commit donation by
    //      - update total balance with current balance
```

```
    //      - reset current balance
    //      - print total balance
    // else
    //    return error message and amount missing
  }
  public string RefundMoney (){
    // print the amount of money refunded
    // reset the current balance
  }
}
```

Now we can see why programming towards interfaces is such a powerful idea. Instantiate and send a `DonationMachine` to the `TicketMachineInteractor` instead.

```
class Program{
  public static void Main(){
    ITicketMachine tm = new DonationMachine(10);
    TicketMachineInteractor interactor = new TicketMachineInteractor(tm);
    interactor.Start();
  }
}
```

Step 12) Run the application! And consider how we didn't have to make any modifications to the `TicketMachineInteractor` class.

Because the `TicketMachineInteractor` class is prpgrammed towards an interface rather than an implementation – it is fully functional with whatever implementation we choose to build, so long as the implementation implements the `ITicketMachine` interface.

1.2.2 Parking Machine

While we're at it, let's create another machine. A regular 'ol Swedish parking machine.

Step 13) Create a new class called `ParkingMachine` and let it implement the `ITicketMachine` interface. It's implementation should follow the specification below.

- The price is the value of 10 minutes of parking.
- The price should (as in the other machines) be set via the constructor.
- 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.

1.3 Wrap up

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 string InsertMoney(int amount){
    // increase current balance locally
  }
  public string ShowBalance(){
    // return current balance locally
  }
  public string ShowPrice(){
    // make a web request to get the price of a ticket
  }
  public string 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 string RefundMoney(){
    // refund money locally
  }
}
```

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

2 Basic GUI coding and Event Handling

In this assignment we're going to build a local network chat system. One server part and one client part. The idea is that you start up one server, then multiple clients. Messages sent from connected clients will be broadcasted to all connected clients.

This assignment is preferably solved in pairs of two. Groups of more than two people are however not allowed.

Step 14) Create a new Visual Studio C# solution. In the solution, create two separate `Windows Forms` projects. One for the `client` and one for the `server`.

To tackle this task as easily as possible we're going to take it in two turns. We'll start off by simply creating a dummy chat client. Basically just the UI. When we've got that all set up, we'll move on to actually building the fully functional chat system.

2.1 Client GUI

Step 15) Open up the `client` project's main `form` and modify the UI so that it contains the following...

- a `ListBox` for displaying chat messages.
- a `TextBox` for writing nickname and/or chat messages.
- a `Button` for connecting/disconnecting.
- a `Button` for sending chat messages.

Step 16) Add logic so that the following behaviour is exhibited:

(a) If the user is not connected. When he/she enters a name in the textbox and presses the connect button, an entry should be posted in the listbox saying `"[nickname] connected"`. The text of the connect button should switch from `"connect"` to `"disconnect"`.

(b) When a connected user presses the `"disconnect"` button, an entry should be posted in the listbox saying `"[nickname] disconnected"`. The text of the connect button should switch from `"disconnect"` to `"connect"`.

(c) Whatever is in the textbox at the time of pressing connect is used as a nickname. It's not allowed to connect without a nickname.

(d) When a connected user presses the send button whatever is in the textbox is sent as a chat message, and consequently shows up in the listbox in the following form: `"[nickname] says: [message]"`. It's not allowed to post empty messages.

2.2 Real chat

Now we'll convert our dummy chat to an actual chat.

Step 17) Make a backup-copy of your chat GUI.

Step 18) Convert your chat GUI into a real chat system that allows for multiple clients to connect to the same server. This means you're going to have to build the server as well. The requirements are simple:

(a) All messages sent to the listbox in the last exercise, should be broadcasted to all clients connected to the chat.

(b) Two clients cannot have the same nickname.

It's basically up to you to figure out how to solve this exercise. However it's reasonable to assume that you're going to have to work with `socket`'s and `threads`. Below you'll find some code snippets to get you started.

2.2.1 Helper methods

These are quite handy to have since we're going to convert from byte arrays to strings and vice versa. These are useful both for the client and for the server so it's a good idea to put the class in a new project, but in the same solution. To add a reference to another project in the same solution, right click the `References` foldr in the `Solution Explorer`. Then choose `Add Reference > Projects > [proj.name]`

```
public class ConversionUtils
{
  public static String ByteArrayToString(byte[] byteArray){
    return System.Text.Encoding.UTF8.GetString(byteArray);
  }
  public static String ByteArrayToString(byte[] byteArray, int index, int count){
    return System.Text.Encoding.UTF8.GetString(byteArray, index, count);
  }
  public static byte[] StringToByteArray(String message){
    return System.Text.Encoding.UTF8.GetBytes(message);
  }
}
```

2.2.2 Client snippets

Connecting to a socket.
Hint: this probably needs to spawn a new thread listening for contents from the socket.

```
try{
  byte[] serverIP = { 127, 0, 0, 1 };
  IPAddress serverAddress = new IPAddress(serverIP);
  Socket socket = new Socket(AddressFamily.InterNetwork,
                             SocketType.Stream,
                             ProtocolType.IP);
  socket.Connect(serverAddress, 5000);
  ...
}
catch(Exception e){
  ...
}
```

Receiving (listening for) contents from a socket.
Hint: This probably needs to be done in a separate thread (i.e. not the UI thread).

```
byte[] buffer = new byte[1024];
int byteCount;
while(true){
    byteCount = socket.Receive(buffer);
    String incomingMessage =
      ConversionUtils.ByteArrayToString(buffer, 0, byteCount);
}
```

2.2.3 Server snippets

Start a server.

```
byte[] byteip = { 127, 0, 0 , 1};
IPAddress ip = new IPAddress( ip );
TcpListener listener = new TcpListener(ip, 5000);
listener.Start();
```

Listen for incoming connections.
Hint: Probably needs to be done in a separate thread because of the eternal loop.

```
while (true)
{
  try
  {
    // Wait for a connection
    Socket clientsocket = listener.AcceptSocket();

    try {
      byte[] buffer = new byte[1024];

      // Wait for data (nickname)
      byteCount = clientsocket.Receive(buffer);
      String requestedNickname =
        ConversionUtils.ByteArrayToString(buffer, 0, byteCount);

      // make sure nickname is not taken
      // and start off a new thread listening for
      // more incoming data on the clientsocket
    }
    catch (Exception ex){ ... }
  }
  catch (Exception ex){ ... }
}
```

Send message to socket.

```
clientsocket.Send(ConversionUtils.StringToByteArray("Hello world"));
```

3 Iterator

In this exercise we'll explore the iterator pattern using the phone book example from the lectures.

Step 19) Create a new `C#` project, and implement the phone book examples from the notes. But refactor the find method to throw an exception instead of returning `-1`.

Step 20) Draw a class diagram for the iterator, without using diagram generation in `Visual Studio`.

Step 21) Implement the `iterator pattern` for the phone book, populate the phone book and iterate over it, printing its contents.