# 1. Understanding Object-Oriented Programming

## Description

OOP is a core and fundamental principles and it’s crucial for developer to understand it and be able to use in application development. Correct using OOP principles gives ability to write extensible and supportable code that helps easily adding new pieces of functionality to existing application and helps saving time for developer. Also, it is easier to read unknown code that was written by other developers while investigating errors or bugs in case when code is followed OOP principles.

## Goal

Current task is going to be used to make understanding how to use OOP in real application and how to transform requirements and relations from human understandable objects to classes, interfaces, etc. For doing that we will use mobile phone evolution from simple gadget with physical keyboard to brand-new smartphone with touch screens. The goal is to create classes which will describe phones with their commons and differences; structure should support extensibility (in case when new smartphone with extra-new features produced) for describing new devices based on existed codebase. Console application will be created.

## Task

For starting you should define core objects which will be used during this task.

#### Task 1. Classes

The first step is determination what common in all the mobile phones and creating appropriate objects (classes) for them. We can easily find a few common things for any mobile phone:

* Screen
* Keyboard
* Battery
* Sim-card
* Dynamic
* Microphone
* …

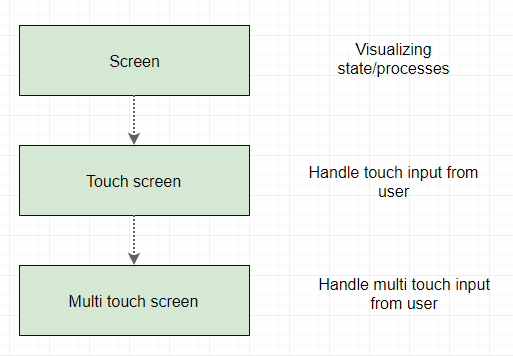
You can add as many other components as you think it makes sense to you. Each component can be described as class, while doing it don’t forget to fill each class with appropriate properties; for the screen it could be size, pixels; for keyboard are figures, letters; for battery are size, volume and so on.

Newly created classes can be combined in Mobile Phone entity and be treated as Properties.

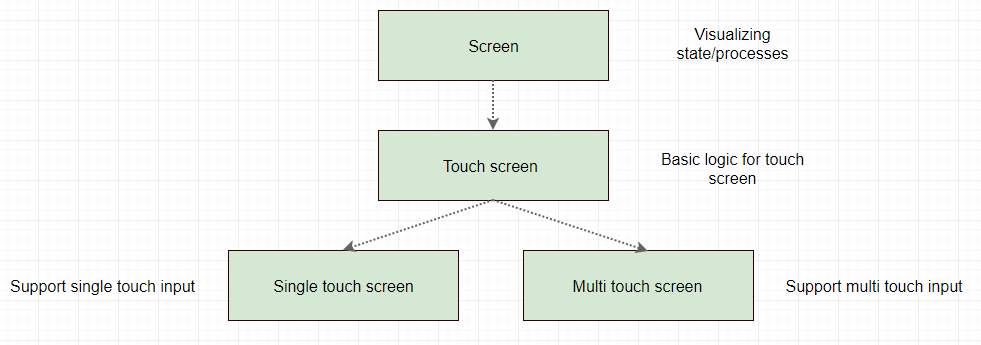
#### Task 2. Inheritance

While creating classes you could notice that such detailing level not enough for describing components. For instance, screen can be monochromatic or colored, be touch screen with multi-touch supporting or not, etc.

There are two ways for reflecting such things in the code: using Properties or Inheritance. We are going to use **inheritance** for showing relations between the objects. Let's look at an example with a screen. The goal of a screen is for showing current Mobile state and showing changes are being made by user. Such screen can be considered as base for any others. Then, there is touch screen which also supports user actions handling, we can say it extends basic screen functions. By digging deeper, we can single out multitouch screen as new screen type with extended functionality; it also can handle input from user but does it differently. Such relations can be visualized with multilevel inheritance diagram.



You can argue with such relation by saying that multi-touch screen doesn’t really extends single-touch screen, there is only different how input is taken from the user. It can be visualized with combination of multilevel & hierarchical inheritances.

 Inheritance type depends on the needs and requirements. There is no correct or incorrect way of inheritance, it is always trade off between different parts of application. One thing you have to care of is next: inheritor have to extend base class but not cut-off basic functions. If you are going to remove base class functionality from inheritor you should stop and think about design, probably you’re doing something wrong.

By using the same principle, create such relations. Inheritance will help you to reflect components evolution.

#### Task 3. Polymorphism

To this step, you have a class structure for describing all the components. Next, you need to fill these classes with appropriate logic, in this case polymorphism will help us. On an example of the screen we will consider polymorphism using.

Create a class structure for different screen types. To begin with, we will declare an interface that is a picture that should be shown on the screen:

public interface IScreenImage {

}

This interface can contain many different information, such as image size, coordinates where to draw, etc., but we will leave it blank for simplicity.

Next, declare Screen base class that knows that can display a picture:

public abstract class ScreenBase {

public abstract void Show(IScreenImage screenImage);

}

This class is basic and tells us that the screen can display a picture, and descendants will decide how to do it. The first screens on mobile phones were monochrome, we will declare it:

public class MonochromeScreen : ScreenBase {

public override void Show(IScreenImage screenImage) {

Console.WriteLine($"I am {nameof(MonochromeScreen)}");

}

}

The monochrome screen implements the Show method to display the image.

As the development appeared phones with a color screen. Display this:

public class ColorfulScreen : ScreenBase {

public override void Show(IScreenImage screenImage) {

Console.WriteLine($"I am {nameof(ColorfulScreen)}");

}

}

The color screen has its own implementation of the Show method. There are also different types of color screens:

public class OLEDScreen : ColorfulScreen {

public override void Show(IScreenImage screenImage) {

Console.WriteLine($"I am {nameof(OLEDScreen)}");

}

}

public class RetinaScreen : ColorfulScreen {

public override void Show(IScreenImage screenImage) {

Console.WriteLine($"I am {nameof(RetinaScreen)}");

}

}

Each color screen has its own implementation of the Show method, and all of them are descendants of the base ColorfulScreen.

Then put the screen in the phone:

public abstract class Mobile {

public abstract ScreenBase Screen { get; }

private void Show(IScreenImage screenImage) {

Screen.Show(screenImage);

}

}

Mobile has a screen and it can use it to display an image without taking into account which type of screen is used.

Also, the screen can display a picture with different degrees of brightness:

public abstract class ScreenBase {

public abstract void Show(IScreenImage screenImage);

public abstract void Show(IScreenImage screenImage, int brightness);

}

By analogy with this example, describe the functionality for all previously created components.

The next step is to enable each component to have a textual representation or description (it will be necessary for the next steps). To do this, we will use the overridden method ToString(). Consider an example based on the already known Screen.

public class MonochromeScreen : ScreenBase {

...

public override string ToString() {

return "Monochrome Screen";

}

}

public class ColorfulScreen : ScreenBase {

...

public override string ToString() {

return "Colorful Screen";

}

}

Similar methods can be created for other Screens. Create the same methods for all components that you already have.

Next we'll write usage of the ToString () method. To do this, we will create a method in the Mobile that will return information about itself.

public abstract class Mobile {

public abstract ScreenBase Screen { get; }

...

public string Description() {

var descriptionBuilder = new StringBuilder();

descriptionBuilder.AppendLine($"Screen Type: {Screen.ToString()}");

return descriptionBuilder.ToString();

}

}

Depending on the type of the ScreenBase descendants, the ToString () method will return the corresponding value, let's say thank you to polymorphism for this.

As in the example above, expand the Description method to include all the components in the Mobile.

When there are descriptions for all components, you can start creating your own phone. Let's consider an example.

public class SimCorpMobile : Mobile {

public override ScreenBase Screen { get { return vOLEDScreen; } }

private readonly OLEDScreen vOLEDScreen = new OLEDScreen();

}

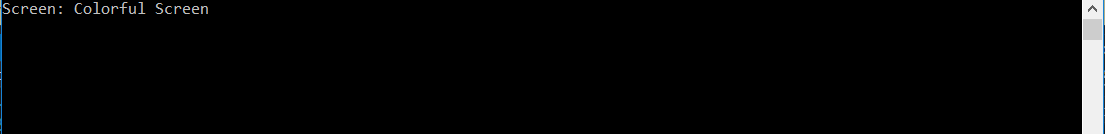
We have a SimCorpMobile mobile with an OLED screen, for this we override the abstract property Screen and in the getter implementation return a screen of the type we need. Well, using the same technique, create your Mobile description with components that interest you.

#### Task 4. Console application

The last step is to create a console application in which your Mobile description will be displayed:

* Create the Console App. (.Net Framework) project in Visual Studio.
* In the Main method of the Program.cs file, create an instance of your Mobile and call the Description method, result must be output to the console using the Console.WriteLine (...) method.
* To ensure that the console window does not close immediately after execution at the end of the method, you need to write Console.ReadKey (). In this case, the console will not close until any key is pressed.

Console example based on above examples:



# 2. Interfaces

## 

## Description

Interfaces provide flexibility in extending applications, and also serve as a primary means of reducing the dependency between program components. The popular technique Dependency Injection (DI) is implemented using interfaces, which in turn can be tested very well. If you need to write a simple one-time program without support in the future, then you can do without interfaces, if you are going to take an active part in the development of the Enterprise product, then the interfaces are must have to know.

## Goal

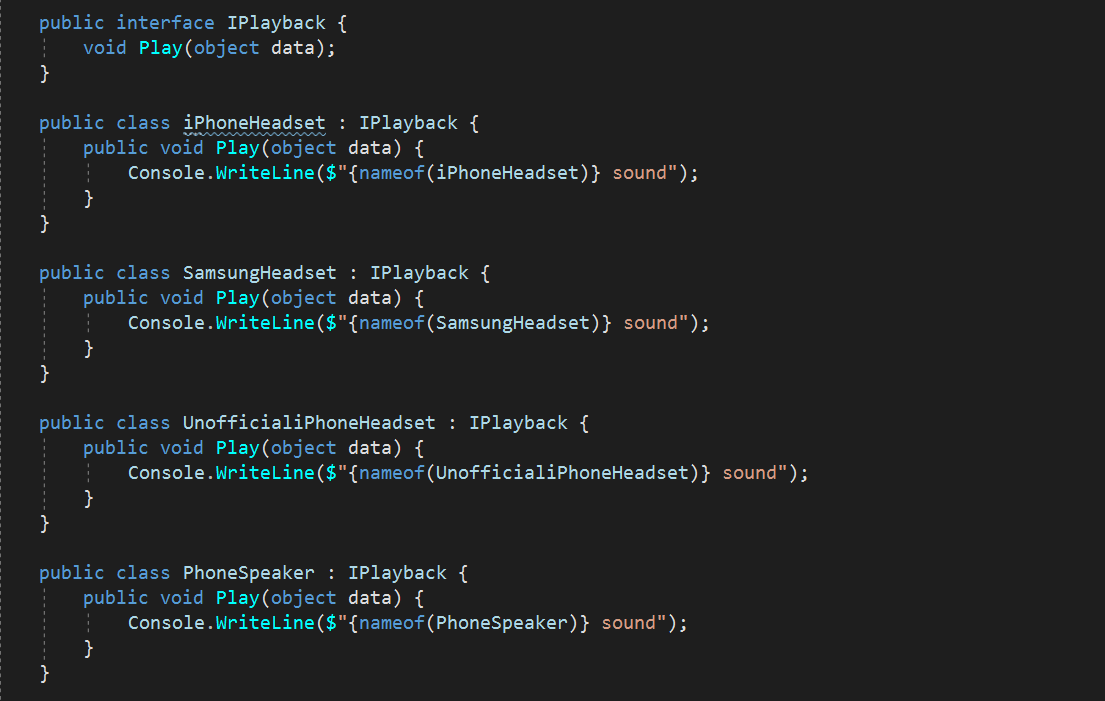
The goal of the task is to understand in what cases it is possible and necessary to use interfaces, and also be able to apply them in practice. Consider and implement the DI pattern.

## Task

The ideal example of the interfaces is in the mobile is audio jack. The mobile phone does not know and can not know what particular device will be used to play sounds, it can be simple headphones, headphones with a microphone or speakers, officially manufactured or faked, for the phone it does not matter, the main thing is that all these devices support the interface provided by mobile.

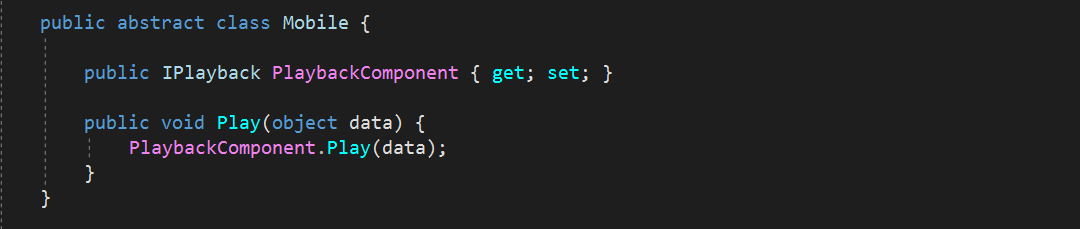
#### Task 1. Declaration

Consider the use of interfaces on a 3.5 mm connector (**phone jack):**



The IPlayback interface describes what the playback device should do-the Play () method. Then there are the classes implementing the interface and method IPlayback Play () in particular. Depending on the device, the Play () method can be implemented in different ways. The PhoneSpeaker class does not related to headphones, but it can also play sounds through the speakers and also implements the IPlayback interface; is a good example of the extensibility: we are not interested in the implementation of the sounds are played, all that we are interested is that device compatible with interface.

The next step is to add the IPlayback playback device to the mobile:



Here we see that the phone knows that it has a playback device but does not have the slightest idea of a particular implementation.

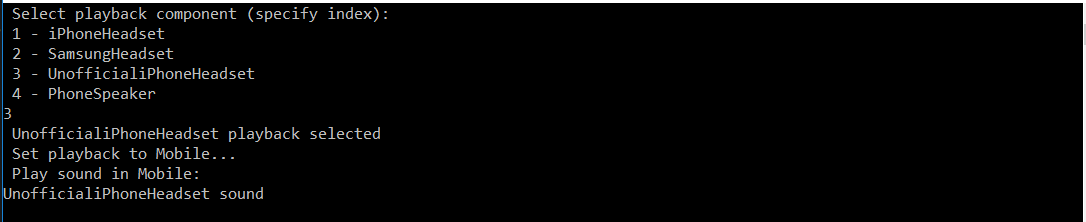
Consider other components in the phone that can be represented as interfaces (charger?) and implement them.

#### Task 2. Console application

By this point you should have several components or plug-ins represented in the form of interfaces. The next step is to create a console application (you can use the application created earlier) with the ability to select components and display information from these components.

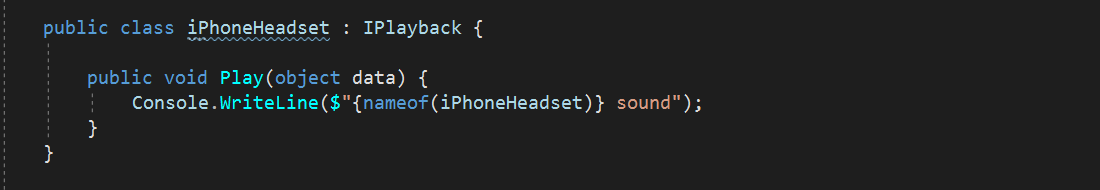
It is necessary to write a program where the user is given the opportunity to select several components for the phone, after which the selected insert / apply in the phone and perform the corresponding functions (method Play () in the headphones). For example, the user can choose headphones, chargers, etc .; add as many components as you like.

In the case of headphones, the application might look like this:



#### Task 3. Dependency Injection (DI)

You have components that can output information to the console, for example:

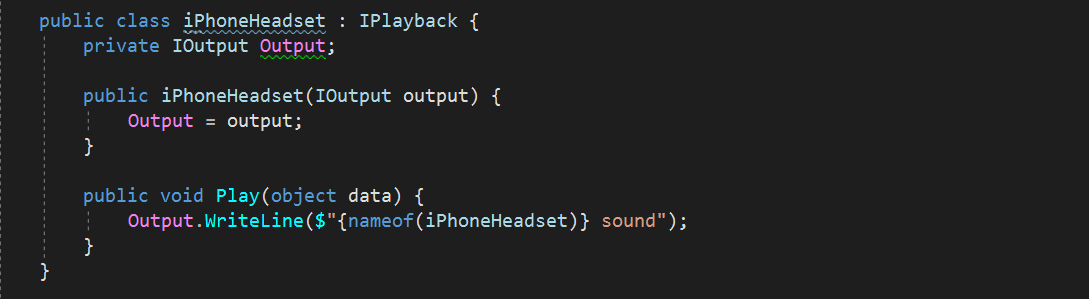


This implementation is limited in extensibility, if you need flexibility in the output of information, how to be?

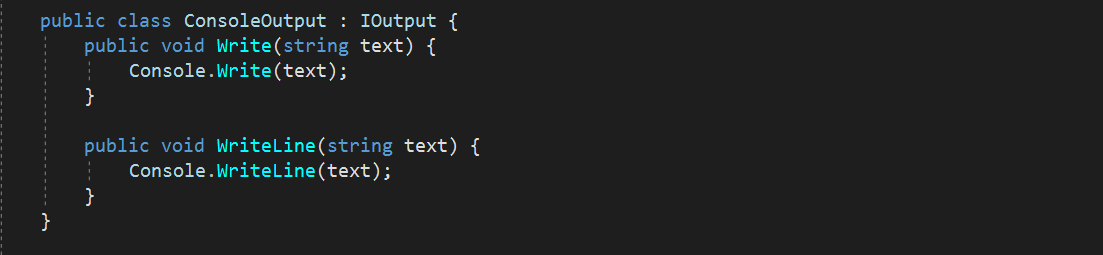
Declare an interface that describes the methods of outputting information:



Then the interface is used in one of the components:



As you can see from the example, the IOutput interface is passed to the iPhoneHeadset constructor and is further used to display information. This approach is called Dependency Injection via the constructor and is one of the decoupling mechanisms, you can now make the implementation of the IOutput interface and use the iPhoneHeadset component without changing the code:



As the example shows, the ConsoleOutput class inherits the IOutput interface and implements its Write () and WriteLine () methods, the console is used to display the information.

Create an interface that describes information output and implement it (you can also output information to the console). Add this interface, using DI, to all components that can output information. Correct the application written in Task 2 using the output interface, the result of the output to the console should be the same.

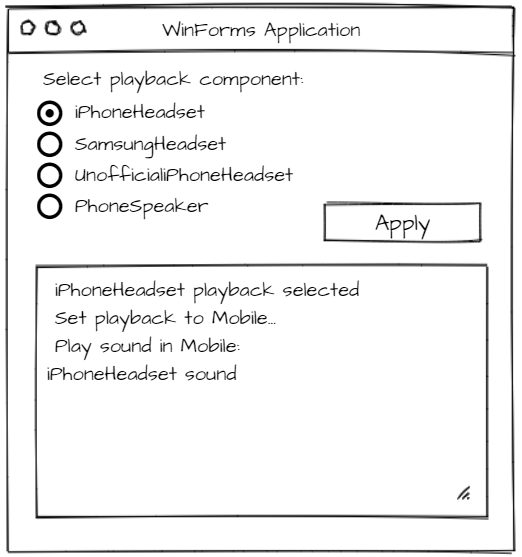
#### Task 4. WinForms application

We continue using the information output interface, but now in the WinForms application:

* Create a Windows Forms App project (.Net Framework) and add it to an existing one solution
* Create a Class Library project (.Net Framework) and place all your classes in it from the console application. In the console application, only Program.cs and the class that implements the IOutput output interface should remain.
* Add the Class Library project to the Console and WinForms application references.

After the actions are completed, you must have a working console application. The next step is to extend the use of the IOutput interface to output data in WinForms.

By analogy with the console application, do the same thing in the WinForms application, for example:



#### Task 5. Unit tests

Благодаря DI написание юнит тестов становится проще, на примере интерфейса IOutput можно создать фейковый класс который будет сохранять выводимое значение которое будет проверяться в Assert текста. Например метод Play() класса iPhoneHeadset можно проверить на выводимое значение передавая применяя реализацию интерфейса IOutput специально для юнит тестов.

Создайте юнит тесты которые будут проверять выводимые значение компонентов

# 3. Events, delegates

## 

## Description

Events are a powerful tool for informing about any changes in the state of components, subsystems or user actions when it comes to UI. Working with the event, do not forget that the main cause of memory leaks is not detaching from events. Events are built based on delegates but have their own limitations and advantages. Delegates allow you to call methods, even private ones, from other classes, components, and threads. Great power generates great responsibility, so delegates need to be used wisely.

## Goal

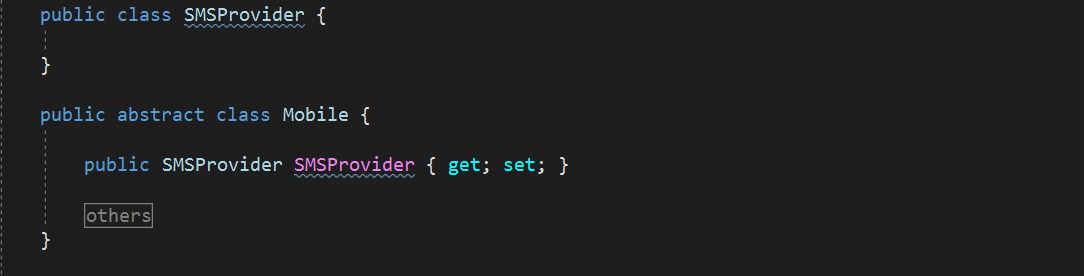
Learn to use events and delegates and be able to apply them only where it makes sense.

## Task

The phone has a very favorite message transfer function that we implement (or simulate) in the WinForms application.

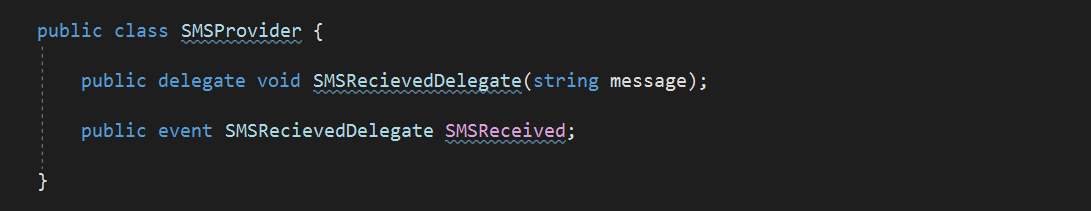
#### Task 1. Events

To begin with, you need to add a SMS module (regular class) to your phone if you do not have one. In order not to add to all phones (if you have more than one), you can add an SMS module to the base class of the phone, for example:

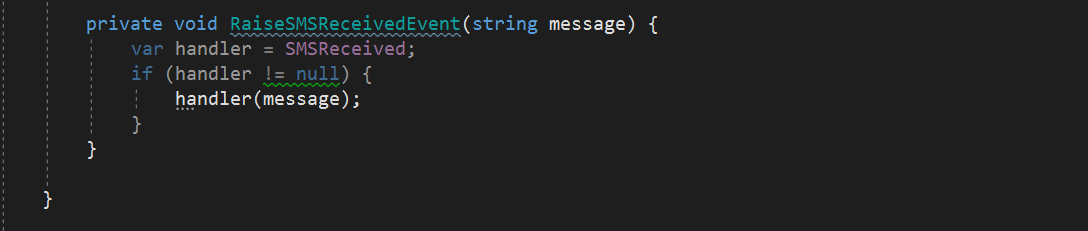


Thus, all phones will automatically contain SMSProvider, which is very convenient.

In a real phone, SMS notifications come through GSM communications, since we do not have GSM it is necessary to simulate the receipt of SMS. To do this, you need to declare the event in SMSProvider:



Next, you need to generate SMS and make a notification about it. To generate SMS, you can use the Timer and every few seconds "issue" a new message. You can generate the event by a simple call to SMSReceived () but as practice shows, this can be a problem, because SMSReceived can be Null if no one subscribes, so you need to check the event on Null. In this case, another problem awaits us - thread safety. There is a way to thread-safe call an event, it must be used in all cases when you call events:

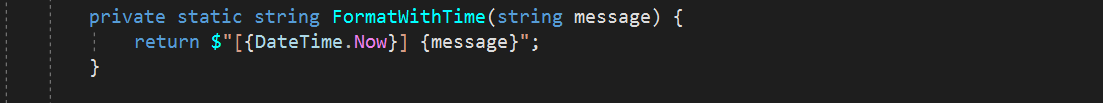


First, the delegate is stored in the handler reference, this step makes the call thread safe, the Null test ensures that the event is called only when someone is subscribed to it.

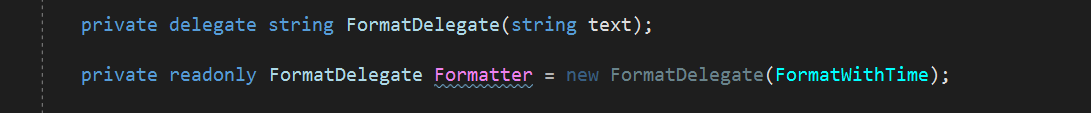
To display messages, you can create a new form and a text field (RichTextBox) where messages will be shown. The interval with which messages are generated select at your discretion. Timer can work in Non-UI thread, so do not forget to check InvokeRequired.

#### Task 2. Delegates

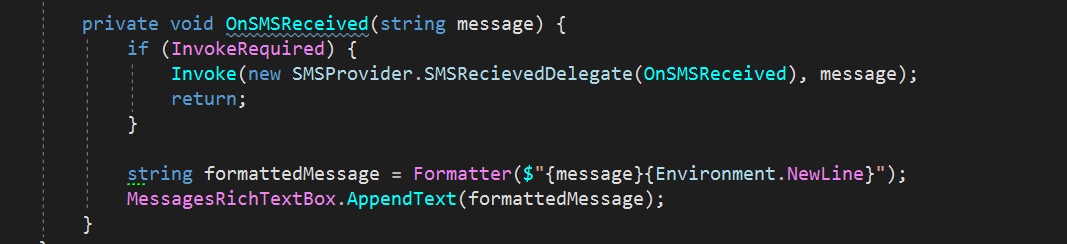
The format of the output of messages can be expanded and made more flexible, for this purpose delegates can be used. Consider adding receiving time into output, the first step is to implement a method that adds a date to the message:



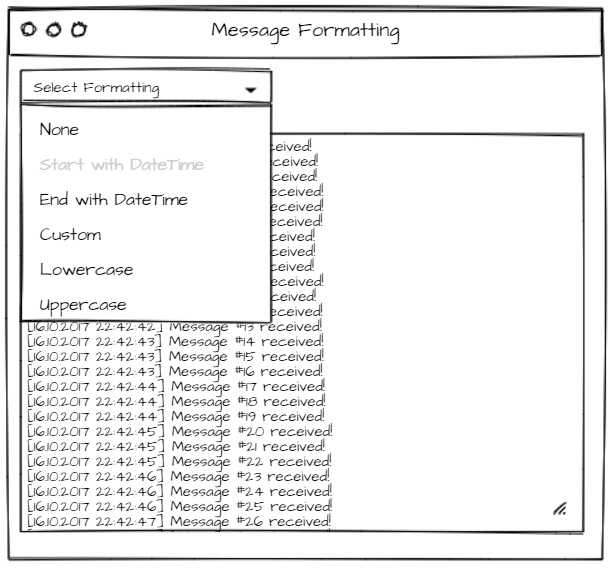
Next, you need to declare the delegate that will be used for message formatting:



First, a FormatDelegate delegate is declared which describes a method with an input parameter of type string and a returned string value. After that, the Formatter delegate is initialized to which the FormaWithTime method is passed. The next step is to apply the created delegate when displaying messages:



This example does not disclose the power of delegates, to see how delegates help write extensible code you should create multiple formatting options. Create 5 different message output formats and implement them through delegates. Formatting options add to the ComboBox, depending on the selected value, the corresponding delegate \ formatter should be applied. Form can look like this:



#### Task 3. Unit tests

Напишите юнит тесты для проверки форматирования. Возможно Вам необходимо изменить код для того, чтобы было возможно протестировать.

Напишите юнит тесты который будет проверять, что Event Raised когда новое сообщение генерируется.

# 4. LINQ, Collections

## Description

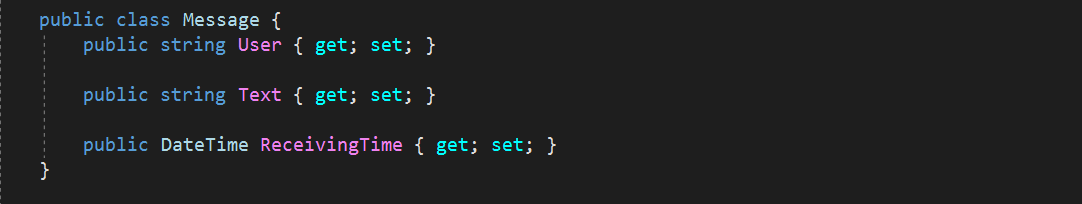
With the release .Net Framework 3.5, LINQ has become very fond of programmers because of its multi functionality and ease of use (of course, one should take into account that it's very simple to write a complex and hard-to-read LINQ query, but this is not an end in itself?). Filtering, sorting, sampling - all this is possible thanks to LINQ, which, in turn, goes shoulder to shoulder with Generics, knowing how LINQ works without knowing Generic is impossible.

## Goal

Learn how to write simple LINQ queries to filter and sort lists.

## Task

In the previous task, messages were only text, such a simplified interpretation of messages has nothing to do with reality. At a minimum, the message consists of a subscriber (number, name), text and time of delivery:

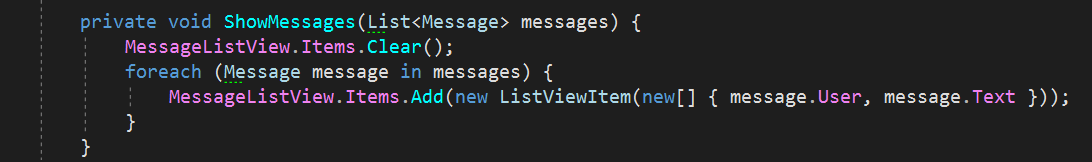


Replace a simple string in generating messages to the Message class, add any information you think is necessary to this class. Messages must be received from at least three subscribers.

After the messages have become more complex and contain additional information, this can be displayed on the form. To be more similar to messages in the phone, we make a list containing information about the subscriber and the message itself.

Instead of outputting a message on the form, make caching - add each message to the list, you can use List <Message> for this.

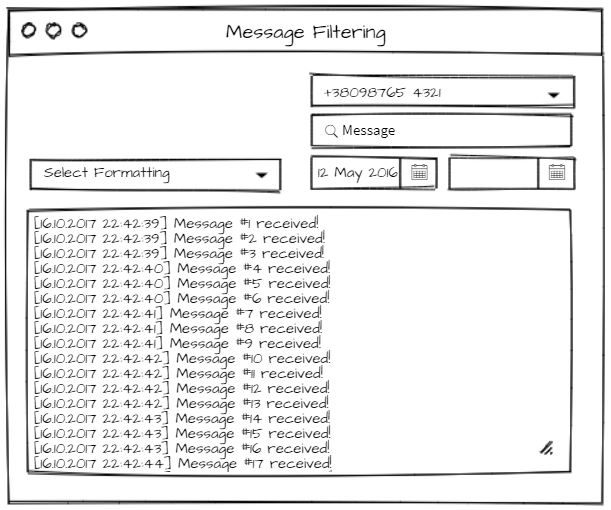
The next step is to display the messages in a more convenient way with the display of the subscriber. To do this, you can use ListView control with View = Tile and also with two columns. To display the subscriber's name and message text, next approach can be used:



#### Task 1. Applying LINQ

Having a list of messages and a way to output information, you can start using Linq:

1. Select from the list of messages all unique subscribers (User) and output them to the form using ComboBox. Depending on the selected subscriber, display only his/her messages.
2. Add a TextBox control, which will serve as a filter by messages. Display only those messages that contain entered text.
3. Add two DateEdit controls to filter messages by date (from and to).
4. All 3 filtration methods can be used simultaneously or one at a time. Add a Checkbox with which you can specify the filtering logic: OR or AND if more than one condition is specified.



#### Task 2. Unit tests

Write unit tests for testing each filter (getting unique Users, search message by text, etc.) separately.

Write unit tests for testing combined filters. All the combinations should be covered. Don’t forget to cover OR / AND logic for combined filters.

#### Task 3. Collections

Since you have option for generating message they should be stored somewhere. The first step is to declare storage class which will have message collection inside. Storage class should have methods for adding and removing messages. Make Storage class be part of Mobile and implement adding new messages to it from SMSProvider class. So that, Mobile will be able to store incoming and outcoming messages automatically.

Add events to Storage class which will notify that message added or removed.

SMSProvider class can be considered as Mobile internal infrastructure and should be marked as internal. After doing that you have to update Form to change subscription from SMSProvider to Storage. So that, SMSProvider is used only for internal Mobile purpose and all incoming messages are saved to Storage and the one is responsible for storing messages and notifying about changes.

#### Task 4. Unit tests

Write unit tests for verifying that messages are being added to Storage collection when new message coming. Also methods for adding and removing messages in Storage class can be tested.

# 

# 5. Threading, Tasks

## 

## Description

Threads and Tasks are also a powerful tool when writing Enterprise applications and are also the source of many problems associated with synchronization, access, locks.

## Goal

Learn how to use threads and tasks at a basic level using synchronization methods.

## Task

In the previous task, you added Timer to generate messages. This is a good candidate for applying threads.

#### Task 1. Using Thread

The first thing which should be done is removing Timer from SMSProvider (or other class you use for message generations). Instead of a timer, you need to add a thread that can be controlled by the Start () and Stop () methods. Also, you can add a Start / Stop button on the form with which you can start and stop message generation.

After these changes, messages must be generated in a separate thread. A good start.

#### Task 2. Thread Synchronization

Consider the use Threads based on the battery. The current battery charge is a property that needs to be "synchronized" depending on the current state (the phone is charging or not).

Create a Battery class if you do not already have this class and add the Charge property to it. The battery can discharge, and this property can be implemented with a Thread that will reduce the charge by 1% every couple of seconds.

On the other hand, the phone can be put on charge and in that case the charge will increase, this property can be implemented with a Thread which will increase the charge by 1% every second. The crucial point is that the charge can not at the same time is being increased and is being decreased, for this you can use the lock () statement when the current state of the charge changes.

Implement two Threads that should simulate charging and discharging the phone.

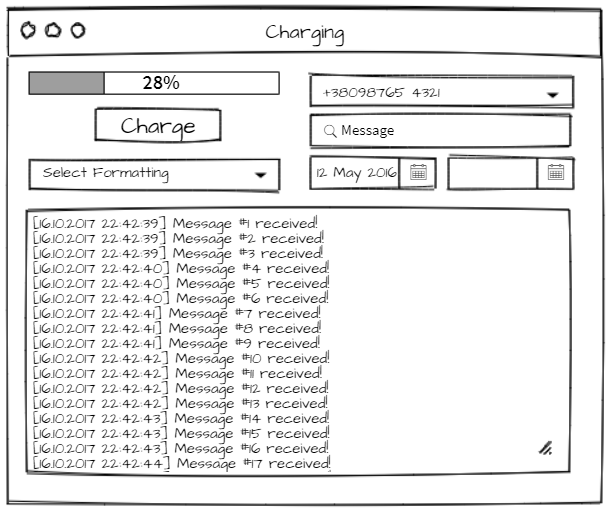
#### Task 3. Progress visualization

After the battery "knows how" to be charged and discharged, you can display the current battery status on the form.

For this indication, you can use the ProgressBar component. The battery can inform you of a change in the current charge by an event that can be used to update the ProgressBar.

Also need to add to the form the ability to "put" or "remove" the phone from charging by starting and stopping the Thread which is responsible for charging.

Form example:



#### Task 4. Using Tasks

Once you have SMSProvider which can generate messages by using Threads let’s create another implementation of that class by using Task instead of Thread. So that, there should be base class for generating messages and two inheritors which use different approach for message generations. These classes should be interchangeable.

Check that messages are being generated identically while using any of these classes.

By analogy with SMS generating use Tasks for simulate Charging and Discharging.

Consider using Factory Method pattern for determination what class should be used – with Thread or with Tasks.

#### Task 5. Unit tests

Write unit tests for verifying that Charge cannot be less than 0% and more than 100%.

Write unit tests for verifying that Charge is being decreased when Charger **Thread** is not turned on and vice versa.

Write unit tests for verifying that Charge is being decreased when Charger **Task** is not turned on and vice versa.

# 6. Equality and Comparisons

## Goal

Basic application and understanding Equality and Comparison.

## Task

The ideal option for considering Equality and Comparisons is the call list. To implement it, you need to add several entities.

#### Task 1. Comparisons

Add a new Contact class that will describe the subscriber, one subscriber may have more than one phone.

It is also necessary to create a Call class that will contain information about the contact, call time and call direction - incoming or outgoing.

The call list is always sorted by the date of the call in descending order. To make this possible, you can implement the IComparable interface in the class Call. After that, calls can be sorted.

#### Task 2. Unit tests

Write unit tests which verify sorting in the list while adding or / and removing item(s) from /to the List .

#### Task 3. Form

Create a form on which calls will be displayed, for this you can use ListView control

To generate calls, you can use a separate thread, similar to SMS.

Calls can be saved to List <Call> and sorted before being added to the ListView.

#### Task 4. Equality

Modern phones are able to combine similar calls into groups, if someone calls you several times in a row, then in the call history such calls are combined into one. To be able to do this it is necessary to somehow determine whether the calls are the same or not.

Override the Equals method for the Call class. Note that the incoming and outgoing calls from the same contact are treated as different calls.

When adding a new call to the list, it is necessary to compare the new call with the last one in the list, if they are the same, then they must be combined. For doing that Call class can be extended for supporting similar calls by adding new property with associated calls out there. Another solution could be creating descendant from Call with additional property for storing Calls. You can think up whatever solution you want but you should have in mind OOP principles while doing it.

#### Task 5. Unit tests

Write unit tests for comparing Calls between each other. Cover all the cases you can think up.

#### Task 6. Form

Implement association of identical calls in the group by analogy with the phone and visualize it on the Form.