Notes

....

....

....

....

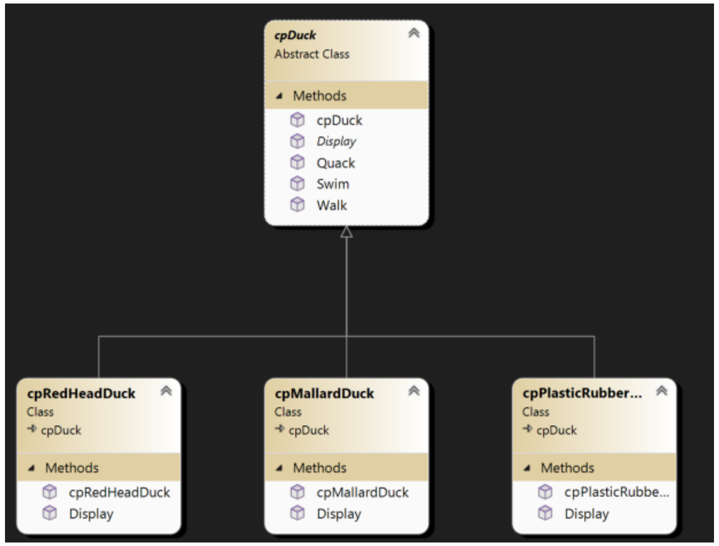
....

....

....

Exercises

	This is an exercise on design patterns using Object Oriented stuff. *There is an assumption that you have a notice of the things below:* • *Class (Base and child thru inheritance), Abstract Class, Abstract Method, Method, Property, Indexer, Interface, Public and Private, ….* *The goal of this exercise is to let you think about how software is build up and what the advantages and disadvantages of some solutions and techniques are.* *The goal is design patterns. How do you use Object Oriented techniques in your solution.* *The goal is to think first and then execute and test.* *You can find solutions for this exercise, but you will not learn anything without doing it by yourself.* *The code, the solutions and a try out routine can be found in the exercise 00999-a FuzzyDuck and DuckyFuzz Starting Point.zip*



Picture 1: Class Diagram Duck Game.

Notes

....

....

....

....

....

....

....

Exercise Fuzzy Duck and Ducky Fuzz

Intro

Let's assume you are part of a developer team that has created a game. The name of that game is "Fuzzy Duck".

The game is working, players can play it on PC, PlayStation. You have a lot of clients who pay for it. Everybody (developers, company and clients) are happy.

The game is evolving, so it is subject to change. And we want to be as "lazy" as possible.

There is a DuckGameLibrary. This contains the needed classes. They are used in 2 other projects.

- DuckGameTest. This should contain the test routines.
  There are no tests yet.
- Fuzzy Duck. The actual game that can be played.
  There is no game yet. Imagine there is a functional game.

There is a base abstract class for a Duck (cpDuck) and all the other Ducks inherit from that class.

In the picture beside you see the class diagram (*Picture 1*).

	Read all the comments in the original code of the DuckGameLibrary. Don't look at the code of the Fuzzy Duck Game. Correct the comments that are wrong. (I know 😊) *The developers has chosen a specific design pattern. It is good enough for the moment. But it will be subject to change (by you).* *You are changing an existing program, so be careful in what you do. Make sure you know why you are doing stuff.*

Part 01

In the Fuzzy Duck Game there are two startup forms.

- frmDuckGameTryout.
  - Here you will make your own exercise.
- frmDuckGame. This is an implementation of this exercise Part 01.
  - It is just for demoing purposes. Don't look at the code. It is important that you make this exercise by yourself.
  - Your solution (battleplan) will be different than mine.

	Run the program to see what is wanted. See the end goal. *When you have executed the tasks, you should have something similar.* *Change in Fuzzy Duck Game the cpProgram.Main() routine so that frmDuckGameTryout is shown and not frmDuckGame.* *This screen is empty.*

Your tasks

- Put a combo box on the screen.
- Put a picture box on the screen.
- In the combo box, you show all the class names that inherits from "cpDuck" class.
  - Example 00124-g AssemblyQuery is an example on how to do this.
  - A technique reflection is used.

	In my solution, I've put the code inside the form. *This is not the best location of your code. This is on purpose. So don't follow my solution. It is just for demoing purposes.* *Think where you should put your code to find all the needed classes.* *Think on how you get them inside your combo box.*

Notes

....
....
....
....
....
....
....

- You have your combo box filled with the correct classes.
- When you run your program and you choose an option, show the correct corresponding picture.
- In the picture box, you show the correct picture depending on what you have chosen in the combo box.
- The pictures can be found in the project "DuckGameLibrary" in the folder "DuckDisplays".
    - Example 00013-g PasswordControl demos to show an image.

	For now, we do this hardcoded in the form. We will not use the method "Display()" that is part of cpDuck. *I say, for now, in the future we will do that. But now we just show the picture using the events of the combo box that shows the correct child classes.*

- Put a label on the screen.
    - Set a good error message in that label when the picture is not found.
- To test your routine, you add an extra child class "cpWoodDuck". Don't add a corresponding picture yet.
- Run your routine. All should work, and for "cpWoodDuck" you should see the error message.
- Now add corresponding picture of a Wood Duck in the folder "DuckDisplays".
- Run your routine again. You should see all the correct pictures.

	Send your solution to Vincent. *Your solution will probably be good, but depending on your solution you need to do some correction steps, or not.* *Do not continue before you have confirmation that it is correct.*

Notes

....

....

....

....

....

....

....

Part 02

- You have a correct solution of part 01.

	Don't' continue with this pages, until you have implemented the needed changes in Part 01. *You have received an evaluation of Vincent that your solution is correct.*

We will add a test routine in the project "DuckGameTest". This is a console application. We will not actually create a game, we will just create a console application to see if the basic functionality works.

Are the methods starting up when we need it? The methods shows text, not in the console window, but in the output window. The reason is that the classes will be used in a game, and there is not a console window present in it.

Your tasks

- Create a list of cpDuck.
- Add an instance of "cpMallardDuck" to it.
- Add an instance of "cpPlasticRubberDuck" to it.
  - Pay attention here, you need to change the access modifier of cpPlasticRubberDuck to be able to use it.
- Add an instance of "cpRedHeadDuck" to it.
- Add an instance of "cpWoodDuck" to it.
- Execute all methods of the Ducks in the list.
  - Display, Quack, Swim and Walk.

	Send your solution to Vincent. *Your solution will probably be good, but depending on your solution you need to do some correction steps, or not.* *Do not continue before you have confirmation that it is correct.*

Notes

....

....

....

....

....

....

....

Part 03

In the Product Backlog, given by business, it is mentioned that we now want the ducks to fly.

Flying ducks would be perfect for the next release of the game.

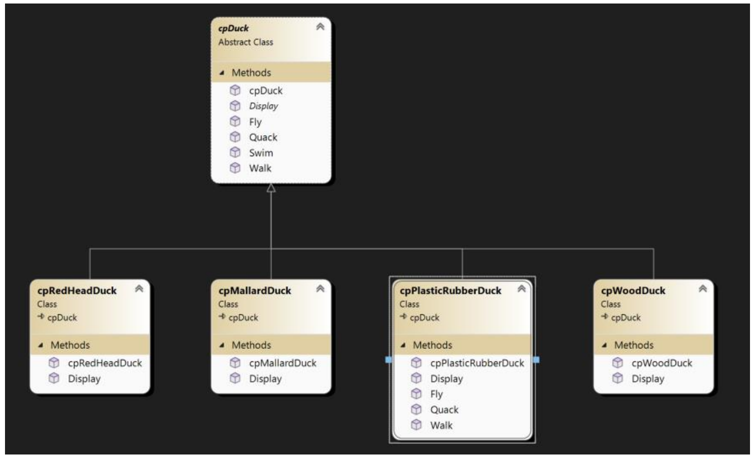
- How would you solve it?

	First think, then execute. *This is really important, that you first plan what you want to do.*

- Implement the solution.
- Adapt your test routine, so the Fly method is also executed.

	Send your solution to Vincent. *Do not continue before you have confirmation that it is correct.*

Notes



Picture 2: Class Diagram Duck Game.

Notes

Part 04

If you did what you have learned, you probably added the method “Fly” to the class “cpDuck”.

It is possible that you used the technique like “Walk”. Because a rubber can’t walk. Look at the virtual and override pattern.

That means that all ducks can fly, but that the rubber duck has an overridden method that mentions that it can’t fly. Just like it can’t walk.

If you did not made an exception for the rubber duck, when somebody test the program, and you have, you will notice that the rubber duck is flying.

	Hey, programmer, I just gave a demo to some potential clients, and there were rubber duckies flying around the screen. *Was this your idea of a joke?*

Extra information

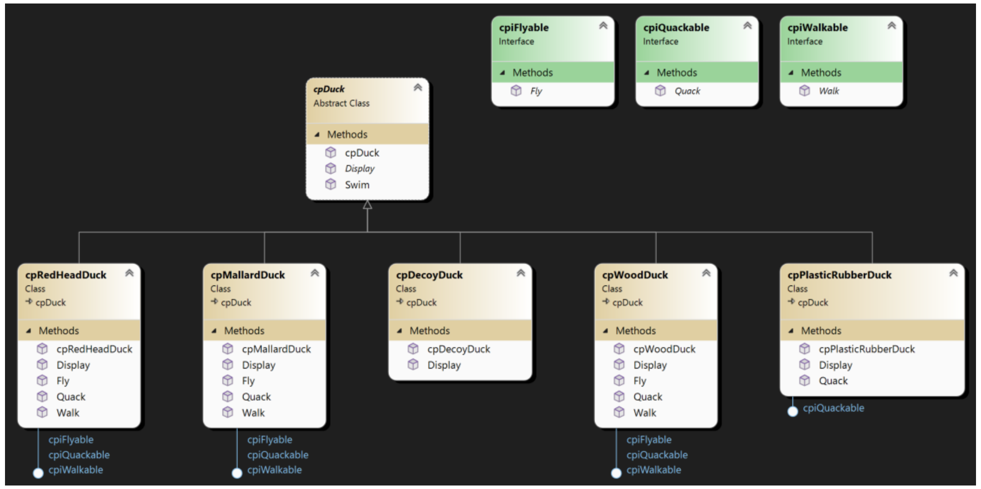
When you have, let’s say, 100 different child classes of a cpDuck, it is hard to put all the override correct. It is boring and a time consuming job.

There are 2 possibilities here.

- You have flying rubber ducks. Solve it.
- You took into account that rubber ducks don’t fly. Great.

	But wait a second. *A rubber duck can’t Quack either. It can be squeaked, but it is absolutely not quacking.* *The Quack() for the rubber duck should be overridden to something that makes a squeaking noise.*

- You have Quacking rubber ducks. Solve it and let it be evaluated.
- When solved it all, you have a class diagram like PICTURE 2.



Picture 3: Class Diagram Duck Game with interfaces.

Notes

Part 05

Think about the solution we have now. And suppose we want to add to our game a decoy duck. That is a wooden duck that will fool other ducks.

Does a decoy duck fly? Does is quack? Can it be squeezed?

Every time a new child of cpDuck will be created, I have to take care of all the members of the base and check it in the child class.

There is another way to take care of this. Interfaces.

Your tasks

You will have a bit of work in doing this.

- Implement an interface that contains the Fly method.
- Implement an interface that contains the Quack method.
- Implement an interface that contains the Walk method.
- Adapt the existing class cpDuck.
  - Remove the fly, quack and walk routines from it.
- Adapt the existing classes cpMallardDuck, cpRedHeadDuck and cpWoodDuck so that the fly, quack and walk functionalities is implemented thru the interfaces.
- Adapt the existing class cpPlasticRubberDuck so that the quack functionality is implemented thru an interface.
- Add a class cpDecoyDuck with the correct interfaces.
  - What can a cpDecoyDuck do?
  - Do you have a picture of a decoy duck?
- Adapt the comments and your test routines (if necessary).
- Test the Fuzzy Duck application.

In the screen beside (*SEE PICTURE 3*), you will find a class diagram, with interfaces to implement the same solution, but using interfaces.

Open questions

	Is this a better technical solution? *Is it a good technical solution?* *Does this solution solve the problem you have after Task 04?* *Was it a lot of work redesigning all the classes?* • *Would this be doable if you had 25 child classes?* *What are the advantages and disadvantages working this way?* …

Notes

....

....

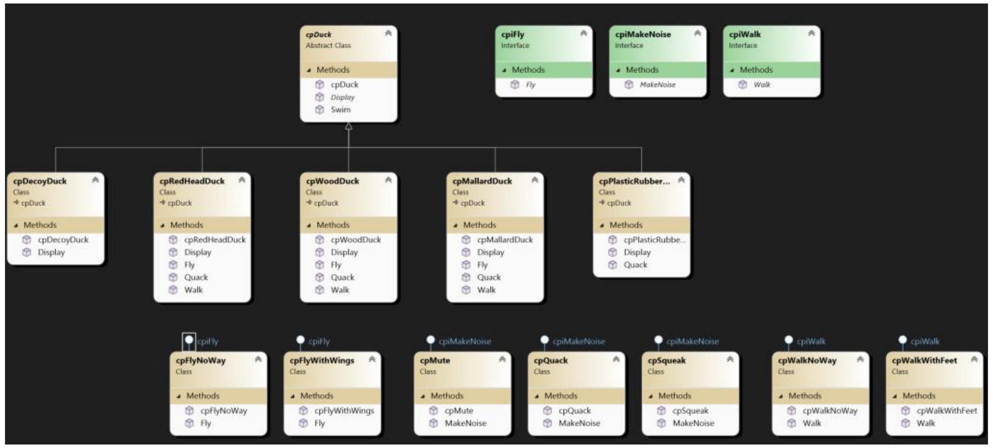
....

....

....

....

....



Picture 4: Class Diagram Duck Game with behaviours (Part 1)

Notes

....

....

....

....

....

....

....

Part 06

It solves a bit of your problem, but it generates a bigger one. You have duplicated code for the fly, quack and walk routines.

Whatever child you make, it has to contain an implementation of the fly, quack and walk methods. And mostly they are the same.

We don't have a flying plastic rubber duck and decoy duck, but the code is not reused anymore.

So the more child classes you have on the cpDuck base class, the more work you will have if something changes.

	And it will change. *If it is not today, it will be changing tomorrow.* *So we want the behaviour out of the cpDuck class and its child classes.*

Open question

How can this be solved?

We will use a good Object Oriented software design principle.

	You will learn a new Design Pattern. *Welcome to the starting point of Events. We will use Delegates.* *You will find the methods that can vary over the child classes and separate them from the methods that stay the same.* *We will encapsulate those methods, so they don't interfere with the methods that don't vary. So in fact, you will pull out the behaviour (methods) of ducks, out of the cpDuck and child classes.* *The pattern can be seen in PICTURE 4.*

Notes

Your tasks

- Implement an interface and some classes that contains all the stuff needed for Flying. I call the interface “cpiFly”, and make a class “cpFlyWithWings” and “cpFlyNoWay”.
    - “cpiFly”, “cpFlyWithWings” and “cpFlyNoWay” will contain all the behaviour for flying. And cpDuck and all the child classes, will use it.
    - The interface already exists, except it will not be directly used by cpDuck anymore. I just renamed it.
- Implement an interface and some classes that contains all the stuff needed for MakeNoise. I call the interface “cpiMakeSound”, and make a class “cpQuack”, “cpSqueak” and “cpMute”.
    - “cpiMakeSound”, “cpQuack”, “cpSqueak” and “cpMute” will contain all the behaviour for making some sound. And cpDuck and all the child classes, will use it.
    - The interface already exists, except it will not be directly use by cpDuck anymore. I just renamed it.
- Do the same stuff for the functionality of walking.

	We don’t do it for Swim. Because all cpDuck and Child classes swim in the same way. *At this moment.* *The moment they are different for some ducks, you will separate the swim stuff in a separated class.*

- Remove all the interfaces used in the classes cpMallardDuck, cpPlasticRubberDuck, cpRedHeadDuck, cpWoodDuck.

	At this moment, your testroutine should still work. *But you will notice some strange stuff now. The implementation of Walk, Fly and Quack are still in the child classes of cpDuck.* *They are not using the interface classes we just created. (See Part 07 to implement this)*

Part 07

We start with the functionality of flying.

Your tasks

- It is possible that you have to change namespaces of the access modifiers.
- If you have put different namespaces, use the using statement.
- If you have private or internal access modifiers, change in public when needed.
- In the class “cpDuck”.
  - Add a delegate with a good name (“cpDelegateFly”).

```
public delegate void cpDelegateFly();
```

  - Add an event with a good name. (“PerformFly”).

```
public event cpDelegateFly PerformFly;
```

  - Add method Fly.
    - Test if PerformFly is null, if not run it.

```
if (PerformFly == null)
{}
else
{
      PerformFly();
}
```

- In the class “cpMallardDuck”.
  - Remove Fly() method. It is moved up to its parent.
  - In the constructor create a variable of data type cpiFly that is a new instance of “cpFlyWithWings”.
  - Add / Delegate the correct functionality to it.

```
cpiFly cpHowToFly = new cpFlyWithWings();
PerformFly += new cpDelegateFly(cpHowToFly.Fly);
```

Notes

....

....

....

....

....

....

....

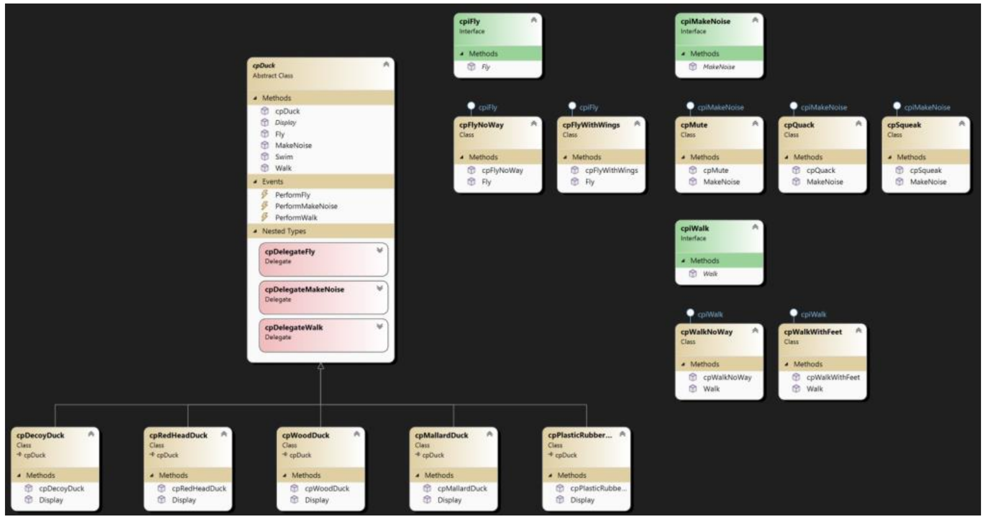
Use the same principles for MakeNoise and Walk in cpMallardDuck.

And do them for also for the other childs of “cpDuck”. So “cpDecoyDuck”, “cpPlasticRubberDuck”, “cpReadHeadDuck” and “cpWoodDuck” will also be changed.

Change also the comments to be coherent with what is implemented.

Change also the test routine if needed. All type of ducks do have now again a Display, Fly, MakeNoise, Swim and Walk functionality.

In the screen beside (SEE PICTURE 5), you see the class diagram that has done this changes.



Picture 5: Class Diagram Duck Game with behaviours (Part 2)

What is the advantage?

With this way of working, using this design, other objects (classes) can reuse our fly, make noise and walk behaviours because they are out of the duck classes.

New behaviours can be added without modifying any of the existing behaviour classes or even touch any of the duck classes. The user of the duck classes can decide what must happen when something flies, makes some noise or walks.

The display and swim functionality can’t be changed at this moment.

This is hard coded inside the duck classes.

To prove this, I have 2 applications that use the library that contains the duck classes and its behaviour.

- A test class.
- A windows form that visualize the ducks.

Notes

....

....

....

....

....

....

....

Part 08

At this moment the functionality of a specific duck is still in the class of that duck. Every constructor in the child classes of cpDuck still contains how the duck should behave.

Mostly this is good enough, but the design pattern that we have, allows us to change this at run time.

The keyword is here at run time. The user of your classes can change this. The developer can add behaviour and can remove behaviour.

The behaviour can change while your program is running. And this is a very nice and cool feature.

In this part we will do several actions:

- We finetune the classes we have.
- The behaviour is now set in the constructors using variables that contain the behaviour.
    - We will change the setting of variables into the setting of properties.
- We add a helper to remove the behaviour that is given by default in the constructor.
    - This is mostly not necessary, but it can be helpful, to ease the work of the developer that will use the classes.
- We change the test routine to change the behaviour of the ducks.
- We change the game application to give the duck another behaviour than the one in the test routine.

	Again a big task, but the design pattern takes form. *After this task, it will be clear how to have the behaviour completely out of the classes.* *When the class needs to execute some behaviour, it is delegated to a class that knows how to execute the behaviour.*

Notes

....

....

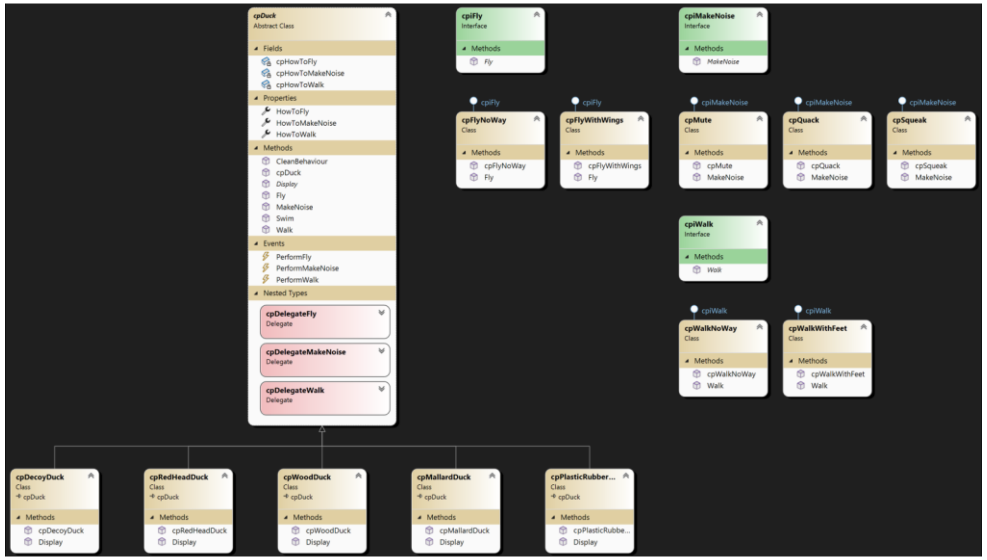
....

....

....

....

....



Picture 6: Class Diagram Duck Game with behaviours (Part 3)

Notes

....

....

....

....

....

....

....

Your tasks

- Every child class of cpDuck contains in the constructor 3 variables.
  - cpHowToFly, cpHowToMakeNoise and cpHowToWalk.
- Move them up to the parent class "cpDuck" and make property (get and set) to allow the developers to change them.
  - Call them "HowToFly", "HowToMakeNoise" and "HowToWalk".
- Move also the assignments to the events towards the properties of cpDuck.
  - Pay attention. Don't forget to remove the previous behaviour (delegate) from the event before adding the new one.

	If you don't remove the previous delegate, your event will trigger two things. *Maybe that is a wanted functionality.*

- Adapt all the classes to use the properties instead of the variables.
- Create a method "CleanBehaviour" in cpDuck, that removes all the functionality that are given inside "PerformFly", "PerformMakeNoise" and "PerformWalk".
  - The basis functionalities are given in the constructor of every duck, but this methods removes that functionality.

	The test program should work without changing it. *The decoy duck does not make any sound at the moment. This is defined in the constructor. The behaviour "cpMute" is used.* *Change this before you loop the duck list into quacking. This is now possible. Also clean the behaviour of the redhead duck.* *When you implement the behaviour inside the classes. The actions above are not possible. This is a huge advantage of this design pattern.*

In the screen beside (SEE PICTURE 6), you see the class diagram.