COMPX102-24B

Assignment 1: Digital Circuit Design

Goals

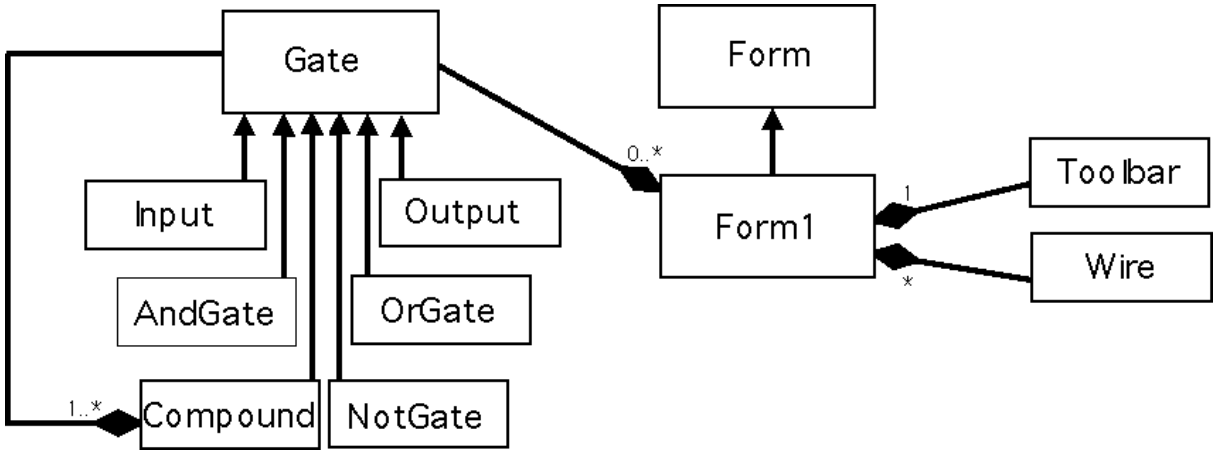
The main goals of this assignment are for you to learn how to write a whole hierarchy of classes, and use subtyping (polymorphism) to make your programs work on just the superclass, rather than all the subclasses. This will simplify your programs and make it easy to add new subclasses. In short, you'll be a far more elegant and sophisticated programmer!

Overview

You are going to write a simple Digital Circuit Design program that electronics engineers could use to design and test simple digital circuits. It will allow you to create various kinds of logic gates (AND/OR/NOT etc.), input sources and output lamps, move them around the design window, add wires to connect them together, and so on. Each logic gate and wire will be represented in the program by an object with methods like Draw, and properties like Left and Top.

You are given a very simple working version of this digital circuits program, which only handles AND gates and wires. Your first task will be to *refactor* this program to use inheritance, so that it is easier to add *new kinds* of logic gates. After you have done this, and added several new kinds of logic gates, you will add *evaluation methods* to the gates, so that each gate behaves like a real digital gate and computes its output signal based on what inputs it is connected to.

The final step will be to add support for *compound* circuits – so that an engineer can design a small circuit, then group all the gates in that circuit into one "uber-gate", which can be moved around, copied and reused as a single object. For example, this could be used to create a *half-adder* gate from several simple gates, then two copies of that half-adder could be combined to give a 1-bit full adder.



Above is a UML class diagram that shows an example of the kind of structure your final program should have (I have omitted Pin and its relationships).

Step 1: Becoming familiar with the example program

Download the example *Circuits.zip* file from Moodle and unzip it in your COMPX102 directory. Open up the *Circuits\Circuits.sln* file with Visual Studio and explore the code and run the program until you understand how it works. Read the documentation at the top of each class to get an overview of the program.

Step 2: Creating an abstract Gate superclass (1 mark)

This is the most important step of this project. You must create a new **Gate** class, then move some of the **AndGate** methods, properties and variables up into your new **Gate** class. Which ones should you move up? The ones that seem to be useful for all kinds of Gates, and are not specific to AND gates. Then change your **AndGate** class so that it inherits from **Gate**. Don't forget to make methods *virtual*.

[Note: It would also be possible to change the Wire class to be a subclass of Gate if you want, but the advantages of doing this seem reasonably minor to me.]

Next change all the references to **AndGate** in the **Form1** class so that they use **Gate** objects instead. (The only call to **AndGate** that you will need to keep is the one in the *buttonAnd_Click* handler, where a new **AndGate** object is created and put into the currentGate variable, which will now be of type **Gate**). Make sure that your program runs correctly, and test its behaviour.

If you find that your **Gate** class needs to define a method (like Draw), so that **Form1** can call it, but the implementation code inside that method is obviously specific to AND gates, then make it an *abstract* method in the Gate class and override it with the correct implementation in the **AndGate** subclass. For example:

```
public abstract void Draw(..);      //in Gate class

public override void Draw(..)
{ ... }                             //in AndGate class
```

Once you have done this successfully, you have *generalized* your **Form1** program so that it can work on all kinds of Gates, not just AND gates!

In the rest of this assignment, we will be adding lots of interesting kinds of new Gates by implementing new subclasses. However, your **Form1** code must not change very much as these subclasses are added.

Typically, you will have to just add one button click handler for each new subclass, which creates an instance of that subclass and assigns it to the `newGate` variable. The goal is to make as few changes to *Form1.cs* as possible throughout the rest of this assignment. This shows how a good object-oriented design allows us to add new functionality to a program by just adding new subclasses to it.

Step 3: New Logic Gates (1 mark each)

Now implement *two new kinds* of Gates: OR gates and NOT gates. Each of these will be a subclass of **Gate**, so must override various **Gate** methods and maybe add some extra methods. Each of these subclasses is worth one mark. Note that your new gates should turn red when selected, just like an AndGate does. As well as OR and NOT gates, you could implement a PAD gate, which is just a small rectangle with one input pin and one output pin.

For each subclass you create, you should add a button to the toolbar with a handler that creates a new instance of your subclass, and assigns it to the `newGate` variable. **Make sure you change the Image property of the toolbar button to the appropriate icon that has been added to the project otherwise it will cause an error.** This allows the engineer to drag it into the circuit and position it with a click.

Step 4: Input Sources and Output Lamps (1 mark each)

Add a new subclass of **Gate** called **InputSource**. This should have no input pins and just one output pin. It should contain a boolean variable that says whether the output pin is currently high voltage (which represents a true value being output) or a zero voltage (which represents a false value being output). An InputSource gate should look different when its boolean value is high. For example, the inside of the gate might be a different colour (avoid red), or it might display a “1” or “0” to show its on/off status. Each time an InputSource gate is selected, its boolean value should *toggle*. That is, selecting it once should change the boolean value from false (the default) to true, then selecting it again should change it from true back to false.

Similarly, create another subclass of **Gate** called **OutputLamp**. This will have one input pin and no output pins. Like the InputSource, it contains a boolean variable that records its current on/off status. It should glow like a small coloured lamp when that variable is true, and be dark when it is false. Add buttons for both of these new subclasses of Gate, and then test that you can add them to circuits and use wires to connect them to other gates.

Step 5: Evaluation Facilities (2 marks)

Now extend your **Gate** class by adding an abstract **Evaluate()** method that returns a boolean result. Each subclass will implement this method in a different way, so that it computes the correct logical result for that kind of gate. For example, the Evaluate() method of an InputSource will just return its internal on/off boolean variable. The Evaluate() method of the AndGate class will return true if both the input pins evaluate to true, or false otherwise. (If an input pin is not connected to a wire, you should display an error message and assume that the pin is false.) The code to do this will look something like this (you should extend this to handle not-connected pins):

```
Gate gateA = pins[0].InputWire.FromPin.Owner;
Gate gateB = pins[1].InputWire.FromPin.Owner;
return gateA.Evaluate() && gateB.Evaluate();
```

The Evaluate() method of an OutputLamp will evaluate its input pin (like gateA of the AndGate code) and set its internal on/off boolean variable to the result of this evaluation. So the output lamps trigger a series of Evaluate() calls to other gates in the circuit, then they display and remember the boolean results of the evaluations.

Add a toolbar button, "Evaluate", whose handler loops through all the gates and asks each OutputLamp to evaluate itself. (Use the C# `is` or `as` operators to find out which gates are OutputLamps). To test your evaluation functions, use your program to design an exclusive-or circuit with two InputSources and one OutputLamp.

Step 6: Copy and Paste of Gates (1 mark)

Now extend your **Gate** class by adding an abstract **Clone()** method that returns a fresh copy of the gate. Each subclass will implement this in a different way, by making a copy of itself. Note that it will also need to make a clone of each of its pins.

Add a "Copy" button to the toolbar, that calls the Clone() method of the currently selected gate (if a gate is selected) and assigns it to the newGate variable. This will allow engineers to copy a single gate. Not very useful, but this feature can become much more useful once you've implemented compound gates.

Step 7: Compound Gates (6 marks)

A very powerful and useful feature of most digital circuit editors is that they allow a collection of Gates to be grouped together into one compound Gate. The resulting compound Gate can then be treated as a new primitive Gate, selected as one whole group, moved around together, and maybe copied and pasted etc.

Implement a **Compound** class, which will be another subclass of **Gate**, but will contain a list of **Gate** objects and an AddGate method that adds

a Gate to that list. Add a "Start Group" button to the toolbar, whose handler creates a new empty Compound gate and stores it in a new instance variable of Form1, called newCompound. If a Gate is selected while newCompound is non-null, that gate is added into the newCompound object. Then add an "End Group" button to the toolbar, with a handler that moves newCompound into newGate and resets newCompound to null.

1. Implement the Compound Move method so that all its gates move together. [2 marks]

2. Implement the Compound Selected property so that either all gates are selected, or none are selected. [2 marks]

3. See if you can implement the Compound Clone method. This is seriously hard, because you need to clone the wires as well as the gates, and make sure that the wires connect between the right gates. [2 marks, 1 mark if only clones gates]

Schedule

Week 7: You should have split the AndGate class into a Gate superclass and an AndGate subclass, and changed the Form1 class to use Gate objects.

Week 8: Implement the other Gate subclasses and the Copy/Paste and Evaluation features.

Week 9: Implement the compound Gates feature. Test and document your program thoroughly.

Marking and Submission: During Week 9, ask a demonstrator in your 102 lab to mark your program, and then submit it. The deadline for submitting this assignment is **Friday 20th September at 5pm,** but you should aim to submit well before this to avoid the rush. Compress your program solution into a .zip file and submit it to Moodle. Both partners should submit the zip file.

This assignment will be marked out of 20. You can earn a maximum of 12 marks for the various features of your program. The remaining 8 marks will be allocated by the tutors, after submission, for good use of inheritance, coding style and documentation. Note that when you override a method in a subclass, we will not require you to document that method, since it will inherit documentation from the superclass definition (which should be documented!).

Questions (add your answers as comments in the form code)

1. Is it a better idea to fully document the Gate class or the AndGate subclass? Can you inherit comments?

2. What is the advantage of making a method abstract in the superclass rather than just writing a virtual method with no code in the body of the method? Is there any disadvantage to an abstract method?

3. If a class has an abstract method in it, does the class have to be abstract?

4. What would happen in your program if one of the gates added to your Compound Gate is another Compound Gate? Is your design robust enough to cope with this situation?

COMPX102-24B

Assignment 1 Hand-in due 5pm Fri 20th Sept 2024

Compress the Visual Studio folder containing your solution program code into a `.zip` file and submit it to Moodle. Both partners need to submit the project zip file.

Student Declaration of Originality

We (the partners listed below) declare that the program that we have had verified and which is submitted in Moodle is entirely our own work. We have not worked together with other people, except for doing pair-programming involving the two partners shown below. We have suitably acknowledged (referenced) any parts of other programs that we used.
We understand that if we have breached the above conditions we will be sent to the University Disciplinary Committee.

Note: This project will be marked only if this Declaration of Originality has been signed by both partners (if doing pair-programming) or by Partner 1 (if the project was done alone).

Partner 1	**Partner 2**
Name: _____	**Name:** _____
ID Number: _____	**ID Number:** _____
Signed: _____	**Signed:** _____
Date: _____	**Date:** _____

Functionality (to be demonstrated in the lab)

Basic Gate 1: And Gate refactor	_____ /1 mark
Basic Gate 2: One of Or/Not/Pad	_____ /1 mark
Basic Gate 3: Another of Or/Not/Pad	_____ /1 mark
Basic Gate 4: Input Source	_____ /1 mark
Basic Gate 5: Output Lamp	_____ /1 mark
Evaluation Facility:	_____ /2 marks
Copy and Paste:	_____ /1 mark
Compound Gate 1: Moving	_____ /2 marks
Compound Gate 2: Selecting	_____ /2 marks
Compound Gate 2: Cloning	_____ /2 marks
Functionality Total (maximum 12):	**_____ /12 marks**

Coding Style & Use of Objects

Objects and Inheritance:	_____ /4 (marked by tutor)
Coding style:	_____ /2 (marked by tutor)
Documentation:	_____ /2 (marked by tutor)
Coding Style Total:	**_____ /8 marks**
Total:	**_____ /20 marks**