1. Write the code for the following program.

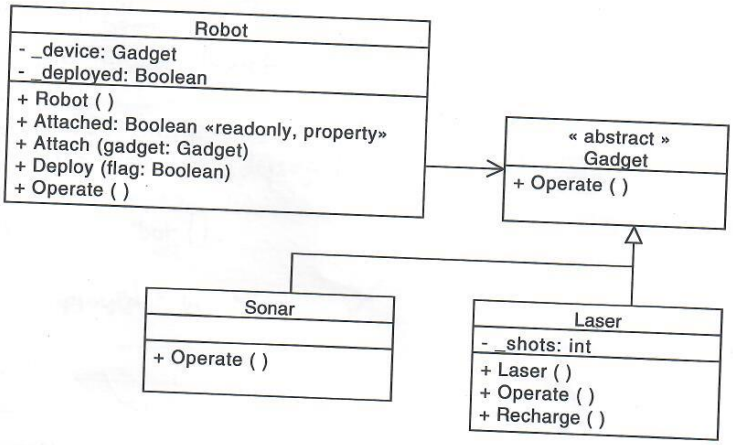


Figure 1: C# Robo-Gadget UML Class Diagram

Robot: A Robot can be equipped with a Gadget using the attach method; the gadget is then stored in the Robot's device field. A Robot can be told to deploy its Gadget via the deploy method, which sets the Robot's deployed field (this field should be set to false in the Robot constructor). When the Robot is told to operate, the following occurs:

1. If the Robot has an attached Gadget and the deployed field is true:
   (a) "Go go gadget" is printed to the console.
   (b) The Robot tells its Gadget to operate.
2. Else ...
   (a) "I'm afraid I can't do that." is printed to the console.

The Robot's attached property is true when its device field is not null.

Gadget: Any device that can be operated; a Gadget can also be attached to a robot. Gadget is an abstract base class.

Sonar: A Sonar is a Gadget that, when operated, prints "Ping".

Laser: A Laser is a Gadget that has energy for 3 shots before it must be recharged. (The constructor sets the shot counter to 3 and calling the recharge method resets the shot counter to 3.) When a Laser is operated, it does one of two things: if the remaining shot count is larger than zero, it prints "Zap!" and the shot count is decremented by one; otherwise it prints "Fizzle.".

Part A.

Write the code for all the classes and interfaces described above. You must follow naming conventions and indent your code appropriately.

Part B.

After completing the code for the system described above, write a small program that creates objects of each of the classes, sets up any collaborations, and calls each of the methods. **Do not write formal unit tests** but do include comments in your code to describe what you are demonstrating. Print the text that would appear on the console when running the program.

```
using System;
Namespace Game
{
public class Robot
{
private Gadget _device;
private Bool _deployed;

public Robot()
{
  _deployed = false;

}

public Bool Attached()
{
  if(_device != null)
  {
    return True;
  } Else return False;
}

public void Attach(Gadget gadget)
{
_device = gadget;
}

public void Deploy(bool flag)
{
  if(flag)
  { _deployed = True;
  }
} else
  _deployed = false;
```

x property
get
{
...
}

```
public void Operate()
{
    if( Attached && _deployed)
    {
        Console.WriteLn("Go go gadget");
        _device.Operate();
    }
    Else
    {
        Console.WriteLn("I'm afraid I can't Do that");
    }
}
```

```
}

Using System;
Namespace Game
{
    public Abstract class Gadget
    {
        public Abstract void Operate();

    }
}
```

```
Using System;
Namespace Game
{
    public class Laser : Gadget
    {
        private int _shots;

        public Laser()
        {
            _shots = 3;
        }
```

```
public override void Operate()
{
    if(_shots > 0)
    {
        Console.WriteLn("Zap!");
        _shots -= 1;
    }
    else
    {
        Console.WriteLn("Fizzle");
    }

public void Recharge()
{
    _shots = 3;
}

}
}

Using System;
Namespace Game
{
    public class Sonar : Gadget
    {
        public override Operate()
        {
            Console.WriteLn("ping");
        }
    }
}
```

Part B

```
Using System;

Namespace Game
{
    public class GameMain
    {
        // testing creation of objects
        Robot R = new Robot();
        Laser L = new Laser();
        Sonar S = new Sonar();
        // testing attaching Gadget to Robot
        R.Attach(L);
        // testing if deploy method deploys field _deployed
        R.DePloy(Attached);
        // testing if properly Operates and prints Go go gadget
        R.Operate();
        // tests if recharge works in filling _shots to 3
        L.Recharge();
        // tests to see if Sonar writes ping into Console
        S.Operate();


    }
```

× Attached is not declared. You need a

and laser output :)

2. The design of the code that you implemented for question 1 has now changed. The Robot is now able to attach to **multiple** Gadgets. Code that uses a Robot object may ask for **a specific** Gadget to be deployed by referring to that Gadget **by name**. Passing an empty string means that no Gadget is deployed.

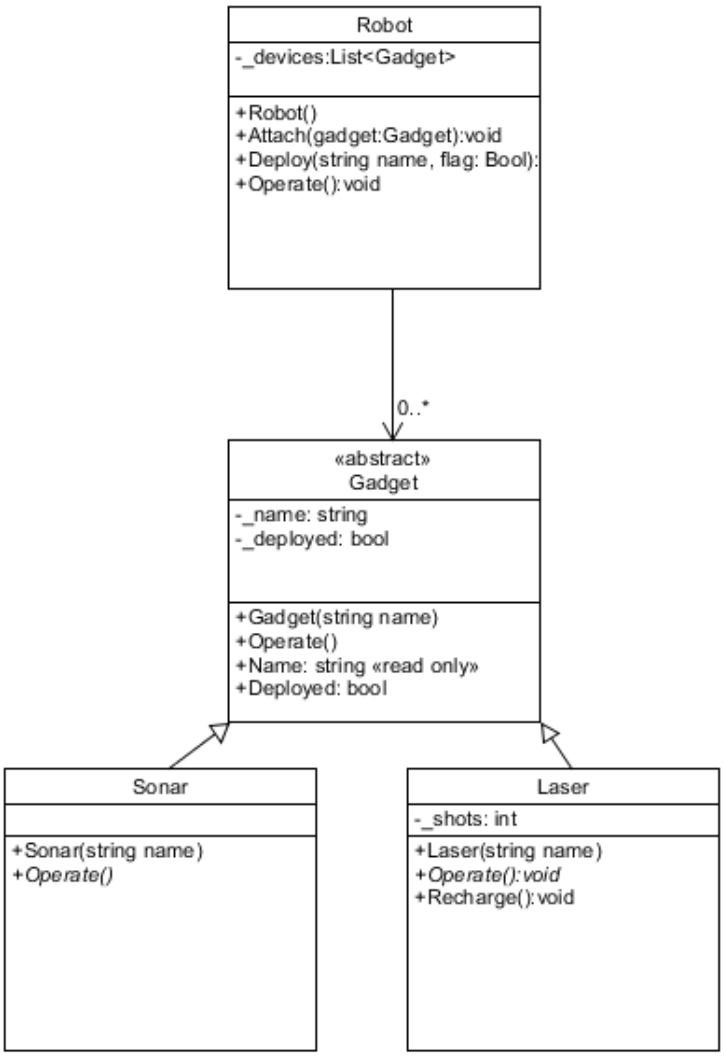
   You are required to

   a) Provide a new class diagram (UML format) that meets the new requirements.

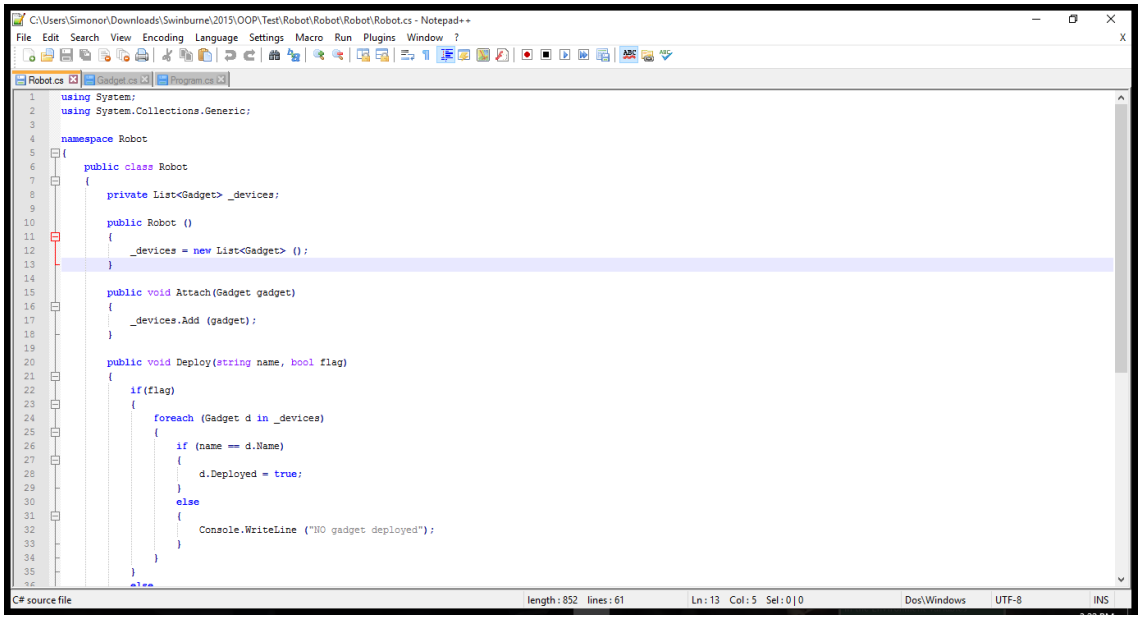
   b) Write the code for all new methods/fields/constructors required.

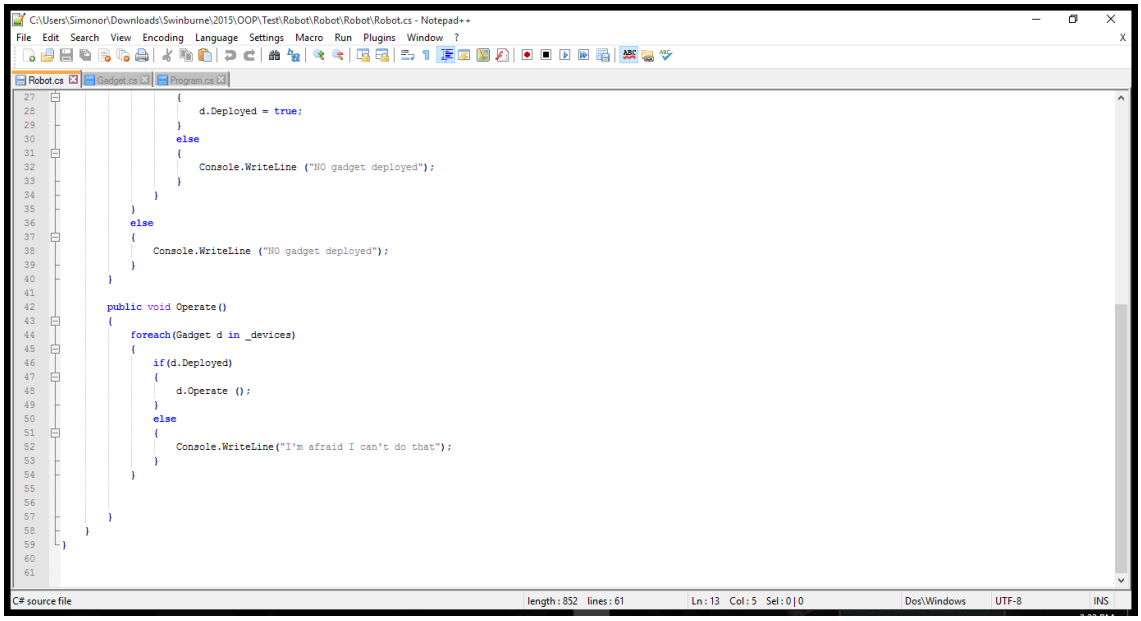
   c) Write a short program that tests this new design.

   Note: Depict only the changes and write only the new code; we will assume that any other code is as presented in Question 1.

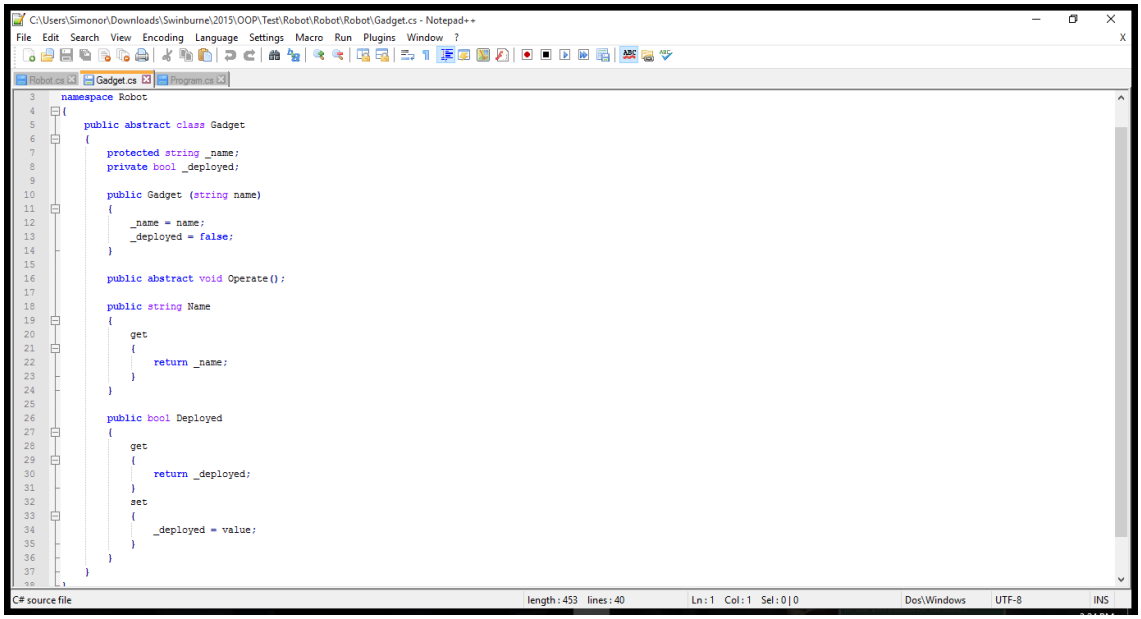


Robot.cs

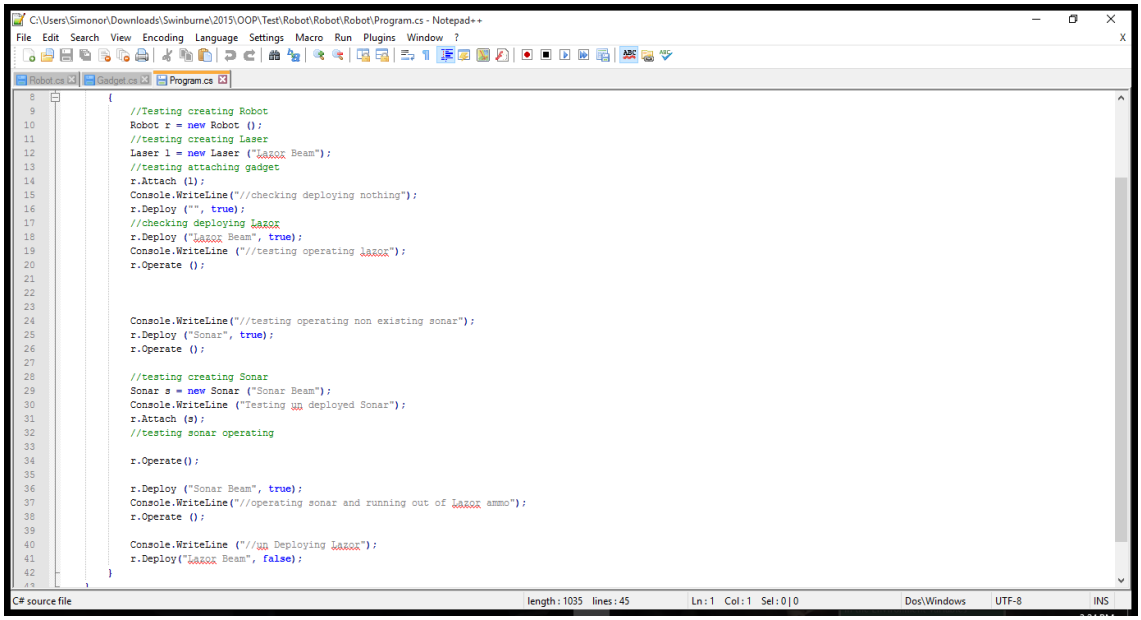

C:\Users\Simonor\Downloads\Swinburne\2015\OOP\Test\Robot\Robot\Robot\Robot.cs - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Robot.cs
Gadget.cs
Program.cs
using System;
using System.Collections.Generic;
namespace Robot
{
public class Robot
{
private List<Gadget> _devices;
public Robot ()
{
_devices = new List<Gadget> ();
}
public void Attach(Gadget gadget)
{
_devices.Add (gadget);
}
public void Deploy(string name, bool flag)
{
if(flag)
{
foreach (Gadget d in _devices)
{
if (name == d.Name)
{
d.Deployed = true;
}
else
{
Console.WriteLine ("NO gadget deployed");
}
}
}
C# source file
length : 852 lines : 61
Ln : 13 Col : 5 Sel : 0 | 0
Dos\Windows
UTF-8
INS


C:\Users\Simonor\Downloads\Swinburne\2015\OOP\Test\Robot\Robot\Robot\Robot.cs - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Robot.cs
Gadget.cs
Program.cs
d.Deployed = true;
Console.WriteLine ("NO gadget deployed");
Console.WriteLine ("NO gadget deployed");
public void Operate()
foreach(Gadget d in _devices)
if(d.Deployed)
d.Operate ();
Console.WriteLine("I'm afraid I can't do that");
C# source file
length : 852 lines : 61
Ln : 13 Col : 5 Sel : 0 | 0
Dos\Windows
UTF-8
INS

Gadget.cs


C:\Users\Simonor\Downloads\Swinburne\2015\OOP\Test\Robot\Robot\Robot\Gadget.cs - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Robot.cs
Gadget.cs
Program.cs
namespace Robot
{
public abstract class Gadget
{
protected string _name;
private bool _deployed;
public Gadget (string name)
{
_name = name;
_deployed = false;
}
public abstract void Operate();
public string Name
{
get
{
return _name;
}
}
public bool Deployed
{
get
{
return _deployed;
}
set
{
_deployed = value;
}
}
}
C# source file
length : 453 lines : 40
Ln : 1 Col : 1 Sel : 0 | 0
Dos\Windows
UTF-8
INS

Testing


C:\Users\Simonor\Downloads\Swinburne\2015\OOP\Test\Robot\Robot\Robot\Program.cs - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Robot.cs
Gadget.cs
Program.cs
{
//Testing creating Robot
Robot r = new Robot ();
//testing creating Laser
Laser l = new Laser ("Lazor Beam");
//testing attaching gadget
r.Attach (l);
Console.WriteLine("//checking deploying nothing");
r.Deploy ("", true);
//checking deploying Lazor
r.Deploy ("Lazor Beam", true);
Console.WriteLine ("//testing operating lazor");
r.Operate ();
Console.WriteLine("//testing operating non existing sonar");
r.Deploy ("Sonar", true);
r.Operate ();
//testing creating Sonar
Sonar s = new Sonar ("Sonar Beam");
Console.WriteLine ("Testing un deployed Sonar");
r.Attach (s);
//testing sonar operating
r.Operate();
r.Deploy ("Sonar Beam", true);
Console.WriteLine("//operating sonar and running out of lazor ammo");
r.Operate ();
Console.WriteLine ("//un Deploying Lazor");
r.Deploy("Lazor Beam", false);
}
C# source file
length : 1035 lines : 45
Ln : 1 Col : 1 Sel : 0 | 0
Dos\Windows
UTF-8
INS

3. Explain the four principles of object oriented programming. For each of the principles, describe a piece of work[1] that you have completed for this unit and explain how it demonstrates the principle. (*Tip:* Consider doing this question last, be concise and list points.)

Abstract
Abstract principle is about defining the structure of your program by breaking it down and removing the unnessassary information. This is normally done by creating UML diagrams. An example of using the Abstract method to create a UML diagram was pass task 12 when we needed to make a UML diagram for the Swinwarts game to include the SpellBook, Spell, Invisibility, Teleport and Heal classes

Inheritance
Inheritance principle is about having a generalised class that has specialised classes sharing what the object knows(fields) and what the object does. (methods) This saves repeating code for similar objects. The children classes (specialised) get information from their parent class (generalised).
An example of this was Task 11 Shape drawer. In the Shape drawer, we created a Shape class that generalised. we then created different shapes such as triangle, cricle, etc. They shared X and Y position and DrawShape

Encapsulation
Encapsulation principle is about what an object knows such as fields and what the object can Do such as methods. Encapsulation is also about what things outside the object know. This includes using private, public and protected. Encapsulation was in all the Tasks
polymorphism
polymorphism is about changing methods from the generalised class for the specific functions of the specified class using abstract, overrides and virtual. We using this in the Shape Drawer Fu change what each different shape Drew.

[1]This piece of work can be anything you have done for this unit, though it is expected to be included in your portfolio.

1/2 what about other types of Polymorphism?

Polymorphism
polymorphism is the process of child or specialised classes possessing methods from the parent/Generalised class and modifying them to perform different behaviours.

Polymorphism heavily depends on inheritance as classes need to inherit the methods to be able to change them. There are 4 different types of polymorphism. Subtype, Adhoc, Parametric and Coercion. Subtype is the main type of polymorphism as it is using derived classes and base classes to change methods. Parametric gives a way to execute code for different types. adhoc allows you to use methods to act differently for different types. Coercion is used when an object is cast into another object.
An example of using polymorphism in the tasks was the Shape Drawer Program. In the shape drawer program each shape required to be drawn so we created a DrawShape method in the parent class. Polymorphism was needed to make the class abstract so we could overrule the method in the child classes so they could each perform SwinGame's unique drawcircle, drawrectangle and drawline methods.