Session 5. Inheritance

In this session we examine the last remaining class relationship, inheritance, and concepts related to inheritance. Inheritance represents a key mechanism for improving code reuse, i.e., the ability to avoid developing functionality that is similar to existing code. We also address the question of what makes for a good object-oriented design? Preparing an object-oriented design is not straightforward, and even the most experienced designers can get it wrong. Although there is no methodical approach to developing a good design, we can identify elements of bad design and learn to avoid them.

Session Objectives

In this session, you will learn:

- The concept of reusability, how it is represented in object-oriented programs, and how to increase the likelihood your code will be reused;
- The concept of inheritance, how it is applied in object-oriented programs, and when to use it; and
- How to identify potential areas of poor/bad design in a program.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*
  We continue developing our knowledge of object-oriented concepts, focusing on inheritance.
- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*
  We continue developing our understanding of object-oriented techniques and how they are applied in the C# programming language in this session.
- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*
  We will not be directly addressing this ULO, however understanding of the C# programming language will contribute to the skills required for this outcome later.
- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*
  We will not be directly addressing this ULO, however the knowledge gained from this session is directly related to the design of object-oriented applications.

Required Reading

Additional reading is required from the textbook Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014, as follows:

Required Reading: Chapter 11

This chapter introduces the basic concepts of inheritance and how to implement it in the C# programming language. Inheritance is a critical concept in any study of object-oriented development.

Required Tasks

The following tasks are mandatory for this week:

- Task 5.1. Protected Encapsulation
- Task 5.2. Find Your Inheritance

5.1. Introduction

In this session we continue our study of object-oriented development through the study of inheritance, one of the major contributors to reusability in object-oriented programs. We begin in Section 5.2 by considering what reusability actually is, why it is beneficial in software development and how to encourage reusability in your programs. We then examine inheritance in Section 5.3, which is one of the most critical concepts to understand for developing object-oriented applications. Unfortunately, inheritance is also often misused, so we also examine the question of when to use inheritance and what the alternatives are, covering substitutability and delegation in Section 5.4. Finally, one of the problems faced by many programmers starting to learn object-oriented development, is how to tell if a design is good or not. To help develop a better understanding of good and bad design, we examine “bad smells in code” in Section 5.5.

5.2. Software Reusability

Reusability refers to the ability to exploit (reuse) previously developed code in the construction of new solutions and is often promoted as one of the big advantages of the object-oriented development approach. The ability to reuse previously developed code offers a number of advantages, for example:

- A reduction in development time – reused code is already complete and does not need to be developed from scratch;
- A reduction in testing time – reused code has usually been tested thoroughly and can be relied upon in a new project;
- A reduction in time/effort to maintain existing code – bug fixes, etc., to code can quickly and easily be carried to all projects in which it was reused; and
- Improved quality of code – code that has been developed to be reusable will usually have been more carefully designed, coded, and tested.

Moreover, reusability reduces the risks associated with developing software (risk of late delivery, risk of bugs, etc.). However, there are also disadvantages and/or concerns with developing reusable code, for example:

- Reusable code takes longer to develop than purpose-built code (it is often said that it takes three times longer to develop reusable code);

- Reusable code is sometimes rarely, if ever, reused, thus wasting the extra effort to develop it in the first place;
- Reusable code can take a long time to learn and/or adapt for (or plug-in to) a new for project;
- Reused code can be restrictive, i.e., if the code does not support a particular feature you may not be able to extend/easily extend that code to support that feature; and
- Bugs in reused code will be present in all projects using that code.

Reusability is seen in object-oriented development through inheritance and delegation (see Section 5.4). In developing classes it is possible to encourage and improve reusability by following a number of simple style rules[1]:

- Keep methods coherent – well written methods have a very clear and distinct purpose in a program. A well written method will perform either a single function or a group of closely related functions;
- Keep methods small – a very long method (greater than one or two pages) is usually a sign of a method that is doing a very specific function that is not particularly reusable. If the function can be broken into smaller parts, it is possible that those individual parts may have a higher level of reusability even if the whole method was not;
- Keep methods consistent – methods that perform the same kind of function should generally have the same method names, argument order, data types, return value, and semantics;
- Separate policy and implementation – keep the code that makes decisions (policy) and the code that performs the logic of the program (implementation) separate methods. Policy methods are application specific whereas implementation methods may be reusable by other applications;
- Provide uniform coverage – if input can be provided in different formats/combinations, provide methods to handle all possibilities, not just those formats that immediately required;
- Broaden the method as much as possible – often the reusability of a method can be increased substantially with very few actual modifications, e.g., modify the method to accept different types of data, make fewer assumptions about how the method will be used, provide more meaningful results for empty/extreme/invalid inputs, and so on;
- Avoid global information – avoid/minimise use of data outside of a method as this can change the functionality of the method depending on that external data. This can often be corrected by passing the data as a parameter instead;
- Avoid methods with state – methods that change depending on how they have been used previously are unlikely to be reused. This dependency on method call history can often be eliminated by replacing the method with several methods.

[1] Blaha, M., and Rumbaugh, J., "Object-Oriented Modeling and Design with UML", Second Edition, Prentice-Hall, 2005.

5.3. Inheritance

In Section 4.2.2 we examined three class relationships: association, aggregation, and composition. ***Inheritance*** is also a class relationship, but unlike the others it does not result in an object-relationship, i.e., it does not result in a link between objects. This is because the inheritance relationship provides us with the ability to construct new classes by building on top of, or extending, existing classes. The result of an inheritance relationship is a new class from which a single object can be instantiated.

As for other class relationships, there are two classes involved in an inheritance relationship: the ***base class***[2] and the ***derived class***[3]. The derived class inherits all of the functionality defined in the base class, i.e., all of the attributes and operations, before adding additional attributes and operations to "specialise" that class. A collection of several classes related through inheritance are referred to as an ***inheritance hierarchy***.

Consider the simple diagram shown in Figure 5.1, which shows two classes: the Person class (base class), which is inherited by the Student class (derived class), and two objects, one instance of the Person class and one instance of the Student class. The Person class shows two attributes for a name and address, which are reflected in the Person object. The Student class then specialises the Person class by adding an ID number, results, and methods to Enrol in and Study a unit. The Student object shows how this specialisation works, where the Student object shows the attributes both from the Person class (name and address) and from the Student class (ID number and results). Importantly, the above example only shows the Person class being specialised in one way, you could also separately define other classes that derive from the Person class, e.g., a Staff class which adds staff ID, salary, etc., as we will see shortly.

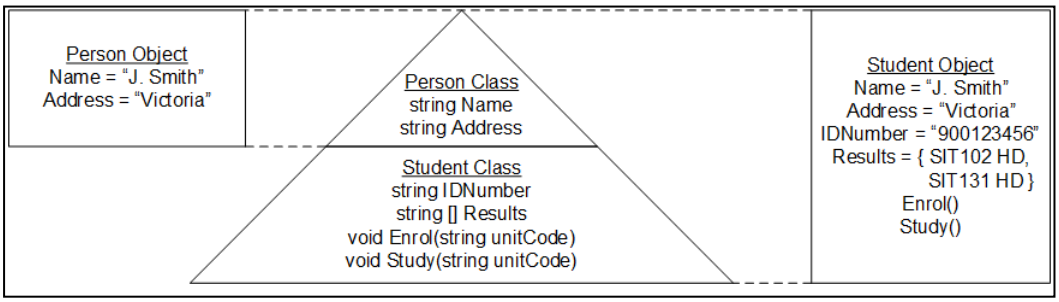

Figure 5.1: How Inheritance Works

There are two views of inheritance that can be used to help gain a better understanding of inheritance: ***generalisation*** and ***specialisation***. Generalisation refers to the base class being a more general version of the derived class. Specialisation, as suggested above, refers to the derived being a more specific version the base class, or makes it more specific, by introducing new features (members).

[2] Also referred to as the super-class, parent class, or ancestor class.

[3] Also referred to as the sub-class, child class, or descendant class.

Figure 5.2 shows two classes (Student and Staff) modified to exploit inheritance using an additional class (Person), drawn in UML class diagram notation. Although we don't examine UML notation in detail until Section 7.3, it is simple enough to understand, and represents classes and inheritance clearly. Each class is represented by a box divided into three sections: class name, attributes and properties[4], and methods. The '+' symbol indicates a class member that is public, data types are indicated after a colon (':'), and inheritance is indicated by an arrow (unshaded), where the head of the arrow points to the base class and the tail/s of the arrow connect to the derived class/es (thus in this example Person is the base, Student and Staff are derived).

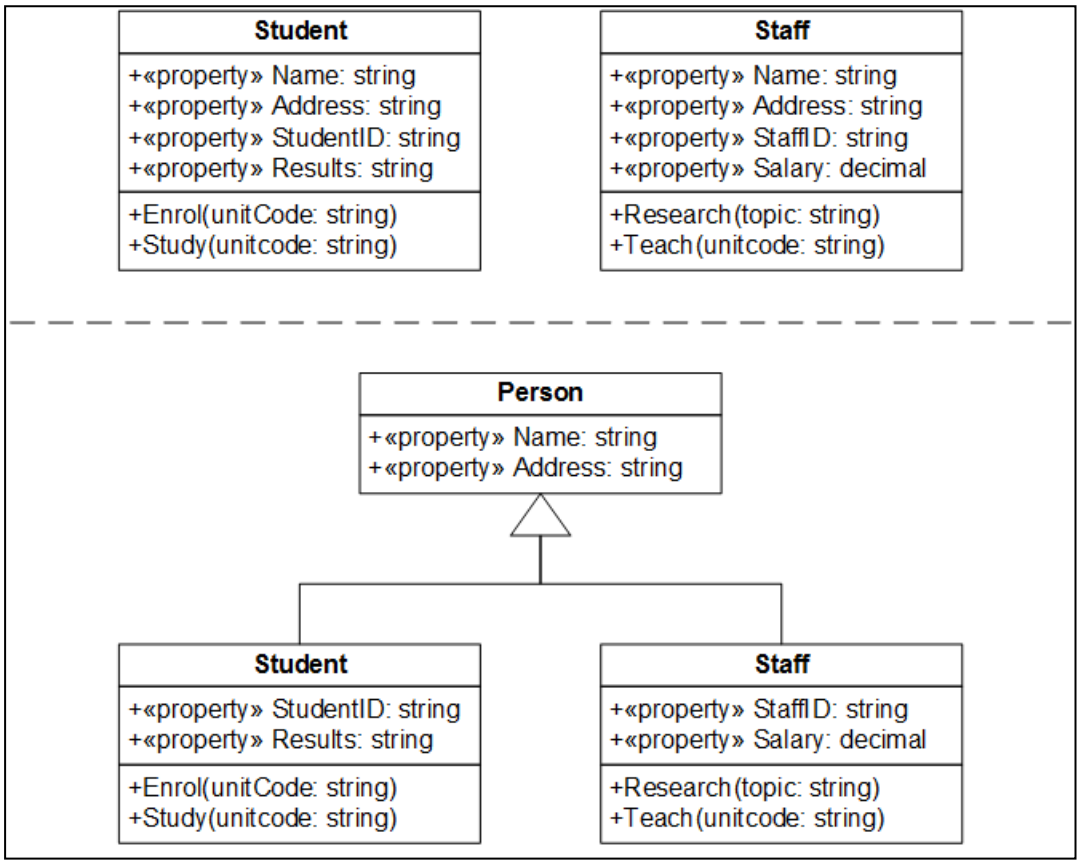

Figure 5.2: Generalisation and Specialisation in Inheritance

In the top section of Figure 5.2, two classes are shown: a Student class and a Staff class. In examining these two classes, you will note that there two common properties: Name and Address. By exploiting inheritance we can extract these properties (and their associated instance variables) to a more general base class, the Person class. This is what is shown in the bottom section of the figure[5]. In the bottom section of the figure, the Person class represents a more generalised version of the Student and Staff classes, and the Student and Staff classes represent more specialised versions of the Person class.

[4] Attributes are not illustrated in this UML class diagram as their illustration would not improve understanding here. From our previous studies you should already be aware that properties do not have any storage space associated with them and require an associated instance variable for this purpose.

[5] Note also the Student class inheriting from the Person classes in the bottom of Figure 5.2 also matches the illustration in Figure 5.1.

Recall from Section 4.2.2 that use of the phrase 'part of' when discussing an aggregation relationship. In a similar manner, we can use the phrases 'is a' and 'is a kind of' for inheritance[6], e.g., a Student is a Person, and a Staff is a kind of Person. The use of these phrases also allow us to test whether or not inheritance is appropriate using common sense. For example, the above phrases clearly make sense, however the phrase "a Unit is a Student" does not, implying that inheritance is not appropriate between the Unit and Student classes.

Inheritance is expressed in C# using the following syntax:

```
[access_modifier ]class  derived_class_name : base_class_name
{
        [access_modifier ]class_member
        ...
}
```

An example of Inheritance in C# is shown in Figure 5.3, which demonstrates a number of new concepts. In the figure, there are two classes: `Account` and `SavingsAccount`. The `Account` class defines an attribute for the account balance, which is encapsulated by a read-only property and a custom constructor is provided to initialise its value. A `ToString()` method has also been defined to show the account balance. The `SavingsAccount` class inherits from the `Account` class, and contains a constructor to initialise the balance and `Deposit` and `Withdraw` methods to process transactions against the account balance. The `Deposit` method increases the account balance, and the `Withdraw` method decreases the account balance but only if there are adequate funds in the account returning a boolean value to indicate success/fail of the withdrawal.

```
class Account
{
    protected Account(decimal balance)
    {
        _Balance = balance;
    }

    protected decimal _Balance;
    public decimal Balance
    {
        get { return _Balance; }
    }

    public override string ToString()
    {
        return string.Format("Balance: {0:c}", _Balance);
    }
}

class SavingsAccount : Account
{
    private const decimal DEFAULT_BALANCE = 0.00M;
```

[6] The phrase 'is like' may also be used, however this phrase does not suggest the more strict semantics that are expected of inheritance.

```
    public SavingsAccount()
        : this(DEFAULT_BALANCE)
    {
    }

    public SavingsAccount(decimal balance)
        : base(balance)
    {
    }

    public void Deposit(decimal amount)
    {
        _Balance += amount;
    }

    public bool Withdraw(decimal amount)
    {
        bool result = true;

        if (_Balance >= amount)
            _Balance -= amount;
        else
            result = false;

        return result;
    }
}
```

Figure 5.3: Example of Inheritance in C#

The first thing to consider is how objects are built when inheritance is involved. When creating an object from a class that inherits the features from another class, the object is constructed beginning with the base class members (whose initialisation is defined by the base class' constructor), then the derived class members(whose initialisation is defined by the derived class' constructor). A key factor to consider here is that constructors are in fact not inherited in the C# programming language[7]. The `Account` class defines a constructor with a parameter to initialise the balance. As there is only one `Account` class constructor provided that requires a `decimal` parameter (the opening balance), this value must be provided somehow when creating a `SavingsAccount` object.

Two constructors are defined in the `SavingsAccount` class, one which requires no parameters, and another that has a matching parameter list to the `Account` class' constructor. Note the object initialisers specified for these constructors, for which for the custom constructor's object initialiser shows a new keyword '`base`' instead of the keyword '`this`' that we are familiar with (Section 4.4). The two keywords are similar in functionality, except the keyword `this` is used for passing data to a constructor defined in the same class, whereas the keyword `base` is used for passing data to a constructor defined in the base class. In the case of the `SavingsAccount` class, the parameter-less constructor invokes the custom constructor in the `SavingsAccount` using the keyword `this` and providing a default zero balance, which in turn invokes the custom

[7] Other programming languages, such as C++, do allow constructors to be inherited so you will need to check the documentation for any language you are using.

constructor in the `Account` class using the keyword `base`. Note that if the `Account` class provided a parameter-less constructor, or if no constructor was provided (leaving only the default constructor), no call to the base constructor would be needed (a call to a parameter-less/default constructor is assumed).

Another point worth noting is the introduction of the `protected` access modifier, which was first mentioned in Section 2.3. Until now, we have only been using the `private` and `public` access modifiers. To explain how the `protected` access modifier, recall that a `public` class member can be accessed by code anywhere in the program whereas a `private` class member can only be accessed by code in the same class. The `protected` access modifier effectively sits between these two, where `protected` class members can only be accessed by code in the same class or a derived class. The `protected internal` access modifier represents the union of the `protected` and `internal` access modifiers, i.e., code in the same class, code in a derived class, or code in another class within the same assembly can access the member.

Given this definition of the `protected` access modifier, it should now be apparent that the constructor that appears in the `Account` class can only be invoked by derived classes, or by another constructor in the `Account` class (of which there are none). The lack of another constructor in the `Account` class means that it is actually not possible create an instance of the `Account` class outside of a class that derives from the `Account` class. Note that it is actually possible to prevent any instance of a class being created through the use of abstract classes, as discussed in Section 6.3.

It is also worth noting that the interface for the custom constructor in the `Account` class has the same syntax as the custom constructor in the `SavingsAccount` class. There is no requirement for this, just like any other method there is no required ordering of parameters, and the names do not need to match. However recall from Section 5.2 that methods should be consistent where possible, keeping the same names, argument order, data types, return value, and semantics. For this reason, it is usually a good idea to keep the same parameter list.

The `_Balance` attribute of the `Account` class is also declared as `protected`, to which a `public` interface for reading the balance is provided in the `Account` class while derived classes are still able to access and modify its value. Importantly, there is no way to hide the public read-only property `Balance`, thus it is important when writing base classes to ensure public members are applicable to all derived classes[8].

Additionally, recall the discussion from Section 2.5 that instance variables should be declared as `private` to allow the representation of data to change throughout the lifetime of the application without such changes impacting upon other classes. Strictly speaking, the same concept applies here. This can be achieved by declaring the `_Balance` attribute as `private`, then providing a `protected` interface for modifying the `_Balance`

[8] Some object-oriented programming languages do in fact allow you to change the protection level of base class members inside a derived class. However, before deciding this a disadvantage of C#, you should review and understand the concept of substitutability in Section 5.4. Substitutability tells us that any need to change the protection level of a base class member implies that inheritance is being applied incorrectly.

attribute. This is examined in Task 5.1, below. However there is also a counter argument to encapsulating instance variables for inheritance in this manner – it is unlikely that anyone other than the original developer will ever see the protected members, and given that developing such an interface would take time, there is a monetary cost associated with such an approach.

The final point to note from Figure 5.3 is the implementation of a `ToString()` method in the `Account` class, which is inherited by the `SavingsAccount` class and applies the same to objects of both classes. It is actually possible to specialise this functionality by defining a new `ToString()` method in the `SavingsAccount` class that accesses the `Account` class `ToString()` by invoking `base.ToString()`.

To demonstrate the remaining elements of inheritance in C#, we introduce an extension to the Figure 5.3 example, as shown in Figure 5.4. In this extension, we add a new class, `CreditCard`, which extends the `Account` class by adding an attribute added for a credit limit (`_Limit`) and two methods for recording payments (`Payment`) and purchases (`Purchase`) made with the credit card. Given that a credit card deals with money owed, rather than money invested, the balance value is negated (`-`) several times to provide sensible values to the user.

```
class CreditCard : Account
{
    public CreditCard(decimal balance, decimal limit)
        : base(balance)
    {
        _Limit = limit;
    }

    private decimal _Limit;
    public decimal Limit
    {
        get { return _Limit; }
        set { _Limit = value; }
    }

    public void Payment(decimal amount)
    {
        _Balance += amount;
    }

    public bool Purchase(decimal amount)
    {
        bool result = true;

        if (-_Balance + amount <= _Limit)
            _Balance -= amount;
        else
            result = false;

        return result;
    }

    public new decimal Balance
    {
        get { return -_Balance; }
```

```
    }

    public override string ToString()
    {
        string result;

        if (_Balance < 0)
            result = string.Format("Balance: {0:c} owed", -_Balance);
        else if (_Balance == 0)
            result = base.ToString();
        else
            result = base.ToString() + " in credit";

        result += ", limit " + _Limit.ToString("c");

        return result;
    }
}
```

Figure 5.4: Example of Inheritance in C# (continued)

The constructor now has an additional parameter for the limit, which is used to initialise the `_Limit` instance variable, however the opening balance is still passed to the constructor in the base `Account` class. The `ToString()` method is somewhat more complex, potentially returning three different strings depending on the balance (assuming $5 owed or in credit and a limit of $500):

- For a negative balance: `Balance: $5.00 owed, limit $500.00`
- For a zero balance: `Balance: $0.00, limit $500.00`
- For a positive balance: `Balance: $5.00 in credit, limit $500.00`

Of particular interest are the lines that add the credit limit. You will note that the `ToString()` of the instance variable `_Limit` has been invoked with a parameter of `"c"`. This parameter has the identical effect as a `:c` used in a format item for composite formatting, resulting in the `decimal` value being converted for display as a currency value.

Finally of note is the definition of the `Balance` property, which has been provided to ensure that a negative balance is not returned for an amount owed. Recall from Figure 5.3 that the `Account` class already defines a `Balance` property, so the definition that appears in the `CreditCard` class will is effectively 'hide' the property in the `Account` class. The C# programming language mandates the use of the keyword `new` whenever one member would hide another member in the base class, which is why it is used here. For methods, this is referred to as ***method hiding***[9].

[9] Note that in C++ this is called function overriding. Method overriding in C# however generally refers to the use of the `override` keyword, as used for the `ToString()` method, which is relevant to polymorphic methods that we examine in Section 6.2.

Task 5.1. Protected Encapsulation

Objective: As discussed above, there is some debate as to whether instance variables should be allowed to be protected. In this task you will explore the difference between the two by converting from one approach to the other.

In this task we explore the basics of inheritance and use of encapsulation for class members that are visible to child classes but not other classes, i.e., protected class members, using the examples in Figure 5.3 and Figure 5.4, above. The full code for these classes can be obtained from the Code Examples file for this week in CloudDeakin. Your tasks are as follows:

a. In the `Account` class, change the access modifier for the `_Balance` property to `private`.
b. At this point, the program no longer compiles because the derived classes (`CreditCard` and `SavingsAccount`) were using this variable. Correct this problem by **adding one or more** `protected` members to the `Account` class to allow the derived classes to modify the balance, replacing the direct references to the `_Balance` instance variable in those classes as required.
c. Add a new class, `TermDeposit`, that satisfies the following requirements:
    i. Inherits from the `Account` class;
    ii. Defines a private instance variable to store the duration (i.e., term) of the term deposit, measured in days;
    iii. Encapsulates the above instance variable in a public property;
    iv. Defines an appropriate constructor method which allows the opening balance and term to be specified;
    v. Defines an appropriate `ToString()` method;
    vi. Defines a method `CalculateInterest()` with a single parameter, interest rate, which calculates and returns the interest for the term deposit (only one interest payment for the entire term deposit);
d. Update the `Main()` method to demonstrate the functionality of your new class.

Hint: Note that the interest can be calculated using the formula:

5.4. Substitutability and Delegation

It is critically important not to over use inheritance in your solutions, a common mistake often made by inexperienced programmers. In Section 5.3 we identified the phrases “is a” and “is a kind of” that can be used to test whether or not inheritance can be used between two classes or not. A more strict test that can be applied is that of substitutability, also known as the ***Liskov Substitution Principle***. Under this principle, inheritance should only be used if for every possible statement where an object of the base type could be used, an object of the derived type could equally be used.

Consider two classes: a `Circle` and an `Ellipse` class, where the `Circle` class inherits from the `Ellipse` class. Considering the geometry of a circle and an ellipse, it would be possible to say that a circle is simply an ellipse with a constant radius, suggesting that the phrase "a Circle is an Ellipse" is acceptable. However, the Liskov Substitution Principle states that for correct inheritance, an object of a derived type must be able to be substituted wherever an object of the base type can be specified, i.e., wherever an `Ellipse` object appears in code, a `Circle` object could appear in its place. This test does not hold in this case, for example, it would be possible to set the width and height of an `Ellipse` object to different values, however this operation would not be possible for a `Circle` object as the height and width of a `Circle` must be equal.

There are three solutions to this problem:

i. Eliminate any relationship from between these classes, potentially resulting in duplicate code;
ii. Introduce an additional class as a common base, e.g., an `Ellipse` is a `RoundedShape`, a `Circle` is a `RoundedShape`; or
iii. Apply delegation.

Eliminating any relationship between these two classes certainly eliminates the problem of inappropriate inheritance in this solution. However, it also fails to exploit the common ideas shared by the classes. A better solution is to introduce an additional class such as a `RoundedShape` class, as suggested above, which both the `Circle` and `Ellipse` classes inherit from.

The third alternative, ***delegation***, exploits an aggregation/composition relationship instead of inheritance. In this case, there is an aggregation between the `Circle` class and the `Ellipse` class, where `Circle` is the aggregate (whole) class, i.e., an `Ellipse` is part of a `Circle`. The implementation of the methods of the Circle class are then delegated to the `Ellipse` class, e.g.,

```
class Circle
{
    ...
    private Ellipse _Ellipse = new Ellipse();
    ...
    public double Area()
    {
        return _Ellipse.Area();
    }
    ...
}
```

In the above example, the `Area()` method of the `Circle` class performs no functionality other than to invoke the `Area()` method of the `Ellipse` class and return its result. Other methods are implemented similarly, except for those that set the dimensions of the `Circle`, which enforce a constant radius/equal width and height.

Task 5.2. Find Your Inheritance

Objective: Now that we have examined the concepts of inheritance, you will explore inheritance on your own in this task.

Consider the following problem carefully:

> *The local library offers a number of different items for loan, including books and periodicals (magazines). Books have an author, title, publisher, and a dewy decimal number. Periodicals have a title, publisher, edition, and a dewy decimal number. A Patron can borrow up to two of each type, up to a maximum of three items.*

Your task is to identify the inheritance in the above program and implement a program according to the above information. Note that you will require two methods in your `Patron` class: `BorrowBook()` and `BorrowPeriodical()`. Write an appropriate `Main()` method that creates a `Patron` object and a number of `Book` and `Periodical` objects and effectively tests/demonstrates your program.

5.5. What is Good Design?

One of the most difficult problems many programmers new to object-oriented design experience is the question of exactly what is “good design”. The primary problem faced by new programmers, is that for any problem there are often many different models that could be equally valid. Unfortunately this problem does not entirely disappear through experience, however the problem is often more a case of maintaining a good design in the face of changes.

Most large and complex applications do not begin their life that way. They begin as relatively simple problems, and evolve over their lifetime to reach a larger size and/or complexity. Unfortunately, this usually means that the design approach taken for the original program are not necessarily appropriate for the new program. As a result of this change in design requirements, the field of ***refactoring*** has come about. Simply put, refactoring is the process of modifying the design of an existing program to either improve the existing design, or to carry out necessary design changes to facilitate more functionality.

We undertake a more detailed study of refactoring in Section 6.7. However one of the primary sources for information on refactoring offers a list of “bad smells in code” which can suggest potential areas for improvement in a program. These “bad smells” may help you to identify areas in your program/design that could use improvement. However, note that improving the design of a program is not straightforward – some of the areas identified below are in fact mutually exclusive and/or opposites of each other. This emphasises that the best design for one program is not necessarily the best design for another program. Here are some examples:

- Duplicated code – already considered in Section 4.4, where the same code appears in more than one place in your program;

- Long method – already considered in Section 5.2 for reusability, a method containing a large number of statements can also suggest the need for design improvements;
- Large class – similar to a large method, a class that has a large number of members can often lead to other problems such as duplicate code and will usually benefit from modularising the class using aggregation/composition;
- Long parameter list – a method with many parameters can often have more parameters than are often required, as some information can be obtained from elsewhere, such as an object (avoiding global information from Section 5.2 may conflict with this however);
- Divergent change – if one class needs to be modified for two or more very different reasons, that class may need to be split into two separate classes;
- Shotgun surgery – the opposite of divergent change, is when to make one change you need to modify many different classes, which may suggest that functionality needs to be moved between different classes or a new class created;
- Feature envy – a method defined in one class constantly working with the members of another class may suggest that the method actually belongs in the other class;
- Data clumps – when several pieces of data are seen together in different areas of a program, it may be better to extract those data elements to a new class which replaces those data elements using aggregation/composition;
- Primitive obsession – programmers new to object-oriented will often be hesitant to create very small classes, preferring instead to use simple types (primitive types). If such simple types are seen regularly, e.g., minimum/maximum values for a range, it may be worth extracting these data elements to a separate class;
- Switch statements – if you find you are regularly modifying several switch statements in a program for the same change, it may suggest that inheritance and polymorphism could be used to eliminate the switch statements entirely (we examine polymorphism in Section 6.2);
- Parallel inheritance hierarchies – whenever adding a subclass for a change you always need to add a subclass to a second hierarchy may suggest the need to combine several classes (also see shotgun surgery, above);
- Lazy class – a class with very little functionality to perform can often be eliminated to improve the design;
- Speculative generality – sometimes classes are made too general, providing functionality that will never be used but was added “just in case”, and removing this functionality may improve the overall design (broadening the method as much as possible from Section 5.2 may conflict with this however); and
- Temporary field – an instance variable (field) that is only occasionally set (used) in a class may be able to be moved to a separate class (linked through aggregation) along with the methods that use that variable.