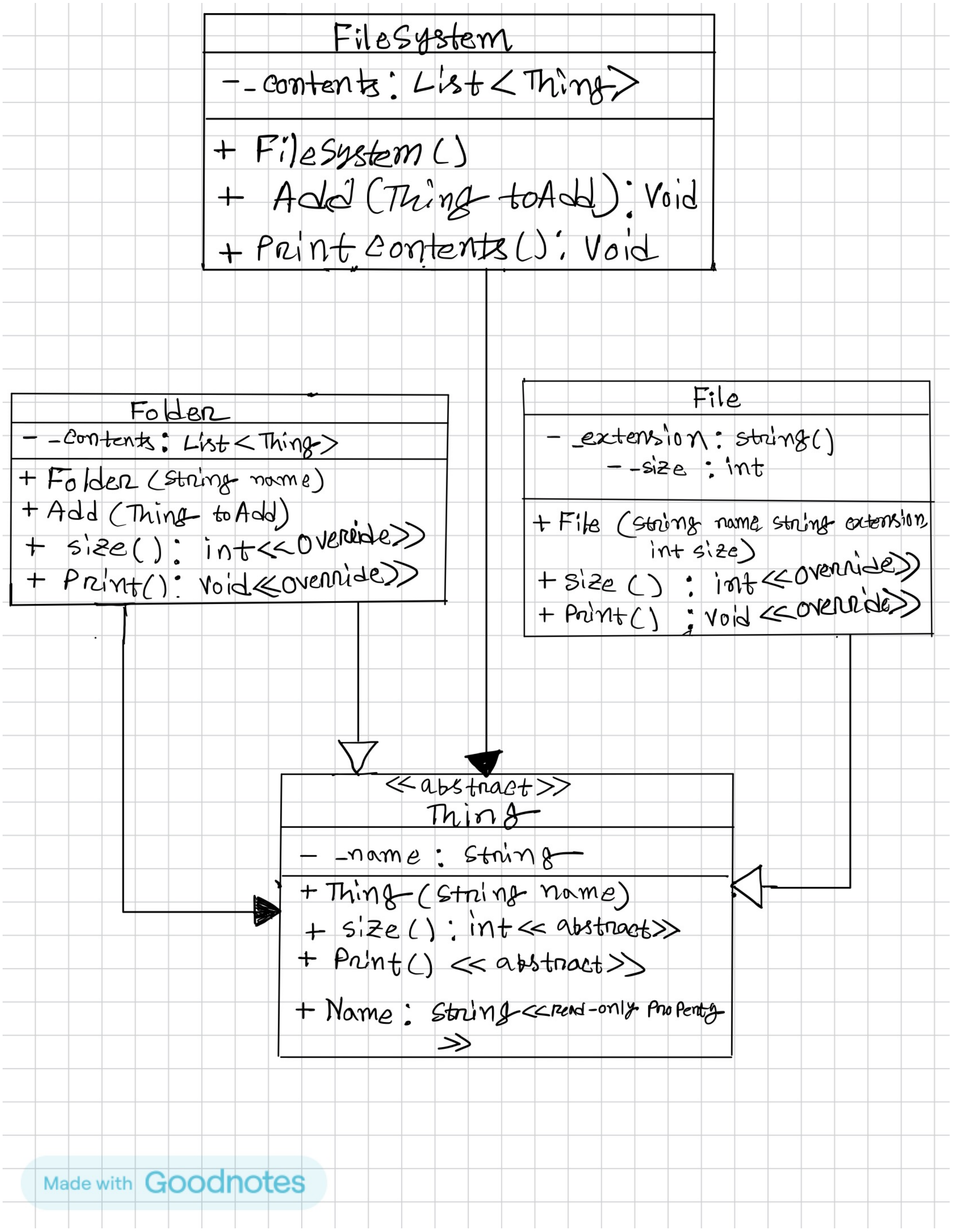
Swinburne University of Technology

COS20007 Object Oriented Programming

# Semester test

PDF generated at 12:59 on Friday 20th October, 2023


FileSystem
- _contents : List<Thing>
+ FileSystem()
+ Add(Thing toAdd) : Void
+ PrintContents() : Void
Folder
- _Contents : List<Thing>
+ Folder(String name)
+ Add(Thing toAdd)
+ size() : int<<override>>
+ Print() : Void<<override>>
File
- _extension : string()
- _size : int
+ File(String name string extension int size)
+ size() : int<<override>>
+ Print() : Void<<override>>
<<abstract>>
Thing
- _name : String
+ Thing(String name)
+ size() : int<<abstract>>
+ Print() <<abstract>>
+ Name : String<<Read-only Property>>
Made with Goodnotes

```
namespace FileSystem
{
    public class Program
    {
        public static void Main()
        {
            FileSystem fs = new FileSystem();

            fs.Add(new File("Dragan", "txt", 100));

            Folder folder1 = new Folder("Documents");
            folder1.Add(new File("Blue_Bloods", "txt", 200));
            fs.Add(folder1);

            Folder folder2 = new Folder("Documents2");
            Folder subFolder = new Folder("Images");
            subFolder.Add(new File("Regan", "jpg", 580));
            Folder subFolder2 = new Folder("Images2");
            subFolder2.Add(new File("Lincoln", "jpg", 600));
            folder2.Add(subFolder);
            folder2.Add(subFolder2);
            fs.Add(folder2);

            fs.Add(new Folder("NewFolder"));

            fs.PrintContents();
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Linq;

namespace FileSystem
{
    class FileSystem
    {
        private List<Thing> _contents;

        public FileSystem()
        {
            _contents = new List<Thing>();
        }

        public void Add(Thing toAdd)
        {
            _contents.Add(toAdd);
        }

        public void PrintContents()
        {
            Console.WriteLine($"The file system contains:");
            foreach (Thing thing in _contents)
            {
                thing.Print();
            }
        }
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FileSystem
{
    abstract class Thing
    {
        private string _name;

        public Thing(string name)
        {
            _name = name;
        }

        public string Name
        {
            get
            {
                return _name;
            }
        }

        public abstract int Size();
        public abstract void Print();
    }
}
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FileSystem
{
    class Folder : Thing
    {
        private List<Thing> _contents;

        public Folder(string name) : base(name)
        {
            _contents = new List<Thing>();
        }

        public void Add(Thing toAdd)
        {
            _contents.Add(toAdd);
        }

        public override int Size()
        {
            int totalSize = 0;
            foreach (Thing thing in _contents)
            {
                totalSize += thing.Size();
            }
            return totalSize;
        }

        public override void Print()
        {
            if (Size() == 0)
            {
                Console.WriteLine($"The folder {Name} is empty");
            }
            else
            {
                Console.WriteLine($"The folder {Name} contains {Size()} bytes
    ↪   total:");
                foreach (Thing thing in _contents)
                {
                    thing.Print();
                }
            }
        }
    }
}
```

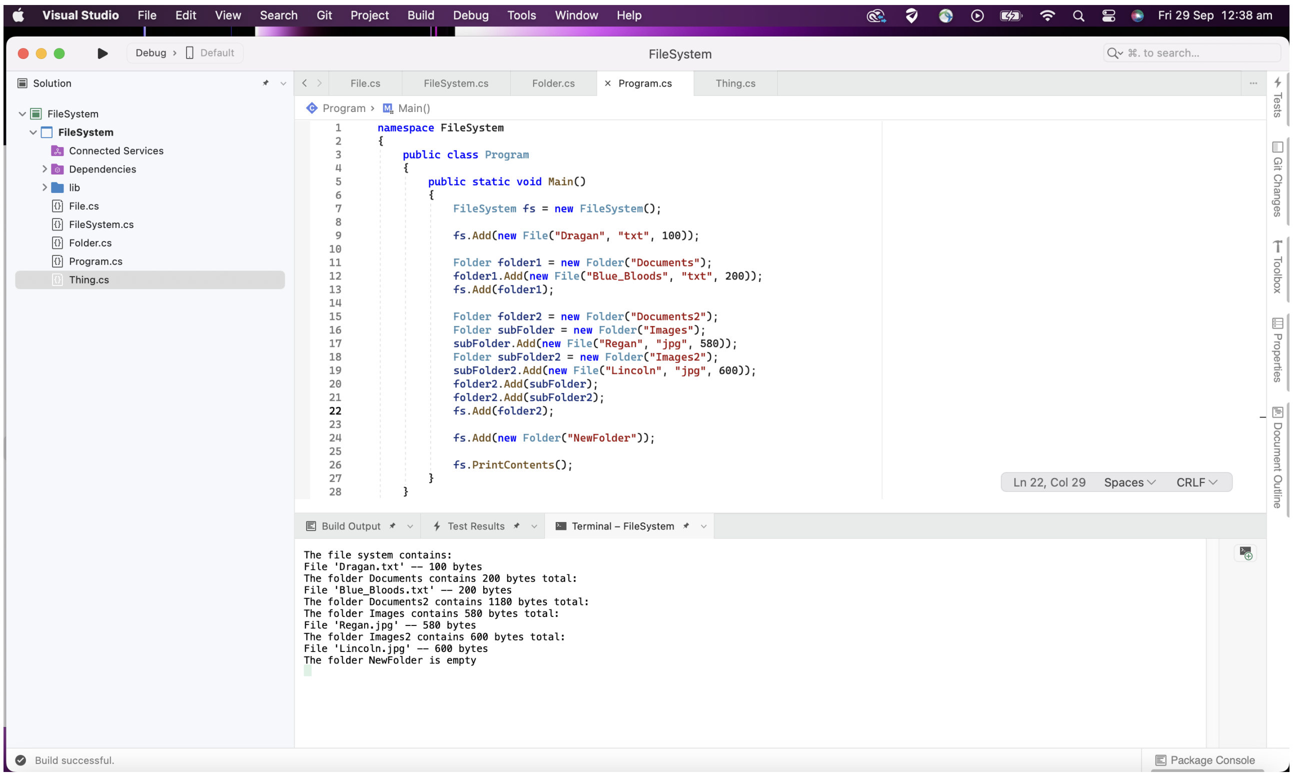
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace FileSystem
{
    class File : Thing
    {
        private string _extension;
        private int _size;

        public File(string name, string extension, int size) : base(name)
        {
            _extension = extension;
            _size = size;
        }

        public override int Size()
        {
            return _size;
        }

        public override void Print()
        {
            Console.WriteLine($"File '{Name}.{_extension}' -- {_size} bytes");
        }
    }
}
```



```csharp
namespace FileSystem
{
    public class Program
    {
        public static void Main()
        {
            FileSystem fs = new FileSystem();

            fs.Add(new File("Dragan", "txt", 100));

            Folder folder1 = new Folder("Documents");
            folder1.Add(new File("Blue_Bloods", "txt", 200));
            fs.Add(folder1);

            Folder folder2 = new Folder("Documents2");
            Folder subFolder = new Folder("Images");
            subFolder.Add(new File("Regan", "jpg", 580));
            Folder subFolder2 = new Folder("Images2");
            subFolder2.Add(new File("Lincoln", "jpg", 600));
            folder2.Add(subFolder);
            folder2.Add(subFolder2);
            fs.Add(folder2);

            fs.Add(new Folder("NewFolder"));

            fs.PrintContents();
        }
    }
```

```
The file system contains:
File 'Dragan.txt' -- 100 bytes
The folder Documents contains 200 bytes total:
File 'Blue_Bloods.txt' -- 200 bytes
The folder Documents2 contains 1180 bytes total:
The folder Images contains 580 bytes total:
File 'Regan.jpg' -- 580 bytes
The folder Images2 contains 600 bytes total:
File 'Lincoln.jpg' -- 600 bytes
The folder NewFolder is empty
```

1. Describe the principle of polymorphism and how it was used in Task 1.

   Polymorphism, along with encapsulation, inheritance, and abstraction, is one of the four essential principles of object-oriented programming (OOP). Polymorphism in the context of OOP allows objects of distinct classes to be viewed as objects of a common super class. The most common application is when a child class and its parent class share a method signature. This means that a child class method can override or extend a method in its parent class.

   The benefits of using Polymorphism:

   - **Flexibility and extensibility:** It enable us to add new classes without modifying old code.
   - **Reusability:** Common behaviour can be factored out and placed in a super class, and specific behaviours can be implemented in subclasses.
   - **Interoperability:** Objects from different classes can be considered as objects from the same class.

In task 1, the Thing.cs class is an abstract class which is one of the most common approaches in OOP to implement polymorphism. The thing class has a private member '_name' and a public property 'Name' to get values. There is a constructor that takes a 'name' parameter. 'Size()' and 'Print()' are two abstract methods. These methods used in thing has no direct implement in Thing.cs but they are expected to be implemented by the subclasses of 'Thing'.

"File' and 'Folder' classes are such subclasses in the task. The File class has attributes for extension and size and methods to return its size and print details. The Folder class contains a list of Thing objects, indicating it can store files or other folders. It calculates its size by adding up the sizes of its contents and prints details by iterating through its content list. Due to polymorphism, despite differences in File and Folder, both can be treated uniformly as "things," simplifying operations without needing type checks.

2. Consider the FileSystem and Folder classes from the updated design in Task 1. Do we need both classes? Explain why or why not.

   Both the FileSystem and Folder classes act as containers for Thing objects. While they do share similarities in storing and adding Thing objects, there are clear distinctions:

   Hierarchy and Purpose: The FileSystem class represents the topmost level of the system, essentially the root. It is a broad representation of the entire system. On the other hand, Folder represents an inner directory or container within this system, indicating a more granulated level of organization.

   Inheritance: The Folder class inherits from Thing, which implies that a Folder can be treated as a Thing and can be nested within other folders or even within the filesystem. This provides flexibility in representing nested structures.

Functionality: The Folder class offers more advanced methods, such as computing the size of its contents, whereas the Filesystem class primarily provides basic operations to interact with its contents.

Given these distinctions, it becomes evident that while both classes serve as containers, they cater to different aspects and levels of the system's organization. Removing one might lead to loss of this structural clarity. However, if there's a desire to further abstract the design, it's conceivable to consider merging the functionalities, but careful consideration should be given to ensure clarity and functionality are not compromised.

3. What is wrong with the class name Thing? Suggest a better name for the class and explain the reasoning behind your answer.

In a file system project, the class name ‘Thing’ is ambiguous and doesn’t adequately describe its role. Effective class names according to programming convention should be descriptive, intuitive, fit in the context, clarity in context, and be more readable. A more appropriate name might be "FileSystemEntity/ FSItem/ FSObject/ FSComponent" or "Entity" because it indicates a component within a file system and is clearer than "Thing", making the code more understandable for future developers.

4. Define the principle of abstraction and explain how you would use it to design a class to represent a Book.

Abstraction in OOP represents the concept of hiding intricate implementation details and showcasing only the fundamental features of an object. It anchors itself in the tenets of OOA, OOD, and OOP:

OOA (Object-Oriented Analysis): This involves understanding and defining the problem domain. In the context of our Book, this might mean determining what makes a book unique and how users interact with it.

OOD (Object-Oriented Design): Here, the identified objects are given structure and relationships. For our Book, this could involve determining its core attributes and potential methods.

OOP (Object-Oriented Programming): This is where the actual implementation occurs, keeping the principles of abstraction in mind.
Designing a ‘Book’ using Abstraction:

For our Book class, we're focusing on the essential attributes and methods that define a book in the broadest sense.

Attributes:

Title
Author

ISBN
Publisher
Date of Publication
Genre

Methods:

Read(): Simulates the act of reading the book. The precise mechanism of how the content is presented or retrieved remains abstracted.
Bookmark(): Instead of specifying a page, this method could be generalized to mark a position in the book, keeping it relevant even for audio books.
SkipTo(): Allows navigation to a specific part or chapter, again keeping the medium in mind.
Summary(): Provides an overview or synopsis of the book. Its implementation could be varied – for instance, it might be a brief textual description, a "word cloud" representation, or even an algorithmic summary.

At this stage, we're abstracting away the internals, such as the specific data structures used, searching algorithms, and other intricate details. The focus is on providing a high-level view of a Book that's versatile across different mediums and uses.