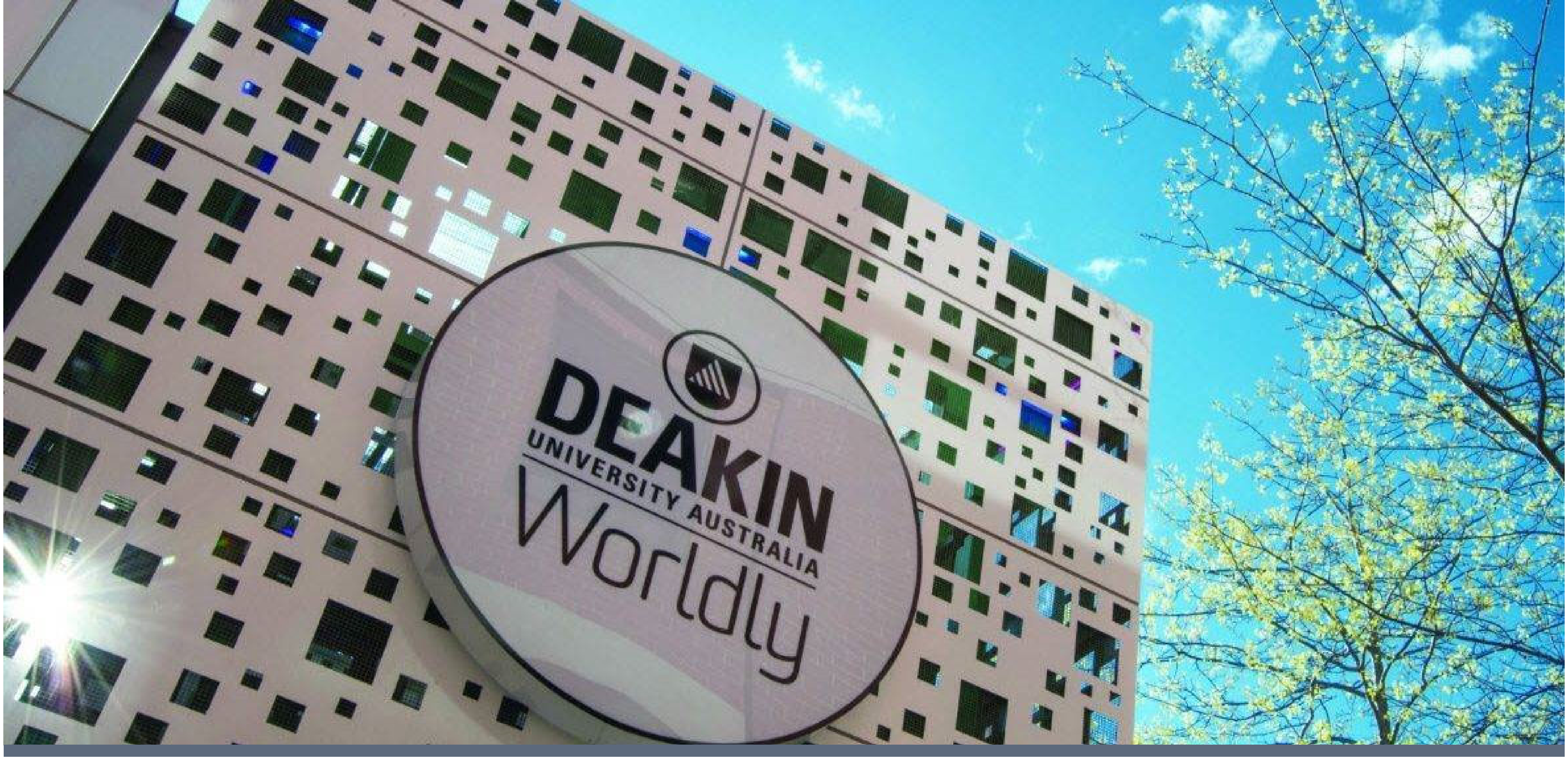


Lecture 07. Interfaces. Responsibility driven design.

SIT232 Object-Oriented Development

Interfaces: The idea

- An object has a state and behaviour, where:
  - state - all the attributes (and their values) of that object
  - behaviour - all the methods (working on the state) of that object
- Objects with common behaviour and meaning of state are grouped into inheritance hierarchies
- But what is with objects that have only common functionality and no common meaning of state?

Interfaces: The idea

- An interface is an OOP construct that is used to capture the idea of only common functionality.
- Also seen as equivalent to an abstract class that only contains abstract methods.
- Interfaces allow for easily maintainable code because they facilitate loose coupling.
- Classes should implement an interface when we say that they **act as** the interface.

Interface: Syntax

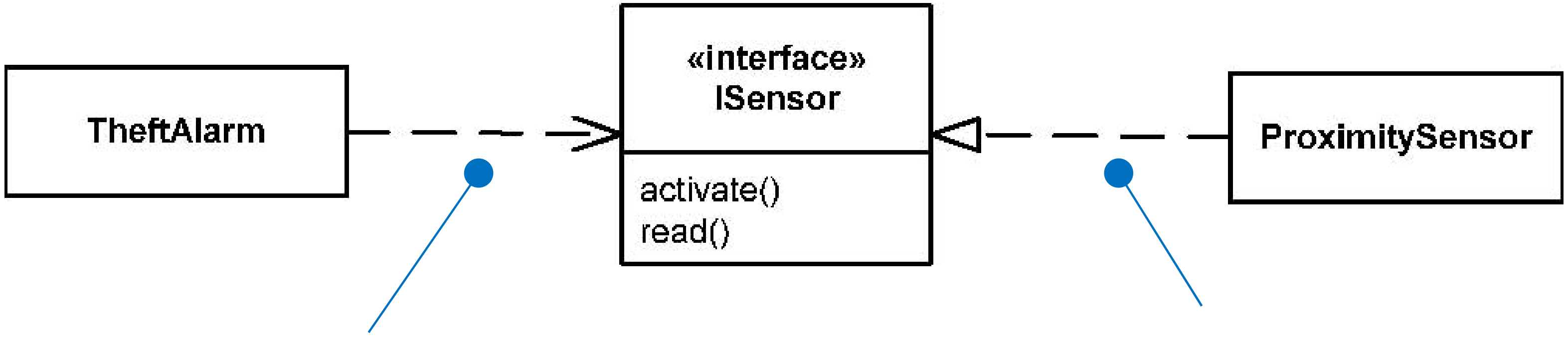
```
// Declaring interface:
[access_modifier] interface interface_name
{
    data_type name { [get;] [set;] }                    // declares property
    return_type method_name([parameter[, ...]]);        // declares method
    ...
}

// Implementing interface
[access_modifier] class derived_class_name : interface_name
{
    ...
}
```

Any class can implement any number of interfaces in addition to inheritance, e.g.,

```
class DerivedClass : BaseClass, ISomeInterface, IDisposable
```

Interfaces in UML



Depends (uses) the interface

Implements the interface

When to use which?

- If you anticipate creating multiple versions of an entity, use ...
  - abstract class
- If the functionality you are creating is useful across a wide range of disparate objects, use ...
  - interfaces
- If you are designing small, concise bits of functionality, use ...
  - interfaces
- If you want to provide common implemented functionality across all the implementations of your entity, use ...
  - abstract class

Further Thoughts

- The objects to be accessed polymorphically must either be related via inheritance or implement a common interface.
- Functions defined as virtual in a class must be applicable to all derived classes.
- Functions with no meaningful implementation at the base class level must be defined as abstract and implemented in a derived class.
- Base class should capture attributes and functions common to all derived classes.

OOP: Why do we need it?

- Strong design methodology (Data abstraction)
- Supports the distributed design of components (Modularity)
- OOP is a good approach for problems that need OOP.
  - Using a well-known approach for solving problems.
  - Learn from the past.
  - Improve speed, efficiency and reliability.

Object-Oriented Design

- Classes:
  - What does a user see? (External view)
  - How does it work? (Internal view)
- Thinking about the outside view
  - Abstraction
  - What are the essential properties that things like this share?
- Once we think about what we are going to present, then we can think about implementation.

Class Design

- Classes do:
  - contain a variety of variables and methods
- Classes don’t:
  - let everyone mess with variables as they want to.
- Exposing the instance variables means that people may start using them and then we can’t change this in the future.
- Users changing the variables directly may make changes that the class would normally block.

Object-Oriented Design: Inheritance

- Is Inheritance fundamental to OO?
- Why do we inherit from another class?
  - To change (at least) one of the behaviours of the inherited class.
- Too often, people use **inheritance** when they could get away with **composition**.
  - Create new behaviour by embedding objects inside a new class.

When would Containment be a better idea than Inheritance?

(Click here)

Object-Oriented Design: Inheritance

```
class Fruit {
   // Return int number of pieces of peel that
   // resulted from the peeling activity.

   public int Peel() {
      Console.WriteLine("Peeling is appealing.");
      return 1;
   }
}

class Apple : Fruit { }

class Example1 {

   public static void Main() {
      Apple apple = new Apple();
      int pieces = apple.Peel();
   }
}
```

Does inheritance imply
a bottleneck?

Object-Oriented Design: Containment

```
class Fruit {
   // Return int number of pieces of peel that
   // resulted from the peeling activity.

   public int Peel() {
       Console.WriteLine("Peeling is appealing.");
       return 1;
   }
}

class Apple {
   private Fruit fruit = new Fruit();
   public int Peel() { return fruit.Peel(); }
}

class Example1 {

   public static void Main() {
       Apple apple = new Apple();
       int pieces = apple.Peel();
   }
}
```

Does Containment give a more flexible solution?

Inheritance versus Containment

Containment

- Simpler structure
- No ‘surprise’ encapsulation breaking.
- Can be dynamic run-time mechanism. (This is good and bad.)
- You define exactly what this class does.

Inheritance

- Complicated hierarchies
- New method in the parent can add (unwanted) behavior in the child.
- Static (compile-time) mechanism.
- The child is influenced by the parent (unless you totally ignore the parent – if so, why inherit?)

Small group discussion!

Should you use inheritance?

- There are many ways to work out whether you should be using inheritance or not.

  For example, if you nullify or override a lot of behavior from the parent, why are you inheriting from it?

- Liskov's substitution principle:

  "Functions that use references to base-class objects must be able to use sub-class objects without knowing it."

Liskov's Substitution Principle

- We must write our subclasses so that they don't violate "is-a-kind-of" class hierarchies.
- We can't restrict the behaviour of the parent.
- We have to be comfortable with the subclass sharing both behaviour and structure with the parent, not just structure.

Liskov's Substitution Principle

LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Things to watch in class design

- Overriding the parent class:
  - Be careful with this. The more you do it, the more likely it is that you shouldn’t be inheriting.
  - Do you call the superclass method you override? GIANT WARNING SIGNS.
- Do you have to keep using **Reflection** (‘is’ operator in C#) to work out which behaviours are available?
- Make sure you’re only using Reflection when you have to.
  - It’s a useful tool but overuse often means poor design.
  - Reflection gives you the ability to inspect classes and dynamically call things at run-time, so you’re modifying your program at run-time based on how things currently look.
  - Powerful but easy to overuse and get wrong.

A Bigger World

- OOP is not enough by itself.
- Big programs have too many objects and classes for one person to maintain a good grasp of their relationships.
- Relationships can be very complex and you may not ever see all of the places where an object is used.
- Eventually, complexity will overwhelm you.

OOP Design Patterns

- Design patterns recognize that we tend to solve certain problems the same way:
  - There are commonly-occurring relationships between classes.
  - If we can work out which pattern to use, we will
    - → Reduce the burden of complexity, and
    - → Be more likely to succeed.
- Think of a design pattern as a general reusable solution to a commonly occurring problem within a given context in software design.
- Not every pattern fits everywhere but where it does it should help.