SOFTWARE MAINTENANCE & EVOLUTION

LINGI2252 – PROF. KIM MENS
SOFTWARE REUSE
& OBJECT-ORIENTED PROGRAMMING

LINGI2252 – PROF. KIM MENS
A. SOFTWARE REUSE
REUSABILITY [DEFINITION]

Reusability is a general engineering principle whose importance derives from the desire to avoid duplication and to capture commonality in undertaking classes of inherently similar similar tasks.


Software reusability is the degree to which a software module or other work product can be used in more than one software system.

Reusable: pertaining to a software module or other work product that can be used in more than one computer program or software system.
SOFTWARE REUSE [DEFINITION]

Software reuse

The reapplication of a variety of kinds of knowledge about one system to another in order to reduce the effort of developing or maintaining that other system.

This “reused knowledge” includes artefacts such as domain knowledge, development experience, requirements, architectural components, design artefacts, code, documentation, and so forth.

SOFTWARE REUSE

REUSABLE COMPONENT [DEFINITION]

Software reuse

The process of implementing new software systems using existing software information.

Reusable component

A software component designed and implemented for the specific purpose of being reused.

Component can be requirement, architecture, design, code, test data, etc.

SOFTWARE REUSE [EXAMPLE]

Using **functions** available in some library.

E.g., C libraries are collections of precompiled functions that have been written to be reused by other programmers.

Reusing **classes** from another object-oriented program.

Adapting the **modules** of a software system with a very similar functionality (member of a same “family”).

Reusing the **architecture** or **design** of a software system when porting it to a new language.
WHY REUSE?

Economic justification:

- more productive by avoiding double work
- better quality by reusing good solutions

Intellectual justification:

- stand on each other's shoulders
- don't reinvent or reimplement old stuff
- focus on what's new and relevant
SOME REUSE TECHNIQUES

Programming abstractions and mechanisms

- procedural and data abstraction
- encapsulation and information hiding
- code sharing and reuse mechanisms

Design patterns

Software architecture

Software libraries & application frameworks

Generative programming & model-driven development
B. OBJECT-ORIENTED PROGRAMMING
It is often “claimed” that object-oriented programming is a better way of writing more modular programs. It leverages code sharing and design reuse. It minimizes maintenance costs. Thanks to its abstraction mechanisms.
ABSTRACTION MECHANISMS

Encapsulation

keep data and operations that act on this data together

Information hiding

isolate and hide design and implementation choices

Polymorphism

allow for different implementations of a same design to co-exist

Code sharing

capture and exploit similarities in data and behaviour (through inheritance)
KEY OBJECT-ORIENTED CONCEPTS

Objects & Classes

Methods & Messages

Polymorphism & Dynamic Binding

Hierarchies of classes

Method overriding, self & super calls

Abstract classes & methods

Different kinds of inheritance
  - Single, Multiple, Interfaces, Mixins

DISCLAIMER

ALTHOUGH WHAT FOLLOWS MAY SEEM LIKE A CRASH COURSE IN OO

OUR FOCUS WILL LIE ON THE MECHANISMS IT PROVIDES FOR ACHIEVING MODULARITY, MAINTAINABILITY, SHARING AND REUSE
TWO MAIN PRINCIPLES OF OBJECT-ORIENTED PROGRAMMING

Everything is an object

Objects respond only to messages
SMALLTALK'S INFLUENCE

Smalltalk is a pure object-oriented language

Was a source of inspiration to many OO languages

- **Ruby** is heavily inspired on Smalltalk
- **Objective-C** and **Swift** heavily inspired on Smalltalk
- **Java** is heavily influenced by Smalltalk

**DISCLAIMER**
THIS SESSION MAY CONTAIN TRACES OF SMALLTALK CODE (FOR DIDACTIC PURPOSES)
KEY OBJECT-ORIENTED CONCEPTS

Objects & Classes

Methods & Messages

Polymorphism & Dynamic Binding

Hierarchies of classes

Method overriding, self & super calls

Abstract classes & methods

Different kinds of inheritance
  
  Single, Multiple, Interfaces, Mixins
**OBJECTS ENCAPSULATE DATA**

Every **object** has its own **data** or state

Values stored in the objects

(But variables declared in classes)

Data is encapsulated

Protected from the outside world

Only accessible through **messages**

<table>
<thead>
<tr>
<th>aPoint</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>y</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>aCircle</th>
</tr>
</thead>
<tbody>
<tr>
<td>center</td>
</tr>
<tr>
<td>radius</td>
</tr>
<tr>
<td>circumference</td>
</tr>
</tbody>
</table>
Classes encapsulate behaviour

Classes

declare the state of objects
(but objects contain the actual values)

define the behaviour of objects

method implementations

shared among all objects of a class

can manipulate the state directly

Behaviour is encapsulated

invoked by sending message to an object
CLASSES ARE FACTORIES OF OBJECTS

A class is a “factory” for producing objects of the same type
e.g., with a Circle class you can create many circle objects

Every object is an instance of the class from which it was created

A class is a blueprint for objects that share behaviour and state

All objects of a class behave in a similar fashion in response
to a same message

Enables reuse of behaviour
CLASSES & OBJECTS PROMOTE MODULARITY

Through encapsulation of both behaviour and state grouping behaviour with the data it acts upon facilitates modularity, code reuse and maintenance.
CLASSES & OBJECTS PROMOTE REUSE

Classes are fine-grained reusable components that enable sharing and reuse of structure and behaviour even across applications for example via application frameworks or reusable class hierarchies
KEY OBJECT-ORIENTED CONCEPTS

- Objects & Classes
- Methods & Messages
- Polymorphism & Dynamic Binding
- Hierarchies of classes
- Method overriding, self & super calls
- Abstract classes & methods
- Different kinds of inheritance
  - Single, Multiple, Interfaces, Mixins
METHODS & MESSAGES (RECAP)

Objects (not functions or procedures) are the main building blocks of OO

Objects communicate through message passing

Objects exhibit behaviour in response to messages sent to them

The actual behaviour is implemented in methods

Methods specify what behaviour to perform on objects

Methods can manipulate the objects’ internal state
POLYMORPHIC METHODS

A same message can be sent to objects of different classes

\[
\text{aCircle.surface} \quad \text{aRectangle.surface}
\]

Different objects can react differently to the same message
different classes can provide different implementations
for methods with the same name

\[
\begin{align*}
\text{Circle} & \ > \ \text{surface} = \pi \cdot \text{radius}^2 \\
\text{Rectangle} & \ > \ \text{surface} = (\text{bottom-top}) \cdot (\text{right-left})
\end{align*}
\]

Responsibility of how to handle the message is decided by the object (depending on the class to which it belongs)

This is called "polymorphism"
ADVANTAGES OF POLYMORPHISM

Cleaner, more maintainable code

Less ≠ method names

Less need for conditionals

More implementation freedom

Each class can decide how best to implement a method

Locality

Every object/class is responsible for its own actions

Easy to change the implementation by another one
EXAMPLE OF POLYMORPHISM

Procedural style vs. object-oriented style

Example:

Write some code that calculates the sum of the surfaces of a collection of different shape objects

\[
\text{surface}\text{(collection)} = \sum_{\text{shape} \in \text{collection}} \text{surface}\text{(shape)}
\]
EXAMPLE OF POLYMORPHISM (PSEUDOCODE)

Procedural style (no polymorphism)

circleSurface(c) = \pi.r \cdot \text{radius}(c)^2

rectangleSurface(r) = (\text{bottom}(r) - \text{top}(r)) \cdot (\text{right}(r) - \text{left}(r))

surface(collection) : Real
    total = 0
    \forall \text{shape} \in \text{collection}:
        if ( \text{shape} == \text{Circle} ) then
            total = total + circleSurface(\text{shape})
        else if ( \text{shape} == \text{Rectangle} ) then
            total = total + rectangleSurface(\text{shape})
    return total
EXAMPLE OF POLYMORPHISM (PSEUDOCODE)

**OO style (using polymorphism)**

Circle {
  Point center ; Real radius ;
  Real surface() : { \( \pi \cdot \text{radius}^2 \) }
}

Rectangle {
  Real bottom, top, right, left;
  Real surface() : { (bottom-top)*(right-left) }
}

Real surface(collection) : {
  total = 0
  \( \forall \quad \text{shape} \in \text{collection} : \text{total} = \text{total} + \text{shape.surface}() \)
  return total
}
EXAMPLE OF POLYMORPHISM (PSEUDOCODE)

OO style (using polymorphism)

Real surface(collection) : {
    total = 0
    ∀ shape ∈ collection : total = total + shape.surface()
    return total
}

Advantages:

Adding a new shape does not require to change the existing implementation

No need to know the kind of objects it manipulates as long as they all share a common interface
LATE BINDING

When sending a message, the actual receiver of a message is not necessarily known until run-time.

Mapping of messages to methods is deferred until run-time, depending on which object actually received the message. We call this *late binding* or *dynamic binding*.

Most traditional languages do this at compile time (static binding).

Smalltalk uses late binding.
abstract Shape {
    abstract Real surface() : {}
}

Circle inherits Shape {
    Point center; Real radius;
    Real surface() : { π.radius² }
}

Rectangle inherits Shape {
    Real bottom, top, right, left;
    Real surface() : { (bottom-top)*(right-left) }
}

test() : {
    Shape shape;
    shape = new Circle(new Point(0,0), 2);
    print(shape.surface());
    shape = new Rectangle(10,10,30,50);
    print(shape.surface());
}
STATIC VS. DYNAMIC BINDING IN JAVA

Smalltalk uses dynamic binding (a.k.a. late binding)

For Java it depends

Binding of overridden methods happens at runtime (dynamic)

Binding for overloaded methods at compile time (static)

Binding of private, static and final methods at compile time (static) since these methods cannot be overridden.

Sources:

http://beginnersbook.com/2013/04/java-static-dynamic-binding/

Apart from syntactic differences and the lack of type declarations in Smalltalk, this example could be recreated nearly “as is” in Smalltalk.

In fact for Smalltalk you could even create an example that doesn’t require inheritance.

Example of Dynamic Binding in Java

```java
public class DynamicBindingTest {
    public static void main(String args[]) {
        Vehicle vehicle = new Car(); // here Type is vehicle but object will be Car
        vehicle.start(); // Car's start called because start() is overridden met
    }
}

class Vehicle {
    public void start() {
        System.out.println("Inside start method of Vehicle");
    }
}
class Car extends Vehicle {
    @Override
    public void start() {
        System.out.println("Inside start method of Car");
    }
}

Output: Inside start method of Car
```

LATE BINDING EXAMPLE IN JAVA

The method call `vehicle.start()` is dynamically bound to the overridden `Car > start()` method.

Because even though `vehicle` is typed as being of class `Vehicle`, it is determined at runtime that it contains an object of type `Car`, and because the method `start()` is overridden.

```java
public class DynamicBindingTest {
    public static void main(String args[])
    {
        Vehicle vehicle = new Car(); // here Type is vehicle but
        vehicle.start();               // Car's start called because start
    }
}

class Vehicle
{
    public void start()
    {
        System.out.println("Inside start method of Vehicle");
    }
}

class Car extends Vehicle
{
    @Override
    public void start()
    {
        System.out.println("Inside start method of Car");
    }
}
```

Output: Inside start method of Car
STATIC BINDING EXAMPLE IN JAVA

Static Binding Example in Java

```java
public class StaticBindingTest
{
    public static void main(String args[])
    {
        Collection c = new HashSet();
        StaticBindingTest et = new StaticBindingTest();
        et.sort(c);
    }
    //overloaded method takes Collection argument
    public Collection sort(Collection c)
    {
        System.out.println("Inside Collection sort method");
        return c;
    }
    //another overloaded method which takes HashSet argument which is sub class
    public Collection sort(HashSet hs)
    {
        System.out.println("Inside HashSet sort method");
        return hs;
    }
}
```

Output: Inside Collection sort method

STATIC BINDING EXAMPLE IN JAVA

The method call `et.sort(c)` is statically determined by the compiler to refer to the `sort(Collection)` method.

Even though `c` is an object of type `HashSet` and the `sort(HashSet)` method is more specific.

Because `c` is statically determined to have type `Collection` and the method `sort` is overloaded, not overridden.

```
public class StaticBindingTest {
    public static void main(String args[]) {
        Collection c = new HashSet();
        StaticBindingTest et = new StaticBindingTest();
        et.sort(c);
    }
    //overloaded method takes Collection argument
    public Collection sort(Collection c) {
        System.out.println("Inside Collection sort method");
        return c;
    }
    //another overloaded method which takes HashSet argument
    public Collection sort(HashSet hs) {
        System.out.println("Inside HashSet sort method");
        return hs;
    }
}
```

Output: Inside Collection sort method

This example cannot be recreated in Smalltalk since Smalltalk has no method overloading. (Nor does it have final methods or private methods.)
CLASSES & OBJECTS PROMOTE MODULARITY

Through information hiding

restricted access to objects through a well-defined interface

users of an object only know the set of messages it will accept

they do not know how the actions performed in response to a message are carried out

This is the responsibility of the receiving object (through polymorphism)

improves modularity by hiding implementation details

How the data is represented internally

How the behaviour is implemented in terms of that data
KEY OBJECT-ORIENTED CONCEPTS

- Objects & Classes
- Methods & Messages
- Polymorphism & Dynamic Binding
- Hierarchies of classes
- Method overriding, self & super calls
- Abstract classes & methods
- Different kinds of inheritance
  - Single, Multiple, Interfaces, Mixins
HIERARCHIES OF CLASSES

Shape

- colour
- (Colour)

Rectangle
- bottom
- top
- left
- right
- surface: height\times width
- circumference: 2\cdot height + 2\cdot width

Circle
- center
- radius\ n
- surface: \pi \cdot radius^2
- circumference: 2\cdot \pi \cdot radius
HIERARCHIES OF CLASSES

Classes are typically organised into hierarchical structures.

Information (data/behaviour) associated with classes higher in the hierarchy is automatically accessible to classes lower in the hierarchy.

Each subclass specialises the definition of its ancestors:

- Subclasses can use ancestor’s behaviour and state.
- Subclasses can add new state and behaviour.
- Subclasses can specialise ancestor’s behaviour.
- Subclasses can override ancestor’s behaviour.
HIERARCHIES OF CLASSES

Inheritance is a powerful incremental reuse mechanism

Often you don’t want to rewrite everything; you just want some small changes to what exists

Classes are the units of reuse

Inheritance is the reuse mechanism

  e.g., *extends* keyword in Java:
  class Automobile extends LandVehicle

Class hierarchies are ideal for sharing declarations and implementation among classes

  Object’s state and behavioural description is broken into pieces and distributed along specialisation paths

Promotes *encapsulation, modularity, and reusability*
KEY OBJECT-ORIENTED CONCEPTS

- Objects & Classes
- Methods & Messages
- Polymorphism & Dynamic Binding
- Hierarchies of classes
- Method overriding, self & super calls
- Abstract classes & methods
- Different kinds of inheritance
  - Single, Multiple, Interfaces, Mixins
SELF AND SUPER CALLS

Methods use:

- **self** calls to reference the receiver object
  - *this* keyword in Java, **self** in Smalltalk
- **super** to reference their implementor’s parent

Attention! Key issue in object-oriented programming:

- **self** = late/dynamically bound
  - method lookup starts again in the class of the receiver object
- **super** = statically bound
  - method lookup starts in the superclass of the class of the method containing the super expression;
  - *not* in the superclass of the receiver class
SELF REFERS TO THE RECEIVER CLASS

SomeSuperclass {
    void printMyself : {
        self.print
    }
    void print : {
        display("Printed in superclass. ")
    }
}

SomeSubclass inherits from SomeSuperclass {
    void print : {
        super.print
        display("Printed in subclass.")
    }
}

SubSubclass inherits from SomeSubclass {
}

test : {
    s = new SubSubclass()
    s.printMyself
}

self refers to the receiver object
receiver class is SubSubclass
self will dynamically look up methods starting from this class
**METHOD OVERRIDING**

Subclasses can re-implement methods that are already implemented in superclasses

*enables fine-grained reuse*

clients do not have to know this (encapsulation and polymorphism)

An overridden method

can either **overwrite** a method with a completely new implementation

or can **specialise** the behaviour of the method defined in its superclass

special keyword for accessing the superclass : super
EXAMPLE OF METHOD SPECIALISATION

SomeSuperclass {
    void print : {
        display("Printed in superclass. ")
    }
}

SomeSubclass inherits from SomeSuperclass {
    void print : {
        super.print
        display("Printed in subclass.")
    }
}

test : {
    s = new SomeSubclass()
    s.print
}

After calling test, the program prints:
Printed in superclass. Printed in subclass.

overridden method

overriding method

specialises overridden method using super keyword
SUPER IS **NOT** THE SUPERCLASS OF THE RECEIVER CLASS

```java
SomeSuperclass {  
    void print : {  
        display("Printed in superclass. ")  
    }  
}

SomeSubclass inherits from SomeSuperclass {  
    void print : {  
        super.print  
        display("Printed in subclass.")  
    }  
}

SubSubclass inherits from SomeSubclass {  
}

test : {  
    s = new SubSubclass()  
    s.print  
}
```

After calling `test`, the program prints:
Printed in superclass. Printed in subclass.

*super* statically refers to this class

receiver class is `SubSubclass`

if *super* would refer to the super class of the receiver class, we would get a loop
A a = new A();

B b = new B();

C c = new C();

A
- x = 1
- s = “Aha”
- n() { print(s) }
- m() { this.n(); print(x) }

B
- y = 2
- m() { print(s+y) }
- s() { super.m() }

C
- x = 3
- s = “Choco”
- n() { super.n(); print(“tof”) }

a.n();
a.m();
a.s();
b.n();
b.m();
b.s();
c.m();
c.n();
c.s();

Predict the result of these method invocations.
KEY OBJECT-ORIENTED CONCEPTS

- Objects & Classes
- Methods & Messages
- Polymorphism & Dynamic Binding
- Hierarchies of classes
- Method overriding, self & super calls
- Abstract classes & methods
- Different kinds of inheritance
  - Single, Multiple, Interfaces, Mixins
CONCRETE VS. ABSTRACT CLASSES

Abstract Class

Holds on to common characteristics shared by other classes

Not expected to have instances

Concrete Class

Contains complete characterisation of actual objects of that class

Expected to have instances
ABSTRACT CLASSES AND ABSTRACT METHODS

cannot be instantiated (in Java)

but can provide some method implementations

methods of which the implementation is shared by all subclasses

methods with a default implementation to be specialised by subclasses

methods with a partial implementation to be completed by a subclass (e.g., template method pattern)

typically have at least one abstract method

a method with an empty implementation that must be provided by each subclass
KEY OBJECT-ORIENTED CONCEPTS

- Objects & Classes
- Methods & Messages
- Polymorphism & Dynamic Binding
- Hierarchies of classes
- Method overriding, self & super calls
- Abstract classes & methods
- Different kinds of inheritance
  - Single, Multiple, Interfaces, Mixins
KINDS OF INHERITANCE

Different kinds of inheritance

Single: 1 superclass

Multiple: 1 or more superclasses

Interface

Mixin modules
SINGLE INHERITANCE

Organises classes in tree structures

Every class has a unique superclass

There is a root class, typically called Object
SINGLE INHERITANCE PROBLEMS

Classes can play several roles

- Factories from which instances can be created
- Units of reuse

Inheritance can play several roles

- Code reuse
- Design reuse

These roles can conflict

Multiple inheritance to the rescue...?
MULTIPLE INHERITANCE

Sometimes it is convenient to have a class which has multiple parents.

Some languages, like C++, support multiple inheritance.

Subclasses inherit instance variables and methods from all parents.

https://en.wikipedia.org/wiki/Multiple_inheritance
THE DIAMOND PROBLEM

A problem arises when the same methods or variables are inherited via different paths

e.g. what version of **location** method to use when called on FlyingCar?

duplicated behaviour

can be solved through manual overriding

or through linearisation

duplicated state

harder to solve
INTERFACES

Java has single inheritance

Java interfaces were introduced to provide some of the functionality of true multiple inheritance

You can inherit from one class and from multiple interfaces simultaneously

Interfaces are like abstract classes with no fields or method implementations

No diamond problem since interfaces contain no data or behaviour

If Java has no multiple inheritance then how should I do something like this?

```java
class FoodTruck extends Truck, Kitchen {
}

foodTruck.drive();
foodTruck.cook(pizza);
```

class FoodTruck extends Truck implements KitchenInterface {
    Kitchen kitchen;
    public void cook(Food foodItem) {
        kitchen.cook(foodItem);
    }
}

foodTruck.drive();
foodTruck.cook(pizza);
MIXINS

Factoring out the increment when subclassing

Mixins can be seen as the “increment” that needs to be applied to a class

Mixin composition is an operation that applies a mixin to a class to produce a more specialised class

Typically, mixin composition is linearised

this can cause problems:

composition order important

introducing extra mixin can change behaviour

Example: mixin modules in Ruby
CONCLUSION

OO promotes maintainability by viewing programs as collections of loosely connected objects

Each object is responsible for specific tasks

It is through the interaction of objects that computation proceeds

Objects can be defined and manipulated in terms of the messages they understand and ignoring the implementation details

OO promotes the development of reusable components

By reducing the interdependency among individual software components

Such components can be created and tested as independent units in isolation from the rest of the software system

Reusable software components permit to treat problems at a higher level of abstraction
ABSTRACTION MECHANISMS (REVISITED)

Encapsulation

objects contain their own data as well as the methods that work on that data

Information hiding

clients of an object know only the set of messages it can receive

implementation details of how it processes these messages remain hidden to external clients

Polymorphism

cleaner and more maintainable code by delegating responsibilities and implementation choices to the objects

Code sharing

classes enable sharing behaviour among objects

class hierarchies and inheritance enable reuse of class definitions
Learning objectives:
- Definition and difference between maintenance, evolution, reuse
- Different types of maintenance
- Causes and changes of evolution
- Technical
SOFTWARE REUSE

LEARNING OBJECTIVES

▸ definitions of reusability, software reuse and reusable component

▸ how object-oriented programming promotes modularity, maintainability and reuse

▸ encapsulation, information hiding, polymorphism and code sharing

▸ key object-oriented concepts: object, classes, methods, messages, inheritance

▸ polymorphism and dynamic binding

▸ method overriding, self and super calls

▸ abstract classes and methods

▸ different kinds of inheritance: single, multiple, interfaces, mixins
SOFTWARE REUSE

POSSIBLE QUESTIONS

16. Define and illustrate the notions of software reuse, reusability and reusable components.

17. Give two economic and two intellectual justifications for software reuse. Explain in detail.

18. Give (and explain) at least 3 different software reuse techniques seen throughout the course.

19. How and why does object-oriented programming promote modularity and maintainability?

20. Explain the object-oriented techniques of encapsulation, information hiding, polymorphism and code sharing and how they relate to software reusability.
21. Explain, using a concrete example, what polymorphism and dynamic binding is, and how it can lead to more maintainable code.

22. Explain on a concrete example the concepts of method overriding, self and super calls.

23. How can abstract classes and methods improve reusability? Explain and illustrate with a concrete example.

24. Explain, using a concrete example, how a multiple inheritance problem could be modelled in terms of single inheritance on classes and interfaces in Java.