Open & Closed

A module is open if can

Add/remove fields
Add/remove methods

A module is closed if

It can be used by other modules
Open-Close Principle

Classes should be open for extension

But closed for modification
1. They are "Open For Extension". This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.

2. They are "Closed for Modification". The source code of such a module is inviolate. No one is allowed to make source code changes to it.

It would seem that these two attributes are at odds with each other. The normal way to extend the behavior of a module is to make changes to that module. A module that cannot be changed is normally thought to have a fixed behavior. How can these two opposing attributes be resolved?

Abstraction is the Key. In C++, using the principles of object oriented design, it is possible to create abstractions that are fixed and yet represent an unbounded group of possible behaviors. The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes. It is possible for a module to manipulate an abstraction. Such a module can be closed for modification since it depends upon an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

Figure 1 shows a simple design that does not conform to the open-closed principle. Both the Client and Server classes are concrete. There is no guarantee that the member functions of the Server class are virtual. The Client class uses the Server class. If we wish for a Client object to use a different server object, then the Client class must be changed to name the new server class.

Figure 2 shows the corresponding design that conforms to the open-closed principle. In this case, the AbstractServer class is an abstract class with pure-virtual member functions. The Client class uses this abstraction. However objects of the Client class will be using objects of the derivative Server class. If we want Client objects to use a different server class, then a new derivative of the AbstractServer class can be created. The Client class can remain unchanged.

Figure 1

Closed Client

Client

Server

Figure 2

Open Client

Client

Abstract Server

Server

---

Programs that conform to the open-closed principle are changed by adding new code, rather than by changing existing code.

Such programs do not experience the cascade of changes exhibited by non-conforming programs.
Open-Close Principle

Decorator & Visiter are ways to satisfy open-closed principle
Strategy Pattern
Favor Composition over Inheritance
Orderable List

Sorted
Reverse Sorted
Random
OrderableList

SortedList

ReverseList

RandomList
One size does not fit all
Issue 1 - Orthogonal Features

Order
Sorted
Reverse Sorted
Random

Threads
Synchronized
Unsynchronized

Mutability
Mutable
Non-mutable
Change behavior at runtime

OrderableList x = new OrderableList();
x.makeSorted();
x.add(foo);
x.add(bar);
x.makeRandom();
Configure objects behavior at runtime
Strategy Pattern

class OrderableList {
    private Object[] elements;
    private Algorithm orderer;

    public OrderableList(Algorithm x) {
        orderer = x;
    }

    public void add(Object element) {
        elements = ordered.add(elements, element);
    }
}
Structure

Context
contextInterface()

Strategy
algorithmInterface()

ConcreteStrategyA
algorithmInterface()

ConcreteStrategyB
algorithmInterface()
The algorithm is the operation

Context contains the data

How does this work?
Prime Directive
Data + Operations
How does Strategy Get the Data?

Pass needed data as parameters in strategy method

Give strategy object reference to context
Strategy extracts needed data from context
Example - Java Layout Manager

```java
import java.awt.*;
class  FlowExample  extends Frame  {

    public FlowExample( int  width, int height ) {
        setTitle( "Flow Example" );
        setSize( width, height );
        setLayout( new FlowLayout( FlowLayout.LEFT) );

        for ( int label = 1; label < 10; label++ )
            add( new Button( String.valueOf( label ) ) );
        show();
    }

    public static void main( String  args[] ) {
        new  FlowExample( 175, 100 );
        new  FlowExample( 175, 100 );
    }
}
```
Example - Smalltalk Sort blocks

| list |
list := #( 1 6 2 3 9 5 ) asSortedCollection.
Transcript
   print: list;
   cr.
list sortBlock: [:x :y | x > y].
Transcript
   print: list;
   cr;
   flush.
Java Sorting

How to sort a Collection in Java?

ArrayList List method - sort(Comparator<? super E> c)

Create a subclass of Comparator

Pass in comparator object to sort method

```java
List<Students> students = new ArrayList<>();
add students
students.sort(new SortByGPA());
```
Lambda & Strategy Pattern

If strategy only contains one method
   Can replace Strategy classes with lambda

In Java may need to define lambda type
Java Sorting Using Lambda

List<Students> students = new ArrayList<>();

add students

students.sort( (a, b) -> (a.gpa() <= b.gpa()) ? -1 : 1);

decreasing GPA

doesn't work
Costs

Clients must be aware of different Strategies

Communication overhead between Strategy and Context

Increase number of objects
Benefits

Alternative to subclassing of Context

Eliminates conditional statements

Replace in Context code like:

```java
switch ( flag ) {
    case A: doA(); break;
    case B: doB(); break;
    case C: doC(); break;
}
```

With code like:

```java
strategy.do();
```

Gives a choice of implementations
Refactoring: Replace Conditional Logic with Strategy

Conditional logic in a method controls which of several variants of a calculation are executed

so

Create a Strategy for each variant and make the method delegate the calculation to a Strategy instance
Replace Conditional Logic with Strategy

class Foo {
    public void bar() {
        switch ( flag ) {
            case A: doA(); break;
            case B: doB(); break;
            case C: doC(); break;
        }
    }
}

class Foo {
    private strategy;
    public void bar() {
        strategy.do(data);
    }
}

See Refactoring to Patterns, Kerievsky, 2005, pp 129–143 for detail steps
Null Object
Null Object

NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing
class LinkedList {
    Node head;

    public toString() {
        if (head == nil) {
            return "()";
        }
        String listAsString = "(";
        Node current = head;
        while (current != null) {
            listAsString += current.value() + ", ";
        }
        listAsString = removetail(listAsString, 2);
        return listAsString + ")";
    }
}
```java
class LinkedList {
    Node head;

    public toString() {
        return head.toString();
    }
}

class HeadNode {
    public String toString() {
        return "(" + next.toString();
    }
}

class Node {
    public String toString() {
        return " " + element + next.toString();
    }
}

class TailNode {
    public String toString() {
        return ")";
    }
}
```
Applicability - When to use Null Objects

Some collaborator instances should do nothing

You want clients to ignore the difference between a collaborator that does something and one that does nothing

Client does not have to explicitly check for null or some other special value

You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way
Applicability - When not to use Null Objects

Very little code actually uses the variable directly

The code that does use the variable is well encapsulated

The code that uses the variable can easily decide how to handle the null case and will always handle it the same way
Consequences

Advantages

Uses polymorphic classes

Simplifies client code

Encapsulates do nothing behavior

Makes do nothing behavior reusable

Disadvantages

Forces encapsulation

Makes it difficult to distribute or mix into the behavior of several collaborating objects

May cause class explosion

Forces uniformity

Is non-mutable
Implementation

Too Many classes

Multiple Do-nothing meanings

Try Adapter pattern

Transformation to RealObject

Try Proxy pattern
Refactoring: Introduce Null Object

You have repeated checks for a null value

Replace the null value with a null object

if (customer == null)
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();

plan = customer.getPlan();
Create Null Subclass

```java
public boolean isNull() { return false;
public static Customer newNull() { return new NullCustomer();

boolean isNull() { return true;
```

Compile
Replace all nulls with null object

```java
class SomeClassThatReturnCustomers {

    public Customer getCustomer() {
        if (_customer == null )
            return Customer.newNull();
        else
            return _customer;
    }

    etc.
}

```

Compile
Replace all null checks with isNull()

```java
if (customer == null)  
    plan = BillingPlan.basic();
else  
    plan = customer.getPlan();
```

```java
if (customer.isNull())  
    plan = BillingPlan.basic();
else  
    plan = customer.getPlan();
```

Compile and test

What is the point of this step?
Find an operation clients invoke if not null
Add Operation to Null class

if (customer.isNull())
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();

class NullCustomer {
    public BillingPlan getPlan() {
        return BillingPlan.basic();
    }
}
if (customer.isNotNull())
    plan = BillingPlan.basic();
else
    plan = customer.getPlan();

plan = customer.getPlan();

Compile & Test
Repeat last two slides for each operation
clients check if null
Special Case
Special Case

Represent special cases by a subclass

Use when multiple places that have same behavior

After conditional check for particular class instance

Or same behavior after a null check
Object-Oriented Recursion
A method polymorphically sends its message to a different receiver

Eventually a method is called that performs the task

The recursion then unwinds back to the original message send
class LinkedList {
    Node head;

    public toString() {
        return head.toString();
    }
}

class HeadNode {
    public String toString() {
        return "(" + next.toString();
    }
}

class TailNode {
    public String toString() {
        return ")";
    }
}

class Node {
    public String toString() {
        return " " + element + next.toString();
    }
}

class Node {  
    public String toString() {
        return " " + element + next.toString();
    }
}
class LinkedList {
    Node head;

    public void add(int value) {
        return head.add(value);
    }
}

class HeadNode {
    public void add(int value) {
        next.add(value);
    }
}

class TailNode {
    public void add(int value) {
        prependNode(value);
    }
}

class Node {
    public void add(int value) {
        if (element > value)
            prependNode(value);
        else
            next.add(value);
    }
}