Agile Design Principles: The Open/Closed Principle

Based on Chapter 9 of Robert C. Martin, *Agile Software Development: Principles, Patterns, and Practices*, Prentice Hall, 2003.

The Open/Closed Principle OCP

- “Software entities (classes, modules, functions etc) should be open for extension but closed for modification”
- The most straight-forward way to extend software is often to modify it e.g. by introducing if-else or switch statements.
- This often introduces brittleness and does not promote reusability.

The Open/Closed Principle OCP

- Instead we should design software entities so that future changes on the same axis require no modification.

OCP in action

- Suppose we need to sort a table representing email messages by time

```
class Sorter {
  public void sort( TableEntry [] a ) {
    for( int i = 0 ; i < a.length-1 ; ++i ) {
      int j = i ;  TableEntry min = a[j] ;
       for( int k = i+1 ; k < a.length() ; ++k ) {
         if( min.getTime().compareTo( a[k].getTime() ) > 0 ) {
           j = k ; min = a[j] ; } }
       a[j] = a[i] ; a[i] = min ; } } }
```

A change

- Sometimes sort by fromAddress

```
class Sorter {
  public void sort( TableEntry [] a, Column col ) {
    assert col==TIME || col==FROM ;
    for( int i = 0 ; i < a.length-1 ; ++i ) {
      int j = i ; TableEntry min = a[j] ;
       for( int k = i+1 ; k < a.length() ; ++k ) {
         if( col == TIME &&
           min.getTime().compareTo( a[k].getTime() ) > 0
         || min.getFrom().compareTo( a[k].getFrom() ) > 0 ) {
            j = k ; min = a[j] ; } }
       a[j] = a[i] ; a[i] = min ; } } }
```

Refactor

- A better plan is to refactor, factoring out the comparison into another class

```
class Sorter {
   private Comparator comparator ;
   public Sorter(Comparator comparator) { this.comparator = comparator ; }
   public void sort( TableEntry [] a) {
      for( int i = 0 ; i < a.length-1 ; ++i ) {
         int j = i ; TableEntry min = a[j] ;
          for( int k = i+1 ; k < a.length() ; ++k ) {
             if( comparator.compare( min, a[k] ) > 0 ) {
                j = k ; min = a[j] ; } }
          a[j] = a[i] ; a[i] = min ; } } }
interface Comparator {
   public int compare( TableEntry x, TableEntry y ) ; }
```

Extend

- Now the original call becomes

```
Comparator timeComparator =
     new Comparator() {
               int compare( TableEntry x, TableEntry y ) {
                    Time xTime = x.getTime() ;
                    Time yTime = y.getTime() ;
                    return xTime.compareTo( yTime ) ; } } ) ;
Sorter timeSorter = new Sorter( timeComparator ) ;
timeSorter.sort( table ) ;
```

An Open and Closed Case

- The Sorter class is now Open/Closed with respect to the axis of “comparison method”
- Closed: We should never have to modify it to accommodate other methods of comparison
- Open: It can be extended with new comparison methods

Another example

- We need to calculate HST tax on items

```
class Item {
    double cost ;
    double hst() { return 0.13 * cost ; }
}
```

- But some items have no GST and some have no PST. Here is a brittle change

```
class Item {
    double cost ;
    boolean gstApplies ;
    boolean pstApplies ;
    double hst() {
        if( gstApplies && pstApplies ) return 0.13 * cost ;
        else if( gstApplies ) return 0.5 * cost ;
        … }
}
```

Another example

- can you develop a solution that supports the open/closed principle?

A Question

- From the example we see that the Strategy pattern supports the Open/Closed Principle.
- Which patterns support OCP and which do not?