CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 2 Module Coupling & Cohesion
Contents

References

Object Coupling and Object Cohesion, chapter 7 of ***Essays on Object-Oriented Software Engineering***, Vol. 1, Berard, Prentice-Hall, 1993

Quality of Objects

Decomposing systems into smaller pieces aids software development

100 functions each 100 line of code long is "better" than

One function 10,000 lines of code long

Parnas (72) KWIC (Simple key word in context) experiment

Parnas compared two different implementations

- Modules based on steps needed to perform task

  Write down in order list of high level tasks to be done

  Each high level task becomes a module (function)

- Modules based on "design decisions"

  List
  - Difficult design decisions
  - Design decisions that are likely to change

  Each module should hide a design decision

All ways of decomposing an application are not equal

Parnas's Criteria

Primary goal of decomposition into modules is reduction of software cost

Specific goals of module decomposition

Each module's structure should be simple enough to understood fully

- It should be possible to change the implementation of one module without knowledge of the implementation of other modules and without affecting the behavior of other modules
- The ease of making a change in the design should bear a relationship to the likelihood of the change needed
- It should be possible to make a major software change as a set of independent changes to individual modules
- A software engineer should be able to understand the responsibility of a module without understanding the module's internal design

Concepts for quality

Coupling

Strength of interaction between objects in system

Cohesion

Degree to which the tasks performed by a single module are functionally related

If some ways of decomposing a system into modules are better than others, how do we tell which are better. When we design/implement a system how can we tell how we are doing. Coupling and cohesion are two concepts that help us out. Understanding these ideas help us design better.

Coupling

Decomposable system

One or more of the components of a system have no interactions or other interrelationships with any of the other components at the same level of abstraction within the system

A nearly decomposable system

Every component of the system has a direct or indirect interaction or other interrelationship with every other component at the same level of abstraction within the same system

Design Goal

The interaction or other interrelationship between any two components at the same level of abstraction within the system be as weak as possible

Coupling

Measure of the interdependence among modules

"***Unnecessary object coupling needlessly decreases the reusability of the coupled objects***"

"***Unnecessary object coupling also increases the chances of system corruption when changes are made to one or more of the coupled objects"***

Types of Modular Coupling
In order of desirability

Data Coupling (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling(strongest – least desirable)

Content Coupling (Unrated)

Modular Coupling
Data Coupling

Output from one module is the input to another

Using parameter lists to pass items between routines

Common Object Occurrence:

Object A passes object X to object B

Object X and B are coupled

A change to X's interface may require a change to B

Example

```
class Receiver
  {
  public void message( MyType X )
    {
    // code goes here
    X.doSomethingForMe( Object data );
    // more code
    }
  }
```

Modular Coupling
Data Coupling

Major Problem

Object A passes object X to object B

X is a compound object

Object B must extract component object Y out of X

B, X, internal representation of X, and Y are coupled

Example: Sorting student records, by ID, by Name

How does the SortedList method add() add the new student record object and resort the list? To do this it needs to access the ID (name) fields of StudentRecord!

```
class StudentRecord {
   Name lastName;
   Name firstName;
   long ID;

   public Name getLastName() { return lastName; }

   // etc.
}

SortedList cs535 = new SortedList();
StudentRecord newStudent;
//etc.
cs535.add ( newStudent );
```

Solution 1 Bad News

Here the add method actually accesses the StudentRecord method to get the ID. What is wrong with that? Why is this bad news?

```
class SortedList
  {
  Object[] sortedElements = new Object[ properSize ];

  public void add( StudentRecord X )
    {
    // coded not shown
    Name a = X.getLastName();
    Name b = sortedElements[ K ].getLastName();
    if ( a.lessThan( b ) )
      // do something
    else
      // do something else
    }
  }
```

Solution 2 Send message to object to compare self to another StudentRecord Object

How is this any better that solution 1? Is it any better? How does it differ?

```
class SortedList{
   Object[] sortedElements = new Object[ properSize ];

   public void add( StudentRecord X ) {
      // coded not shown
      if ( X.lessthan( sortedElements[ K ] ) )
         // do something
      else
         // do something else
   }
}

class StudentRecord{
   private Name lastName;
   private long ID;

   public boolean lessThan( Object compareMe ) {
      return lastName.lessThan( compareMe.lastName );
   }
   etc.
}
```

Solution 3 Interfaces or "required operations"

Notice how the SortedList is no longer coupled to the StudentRecord class. It can be used to sort any list of objects of the same class than implement Comparable.

```
interface Comparable {
   public boolean lessThan( Object compareMe );
   public boolean greaterThan( Object compareMe );
   public boolean equal( Object compareMe );
}

class StudentRecord implements Comparable {
   private Name lastName;
   private long ID;

   public boolean lessThan( Object compareMe ) {
      return lastName.lessThan( ((Name)compareMe).lastName );
   }
}

class SortedList {
   Object[] sortedElements = new Object[ properSize ];

   public void add( Comparable X ) {
      // coded not shown
      if ( X.lessthan( sortedElements[ K ] )
         // do something
      else
         // do something else
      }
   }
```

Solution 4 Function Pointers

Code is neither legal C/C++ nor Java. The idea is to pass in a function pointer to the SortList object, which it uses to compare the objects in the list.

```
typedef int (*compareFun ) ( StudentRecord, StudentRecord );
class SortedList {
   StudentRecord[] sortedElements =
         new StudentRecord[ properSize ];

    int (*compare ) ( StudentRecord, StudentRecord );

   public setCompare( compairFun newCompare )
      { compare = newCompare; }

   public void add( StudentRecord X ) {
      // coded not shown
      if ( compare( X,  sortedElements[ K ] ) )
      // code not shown
   }
}

int compareID( StudentRecord a, StudentRecord b )
   { // code not shown }

int compareName( StudentRecord a, StudentRecord b )
   { // code not shown }

SortedList myList = new SortedList();
myList.setCompair( compareID );
```

Functor Pattern
Functions as Objects

Functors are functions that behave like objects

They serve the role of a function, but can be created, passed as parameters, and manipulated like objects

A functor is a class with a single member function

Note 1: Functors violate the idea that a class is an abstraction with operations and state. Beginners should avoid using the Functor pattern, as they can lead to bad habits. The functor pattern is used here only as a last resort.

Note 2: The Command pattern is similar to the Functor pattern, but contains operations and state.

Note 3: For more information about patterns see: http://www.eli.sdsu.edu/courses/spring98/cs635/notes/patternIntro/patternIntro.html .For more information about Functor and Command pattern see: http://www.eli.sdsu.edu/courses/spring98/cs635/notes/command/command.html

Function Pointers in Java
Comparator in Java 2 (JDK 1.2)

In Java 2, the Comparator interface defines an interface for objects that act like functions pointers to compare objects.

Methods in Comparator Interface

int compare(Object o1, Object o2)
Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second

boolean equals(Object obj)
Indicates whether some other object is "equal to" this Comparator.

The implementer must ensure that:

sgn(compare(x, y)) == -sgn(compare(y, x)) for all x and y

compare(x, y) must throw an exception if and only if compare(y, x) throws an exception.)

((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0.

x.equals(y) || (x==null && y==null) implies that compare(x, y)==0.

compare(x, y)==0 implies that sgn(compare(x, z))==sgn(compare(y, z)) for all z.

Comparator Example

```
import java.util. Comparator;

class Student  {
   String name;
   int id;

   public Student( String newName, int id ) {
      name = newName;
      this.id = id;
      }

   public String toString() {
      return name + ":" + id;
      }
   }

final class StudentNameComparator implements Comparator {

   public int compare( Object leftOp, Object rightOp ) {
      String leftName = ((Student) leftOp).name;
      String rightName = ((Student) rightOp).name;
      return leftName.compareTo( rightName );
   }

   public boolean equals( Object comparator ) {
      return comparator instanceof StudentNameComparator;
   }
}
```

//Comparator Example Continued

```
final class StudentIdComparator implements Comparator {
   static final int LESS_THAN = -1;
   static final int GREATER_THAN = 1;
   static final int EQUAL = 0;

   public int compare( Object leftOp, Object rightOp ) {
      long leftId = ((Student) leftOp).id;
      long rightId = ((Student) rightOp).id;
      if ( leftId < rightId )
         return LESS_THAN;
      else if ( leftId > rightId )
         return GREATER_THAN;
      else
         return EQUAL;
   }

   public boolean equals( Object comparator ) {
      return comparator instanceof StudentIdComparator;
   }
}
```

//Comparator Example Continued

```
import java.util.*;

public class Test  {
   public static void main(String args[])  {
      Student[] cs596 = { new Student( "Li", 1 ), new Student( "Swen", 2 ),
                                 new Student( "Chan", 3 ) };

      //Sort the array
      Arrays.sort( cs596, new StudentNameComparator() );
      for ( int k = 0; k < cs596.length; k++ )
         System.out.print( cs596[k].toString() + ", " );
      System.out.println( );

      List cs596List = new ArrayList( );
      cs596List.add( new Student( "Li", 1 ) );
      cs596List.add( new Student( "Swen", 2 ) );
      cs596List.add( new Student( "Chan", 3 ) );
      System.out.println( "Unsorted list " + cs596List );

      //Sort the list
      Collections.sort( cs596List,  new StudentNameComparator() );
      System.out.println( "Sorted list " + cs596List );

      //TreeSets are aways sorted
      TreeSet cs596Set = new TreeSet(  new StudentNameComparator() );
      cs596Set.add( new Student( "Li", 1 ) );
      cs596Set.add( new Student( "Swen", 2 ) );
      cs596Set.add( new Student( "Chan", 3 ) );
      System.out.println( "Sorted Set " + cs596Set );
   }
}
```

//Comparator Example Continued
Output

Chan:3, Li:1, Swen:2,
Unsorted list [Li:1, Swen:2, Chan:3]
Sorted list [Chan:3, Li:1, Swen:2]
Sorted Set [Chan:3, Li:1, Swen:2]

Sorting With Different Keys

```
import java.util.*;

public class MultipleSorts {
   public static void main(String args[]) {

      List cs596List = new ArrayList( );
      cs596List.add( new Student( "Li", 1 ) );
      cs596List.add( new Student( "Swen", 2 ) );
      cs596List.add( new Student( "Chan", 3 ) );

      Collections.sort( cs596List,  new StudentNameComparator() );
      System.out.println( "Name Sorted list " + cs596List );

      Collections.sort( cs596List,  new StudentIdComparator() );
      System.out.println( "Id Sorted list " + cs596List );

      TreeSet cs596Set = new TreeSet( new StudentNameComparator());
      cs596Set.addAll( cs596List );
      System.out.println( "Name Sorted Set " + cs596Set );

      TreeSet cs596IdSet = new TreeSet( new StudentIdComparator() );
      cs596IdSet.addAll( cs596List );
      System.out.println( "Id Sorted Set " + cs596IdSet );
   }
}
```

Output

```
Name Sorted list [Chan:1, Li:2, Swen:1]
Id Sorted list [Chan:1, Swen:1, Li:2]
Name Sorted Set [Chan:1, Li:2, Swen:1]
Id Sorted Set [Chan:1, Li:2]
```

Solution 5 Function Pointers using generic types

```
typedef int (*compairFun ) ( <type>, <type> );
```

Modular Coupling
Control Coupling

Passing control flags between modules so that one module controls the sequencing of the processing steps in another module

Common Object Occurrences:

A sends a message to B

B uses a parameter of the message to decide what to do

```
class Lamp {
   public static final ON = 0;

   public void setLamp( int setting ) {
      if ( setting == ON )
         //turn light on
      else if ( setting == 1 )
         // turn light off
      else if ( setting == 2 )
         // blink
   }
}

Lamp reading = new Lamp();
reading.setLamp( Lamp.ON );
reading.setLamp)( 2 );
```

Cure:
Decompose the operation into multiple primitive operations

```
class Lamp {
   public void on() {//turn light on }
   public void off() {//turn light off }
   public void blink() {//blink }
}


Lamp reading = new Lamp();
reading.on();
reading.blink();
```

Control Coupling

Common Object Occurrences:

A sends a message to B

B returns control information to A

Example: Returning error codes

```
class Test {
   public int printFile( File toPrint ) {
      if ( toPrint is corrupted )
         return CORRUPTFLAG;
      blah blah blah
   }
}


Test when = new Test();
int result =  when.printFile( popQuiz );
if ( result == CORRUPTFLAG )
   blah
else if ( result == -243 )
```

Cure: Use exceptions

How does this reduce coupling?

```
class Test  {
   public int printFile( File toPrint ) throws PrintExeception {
      if ( toPrint is corrupted )
         throws new PrintExeception();
      blah blah blah
   }
}

try {
   Test when = new Test();
   when.printFile( popQuiz );
}
catch ( PrintException printError ) {
   do something
}
```

Modular Coupling
Global Data Coupling

Two or more modules share the same global data structures

Common Object Occurrence:

A method in one object makes a specific reference to a specific external object

A method in one object makes a specific reference to a specific external object, and to one or more specific methods in the interface to that external object

A component of an object-oriented system has a public interface which consists of items whose values remain constant throughout execution, and whose underlying structures/implementations are hidden

A component of an object-oriented system has a public interface which consists of items whose values remain constant throughout execution, and whose underlying structures/implementations are ***not*** hidden

A component of an object-oriented system has a public interface which consists of items whose values ***do not*** remain constant throughout execution, and whose underlying structures/implementations are hidden

A component of an object-oriented system has a public interface which consists of items whose values ***do not*** remain constant throughout execution, and whose underlying structures/implementations are ***not*** hidden

Internal Data Coupling

One module directly modifies local data of another module

Common Object Occurrence:

C++ Friends

A friend of a class in C++ has complete access to all private members of the class. This is a clear violation of the information hiding feature of the class. Since the class must list its friends, the violation is controlled. There are situations (defining the io operators <<, >>) where the use of friends can not be avoided

Modular Coupling
Lexical Content Coupling

Some or all of the contents of one module are included in the contents of another

Common Object Occurrence:

C/C++ header files

Decrease coupling by:

Restrict what goes in header file

C++ header files should contain only class interface specifications

Cohesion

"Cohesion is the degree to which the tasks performed by a single module are functionally related."

IEEE, 1983

"Cohesion is the "glue" that holds a module together. It can be thought of as the type of association among the component elements of a module. Generally, one wants the highest level of cohesion possible."

Bergland, 1981

"A software component is said to exhibit a high degree of cohesion if the elements in that unit exhibit a high degree of functional relatedness. This means that each element in the program unit should be essential for that unit to achieve its purpose."

Sommerville, 1989

Types of Module Cohesion
From Worst to Best

Coincidental (worst)

Logical

Temporal

Procedural

Communication

Sequential

Functional (best)

Module Cohesion
Coincidental

Little or no constructive relationship among the elements of the module

Common Object Occurrence:

Object does not represent any single object-oriented concept

Collection of commonly used source code as a class inherited via multiple inheritance

```
class Rous
  {
  public static int findPattern( String text, String pattern)
    { // blah}

  public static int average( Vector numbers )
    { // blah}

  public static OutputStream openFile( String fileName )
    { // blah}

  }
```

Module Cohesion
Logical

Module performs a set of related functions, one of which is selected via function parameter when calling the module

Similar to control coupling

Cure:

Isolate each function into separate operations

```
public void sample( int flag )
  {
  switch ( flag )
    {
    case ON:
      // bunch of on stuff
      break;
    case OFF:
      // bunch of off stuff
      break;
    case CLOSE:
      // bunch of close stuff
      break;
    case COLOR:
      // bunch of color stuff
      break;
    }
  }
```

Module Cohesion
Temporal

Elements are grouped into a module because they are all processed within the same limited time period

Common example:

"Initialization" modules that provide default values for objects

"End of Job" modules that clean up

```
procedure initializeData()
   {
   font = "times";
   windowSize = "200,400";
   foo.name = "Not Set";
   foo.size = 12;
   foo.location = "/usr/local/lib/java";
   }
```

Cure: Each object should have a constructor and destructor

```
class foo
   {
   public foo()
      {
      foo.name = "Not Set";
      foo.size = 12;
      foo.location = "/usr/local/lib/java";
      }
   }
```

Sample Configuration File

[Macintosh]	[General]
EquationWindow=146,171,406,661	Zoom=200
SpacingWindow=0,0,0,0	CustomZoom=150
	ShowAll=0
	Version=2.01
[Spacing]	OptimalPrinter=1
LineSpacing=150%	MinRect=0
MatrixRowSpacing=150%	ForceOpen=0
MatrixColSpacing=100%	ToolbarDocked=1
SuperscriptHeight=45%	ToolbarShown=1
SubscriptDepth=25%	ToolbarDockPos=1
LimHeight=25%	
LimDepth=100%	[Fonts]
LimLineSpacing=100%	Text=Times
NumerHeight=35%	Function=Times
DenomDepth=100%	Variable=Times,I
FractBarOver=1pt	LCGreek=Symbol,I
FractBarThick=0.5pt	UCGreek=Symbol
SubFractBarThick=0.25pt	Symbol=Symbol
FenceOver=1pt	Vector=Times,B
SpacingFactor=100%	Number=Times
MinGap=8%	
RadicalGap=2pt	[Sizes]
EmbellGap=1.5pt	Full=12pt
PrimeHeight=45%	Script=7pt
	ScriptScript=5pt
	Symbol=18pt
	SubSymbol=12pt

Call these constructors/destructors from a nonobject-oriented routine that performs a single, cohesive task

Module Cohesion
Procedural

Associates processing elements on the basis of their procedural or algorithmic relationships

Procedural modules are application specific

In context the module seems reasonable

Removed from the context these modules seem strange and very hard to understand

Why is that being done here?

Can not understand module without understanding the program and the conditions existing when module is called

Makes module hard to modify, understand

Class Builder verse **Program writer**

Cure:

Redesign the system

If a module is necessary, remove it from objects

Module Cohesion
Communication

Operations of a module all operate upon the same input data set and/or produce the same output data

Cure:

Isolate each element into separate modules

Rarely occurs in object-oriented systems due to polymorphism

Module Cohesion
Sequential

Sequential association the type in which the output data from one processing element serve as input data for the next processing element

A module that performs multiple sequential functions where the sequential relationship among all of the functions is implied by the problems or application statement and where there is a data relationship among all of the functions

Cure:

Decompose into smaller modules

Module Cohesion
Functional

If the operations of a module can be collectively described as a single specific function in a coherent way, the module has functional cohesion

If not, the module has lower type of cohesion

In an object-oriented system:

- Each operation in public interface of an object should be functional cohesive
- Each object should represent a single cohesive concept

Module Cohesion
Informational Strength

Myers states:

"The purpose of an informational-strength module is to hide some concept, data structure, or resource within a single module.

An informational-strength module has the following definition:

- It contains multiple entry points
- Each entry point performs a single specific function
- All of the functions are related by a concept, data structure, or resource that is hidden within the module"