Object Coupling and Object Cohesion, chapter 7 of Essays on Object-Oriented Software Engineering, Vol. 1, Berard, Prentice-Hall, 1993, pp. 72-86

On the Criteria To Be Used in Decomposing Systems into Modules, D. L. Parnas, http://www.acm.org/classics/may96/
In the Beginning

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

Read lines of words
Output all circular shifts of all lines in alphabetical order
Circular shift
remove first word of line and add it to the end of the line

**Solution 1**
Each major step in processing is a module
Create flowchart and make each major part a module

**Solution 2**
Modules based on design decisions
List design decisions that are
Difficult
Likely to change
Each module should hide a design decision

**Solution 1**
More complex
Harder to understand
Much harder to modify
Metrics for Quality

**Coupling**

Strength of interaction between objects in system

**Cohesion**

Degree to which the tasks performed by a single module are functionally related
## Relationships between Objects

<table>
<thead>
<tr>
<th>Type of Relation</th>
<th>Relation between</th>
</tr>
</thead>
<tbody>
<tr>
<td>Uses</td>
<td>(Object)</td>
</tr>
<tr>
<td>Containment</td>
<td>(Object)</td>
</tr>
<tr>
<td>Inheritance</td>
<td>(Class)</td>
</tr>
</tbody>
</table>

### Uses

Object A uses object B if A sends a message to B
Assume that A and B objects of different classes
A is the sender, B is the receiver

### Containment

Class A contains class B when A has a field of type B
That is an object of type A will have an object of type B inside it
How one object can use another

How does the sender access the receiver?

**Containment**

The receiver is a field in the sender

```java
class Sender {
    Receiver here;

    public void method() {
        here.sendAMessage();
    }
}
```
How one object can use another

Argument of a method

The receiver is an argument in one of the sender's methods

class Sender {
    public void method(Receiver here) {
        here.sendAMessage();
    }
}

How one object can use another

Ask someone else

The sender asks someone else to give them the receiver

class Sender {
    public void method() {
        Receiver here = someoneElse.getReceiver();
        here.sendAMessage();
    }
}
How one object can use another

Creation

The sender creates the receiver

class Sender {
    public void method() {
        Receiver here = new Receiver();
        here.sendAMessage();
    }
}


How one object can use another

Global

The receiver is global to the sender
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"

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
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)
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 ObjectBClass{
    public void message( ObjectXClass X ){
        // code goes here
        X.doSomethingForMe( Object data );
        // more code
    }
}
Data Coupling

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

public class HiddenCoupling {
    public bar someMethod(SomeType x) {
        AnotherType y = x.getY();
        y.foo();
        blah;
    }
}
Example – Sorting

How to write a general purpose sort
Sort the same list by
   ID
   Name
   Grade

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

   public Name getLastName() { return lastName; }

   // etc.
}

SortedList cs635 = new SortedList();
StudentRecord newStudent;
//etc.
cs535.add ( newStudent );
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
    }
}
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.
interface Comparable {  
    public boolean lessThan( Object compareMe );
    public boolean greaterThan( Object compareMe );
    public boolean equal( Object compareMe );
}

class StudentRecord implements Comparable {
    blah
    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
    }
}
interface Comparing {
    public boolean lessThan( Object a, Object b );
    public boolean greaterThan( Object a, Object b );
    public boolean equal( Object a, Object b );
}

class StudentNameComparing implements Comparing {
    public boolean lessThan( Object a, Object b ) {
        return ((Student) a).lastName() < ((Student) b).lastName();
    }
    etc.
}

class SortedList {
    Object[] sortedElements = new Object[ properSize ];
    Comparing comparer;
    public SortedList(Comparing y) {comparer = y;}

    public void add( Comparable X ) {
        // coded not shown
        if ( X.lessThan( sortedElements[ K ] )
            // do something
        else
            // do something else
    }
}
C++ Version

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

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.
Types of Coupling

Data Coupling  (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling  (strongest – least desirable)

Content Coupling  (Unrated)
Control Coupling

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

Common Object Occurrence

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

```java
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 );
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.on();
reading.blink();
class BankAccount {
    public void withdrawal(Float amount) {
        balance = balance - amount;
    }
}

Is this Control Coupling

class BankAccount {
    public void withdrawal(Float amount) {
        if (balance < amount)
            this.bounceThisCheck();
        else
            balance = balance - amount;
    }
}

Is this Control Coupling

etc.

etc.
Control Coupling

Common Object Occurrence

A sends a message to B
B returns control information to A

Example: Returning error codes

```java
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 PrintException {
        if ( toPrint is corrupted )
            throws new PrintException();
        blah blah blah
    }
}

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

Data Coupling  (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling  (strongest – least desirable)

Content Coupling  (Unrated)
Global Data Coupling

Global Data is evil
Global Data Coupling

What are the following?

System.out
Integer.MAX_VALUE
Types of Global Data Coupling in increasing order of "badness"

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
Types of Coupling

Data Coupling  (weakest – most desirable)

Control Coupling

Global Data Coupling

**Internal Data Coupling**  (strongest – least desirable)

Content Coupling  (Unrated)
Internal Data Coupling

One module directly modifies local data of another module

Common Object Occurrences

C++ Friends
Smalltalk reflection
Java reflection
Internal Data Coupling

Implement a debugger without using internal data coupling
Types of Coupling

Data Coupling  (weakest – most desirable)

Control Coupling

Global Data Coupling

Internal Data Coupling  (strongest – least desirable)

Content Coupling  (Unrated)
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