CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2005
Doc 8 Coupling
Contents

References

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

Object-Oriented Design Heuristics, Riel, Addison-Wesley, 1996

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

Reading

Object Coupling and Object Cohesion. pp. 72-86, 92-111

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

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

Coupling

Relationships between Objects

Type of Relations:

Type	Relation between
Uses	(Object)
Containment	(Object)
Inheritance	(Class)
Association	(Object)

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

Different Ways to Implement Uses

How does the sender access the receiver?

1. Containment

The receiver is a field in the sender

```
class Sender {
  Receiver here;

  public void method() {
     here.sendAMessage();
  }
}
```

2. 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();
  }
}
```

3. **Ask someone else**

The sender asks someone else to give them the receiver

```
class Sender {
  public void method() {
     Receiver here = someoneElse.getReceiver();
     here.sendAMessage();
  }
}
```

4. **Creation**

The sender creates the receiver

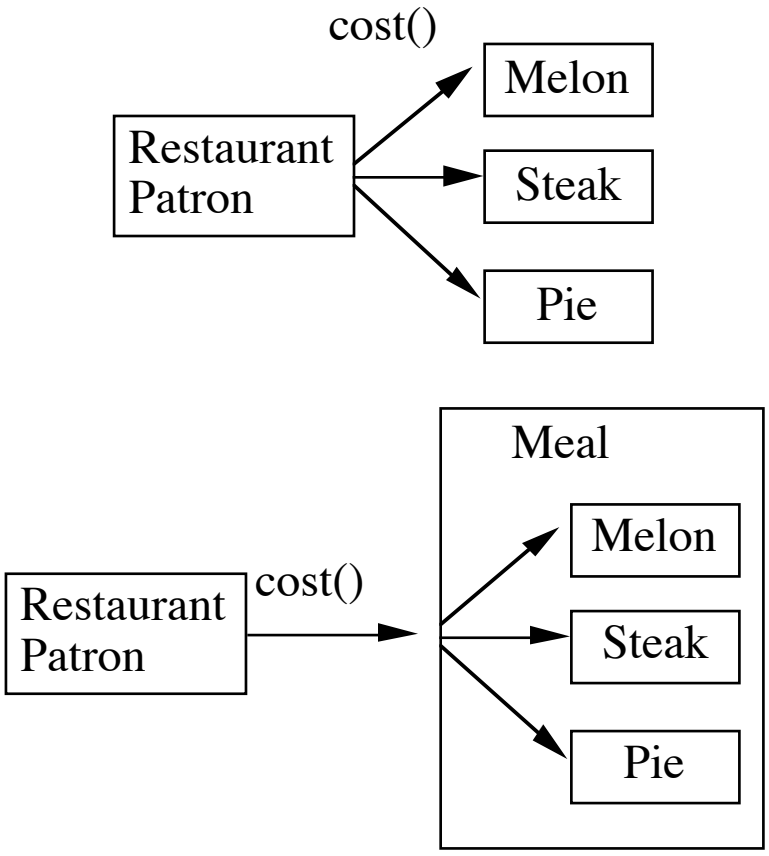
```
class Sender {
  public void method() {
     Receiver here = new Receiver();
     here.sendAMessage();
  }
}
```

5. **Global**

The receiver is global to the sender

Heuristics for the Uses Relationship

4.1 Minimize the number of classes with another class collaborates

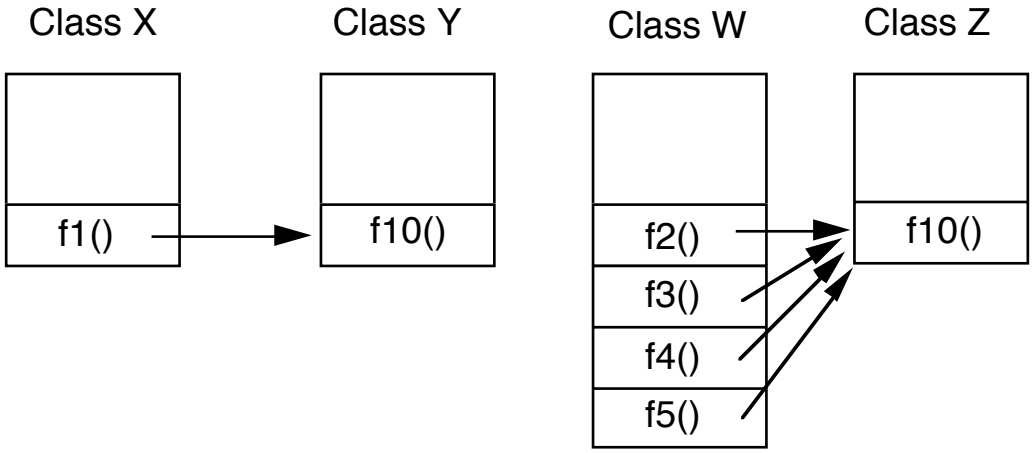


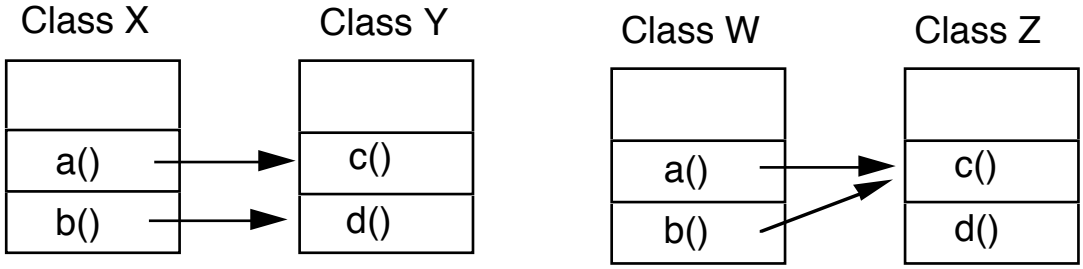
4.2 Minimize the number of message sends between a class and its collaborator

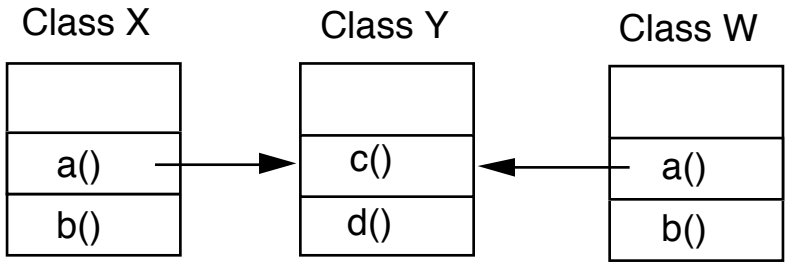
4.3 Minimize the number of different messages a class sends to another class.

4.4 Minimize the product of the number of methods in a class and the number of different messages they send.

Which is more complex?







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 ObjectBClass{
  public void message( ObjectXClass 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

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

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
      }
   }

SortList>>add: aStudentRecord
   Blah
   a := aStudentRecord lastName.
   b := sortedElements at: k.
   blah
```

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.
}

SortList>>add: aStudentRecord
   Blah
   aStudentRecord <  sortedElements last
      ifTrue: [ more blah ]
      ifFalse: [ blah blah ]
   blah
```

Solution 3 Program to an Interface 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
      }
   }
SortList>>add: anObject
   anObject <  sortedElements last
      ifTrue: [ more blah ]
      ifFalse: [ blah blah ]
   blah
```

Solution 4 Strategy Pattern & Blocks

```
| sortedStudents |

sortedStudents := SortedCollection sortBlock:
        [:x :y | x lastName < y lastName].

blah

sortedStudents
   add: roger;
   add: pete;
   add: sam.

sortedStudents sortBlock: [:x :y | x grade < y grade ]
```

Solution 4 Strategy Pattern & 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.

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]
```

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();
```

Is this Control Coupling?

```
BankAccount>>withdrawal: aFloat
   balance := balance – aFloat.
```

What about?

```
BankAccount>>withdrawal: aFloat
   balance < aFloat
      ifTrue: [ self bounceThisCheck ]
      ifFalse: [balance := balance – aFloat]
```

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

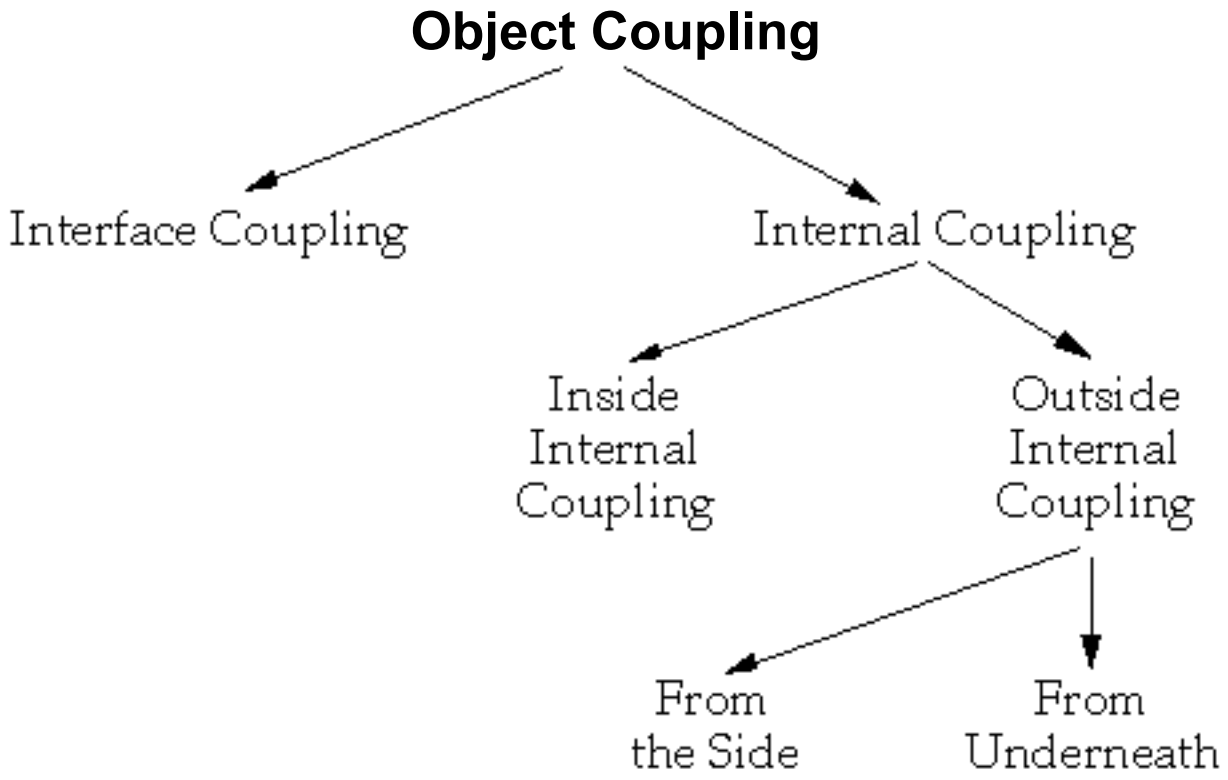
Common Object Occurrence:

C/C++ header files

Decrease coupling by:

Restrict what goes in header file

C++ header files should contain only class interface specifications



Very little is written about object coupling. For more information see "Managing Class Coupling: Apply the Principles of Structured Design to Object-Oriented Programming," UNIX Review, Vol. 2, No. 1, May/June 1989, pp. 34-40.

Coupling measures the strength of the physical relationships among the items that comprise an object

Cohesion measures the logical relationship among the items that comprise an object

Interface coupling is the coupling between an object and all objects external to it. Interface coupling is the most desirable form of object coupling. Internal coupling is coupling among the items that make up an object.

Object Coupling
Interface Coupling

Interface coupling occurs when one object refers to another specific object, and the original object makes direct references to one or more items in the specific object's public interface

Includes module coupling already covered

Weakest form of object coupling, but has wide variation

Sub-topics

- Object abstraction decoupling
- Selector decoupling
- Constructor decoupling
- Iterator decoupling

Object Abstraction Decoupling

Assumptions that one object makes about a category of other objects are isolated and used as parameters to instantiate the original object.

Example: List items

C++ templates and Ada's generics are the constructs Berard is talking about. Making the LinkedListCell a template removes any type specific code from the LinkedListCell class. This helps insure that the class can hold any type.

C++ Example

```
class LinkedListCell {
   int cellItem;
   LinkedListCell* next;

   // code can now use fact that cellItem is an int
   if ( cellItem == 5 ) print( "We Win" );
}

template <class type>
class LinkedListCell#2 {
   type cellItem;
   LinkedListCell* next;

   // code does not know the type, it is just a cell item,
   // it becomes an abstraction
}
```

Java Example

Object Abstraction Decoupling Java does not support templates. Instead it supports Object as a root type. Using an Object as a type in the LinkedListCell class has some of the decoupling that Ada generics or C++ templates achieve. However, it provides only one category of objects (all of them). This solution that Smalltalk (with no compile time type checking) also supports. Java interfaces can be used to achieve decoupling in the same situations as Ada generics or C++ templates.

```
class LinkedListCellA {
   int cellItem;
   LinkedListCell next;

   if ( cellItem == 5 ) print( "We Win" );
}

class LinkedListCellB {
   Object  cellItem;
   LinkedListCell next;

   if ( cellItem.operation1() ) print( "We Win" );
}
```

Selector Decoupling

Example: Counter object

```
class Counter{
   int count = 0;

   public void increment()    {  count++; }
   public void reset()     { count = 0; }
   public void display()        {
      code to display the counter in a slider bar
}
```

Display of Counter

"display" couples the counter object to a particular output type

The counter class can not be used in other setting due to this coupling

Better Counter Class

```
class Counter{
   int count = 0;

   public void increment()    {  count++; }
   public void reset()     { count = 0; }
   public int count()              {return count;}
   public String toString() {return String.valueOf( count );}
   }
```

Primitive Methods

A **primitive method** is any method that cannot be implemented simply, efficiently, and reliably without knowledge of the underlying implementation of the object

Primitive methods are:

Functionally cohesive, they perform a single specific function

Small, seldom exceed five "lines of code"

A **composite method** is any method constructed from two or more primitive methods – sometimes from different objects

Types of Primitive Operations

Selectors (get operations)

Constructors (not the same as class constructors)

Iterators

Selectors

Selectors are encapsulated operations which return state information about their encapsulated object and do not alter the state of their encapsulated object

Replacing

```
public void display()        {
   code to display the counter
}
```

with

```
public String toString() {return String.valueOf( count );}
```

is an example of Selector decoupling.

By replacing a composite method (display) with a primitive method the Counter class is decoupled from the display device

This makes the Counter class far more useful

It also moves the responsibility of displaying the counter elsewhere

Constructors

Operations that construct a new, or altered version of an object

Java and C++ both have language constructs called constructors. Berard has in mind a larger class of operations than those. Often static methods are used as constructors to create new objects.

Berard's example illustrating constructor decoupling is extremely vague. The fromString method below does make it clear what type of parameter is needed to create a new calendar object. One point to learn from his discussion is the desirability to have well defined interface to creating objects from primitive objects.

```
class Calendar {
   public void getMonth( from where, or what) { blah }
}

class Calendar {
   public static Calendar fromString( String date ) { blah}
}
```

Primitive Objects

Primitive objects are objects that are both:

- Defined in the standard for the implementation language

  This can include standard libraries and standard environments

- Globally known

  That is any object that is known in any part of any application created using the implementation language

Primitive objects don't count in coupling with other objects

"*An object that refers to itself and to primitive objects is considered for all intents and purposes, totally decoupled from other objects***"**

The motivation here is that primitive objects are very stable, that is will not change. If they do not change, then we do not have to be concerned about coupling with them. One reason to reduce coupling is to make it easier to deal with changes. A second reason to reduce coupling is to improve reuse. If class A uses class B, which is universally available to all programs using the language, then class A's reusability is not affected by using class B. Berard's argument has two problems. First, standard libraries do change over time. Look at the number of deprecated methods in the Java API. Of course, the Java API is very young. As the language ages, its core API should be more stable. The second problem is one can delude oneself about a company's or personal class library as being "standard" and stable (and hence primitive) when they are not.

Composite Object

Object **conceptually** composed of two or more objects

Heterogeneous Composite Object

Object **conceptually** composed from objects which are not all **conceptually** the same

The date class below is composed of three items that are the same type: ints. However, these ints represent different conceptual entities.

```
class Date{
   int year;
   int month;
   int day;
}
```

Homogeneous Composite Object

Object **conceptually** composed from objects which are all **conceptually** the same

list of names - each item is a member of the same general category of object – a name

Berard's homogeneous composite objects are basically container objects.

Iterator

Allows the user to visit all the nodes in a homogeneous composite object and to perform some user-supplied operation at each node

Both Java and C++ support iterators

Iterators and Coupling

Using iterators reduces coupling by hiding the details of traversing through elements of a collection. If one used the non-iterator method of accessing the elements of collections, it becomes a lot of work to replace the use of one collection with another. One might want to replace an array with a binary search tree for better performance.

Array

```
int[] list

for (int k = 0; k < list.length; k ++ )
   System.out.println( list[k] );
```

Vector

```
Vector list

for (int k =0; k < list.size(); k++ )
   Sytem.out.println( list.elementAt( k ) );
```

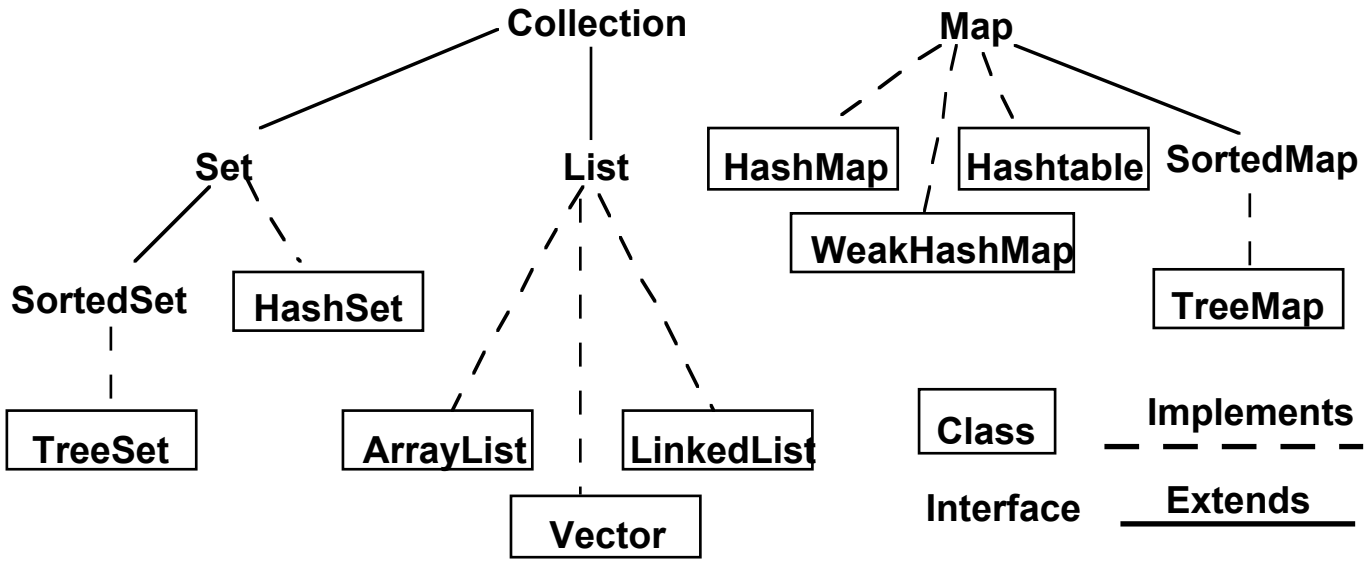
Binary Search Tree

```
BinarySeachTree list

Node current = list.root();
Stack previous = new Stack();
Previous.push( current );

while (current != null )
   {
   a lot of code here
   }
```

Java Collection Classes



There are synchronized, unsynchronized, modifiable unmodifiable versions of each collection/map

One can set the modifiable and synchronized property separately

What about Arrays?

One of Java's defects is not making an Array class and making it part of the collection class hierarchy. As a result one has to treat arrays differently from all other collections. Since arrays are very common, the effectiveness of the collection class hierarchy is greatly lessened. However, since most programmers have not used a uniform collection class structure they do not realize how much easier life can be.

One can convert an array of objects to a list

```
String[] example = new String[10];
List listBackedByArray = Arrays.asList( example );
```

Changes to the array(list) are reflected in the list(array)

Less Coupling with Iterators

```
Collection list;

Iterator elements = list.iterator();

while (elements.hasNext() ) {
   System.out.println( elements.next() );
}
```

In this code list could be any type of collection, so is more flexible. It is not coupled to a particular type of collection.

Inside Internal Object Coupling

Coupling between state and operations of an object

The big issue: Accessing state

Changing the structure of the state of an object requires changing all operations that access the state including operations in subclasses

Solution: Access state via access operations

C++ implementation

Provide private functions to access and change each data member

Simple Cases:

One function to access the value of the date member

One function to change the value of the data member

Only these two functions can access the data member

When an object is used as state, then providing access methods for that object can be far more complex. Assume that the state object itself has 10 methods. Now we may need to provide 12 access methods not just two. If a class have three such state objects, then it may need far too many access methods to be practical.

Accessing State
C++ Example

```
class Counter{
public:
   void increment(void);

private:
   int  value;

   void  setValue(int newValue);
   int  getValue(void);
};

void Counter::increment(void)    //Increase counter by one {
   setValue(getValue() + 1);
};

void Counter::setValue(int newValue) {
   value = newValue;
};

int Counter::getValue {
   return value;
};
```

Outside Internal Coupling from Underneath

Coupling between a class and subclass involving private state and private operations

Major Issues:

- Access to inherited state

  Direct access to inherited state

    See inside internal object coupling

  Access via operations

    Inherited operations may not be sufficient set of operations to access state for subclass

- Unwanted Inheritance

  Parent class may have operations and state not needed by subclass

  Unwanted inheritance makes the subclass unnecessarily complex. This reduces understandability and reliability.

Outside Internal Coupling from the Side

Class A accesses private state or private operations of class B

Class A and B are not related via inheritance

Main causes:

Using nonobject-oriented languages

Special language "features"

C++ friends

Donald Knuth

"First create a solution using sound software engineering techniques, then if needed, introduce small violations of good software engineering principles for efficiency's sake."