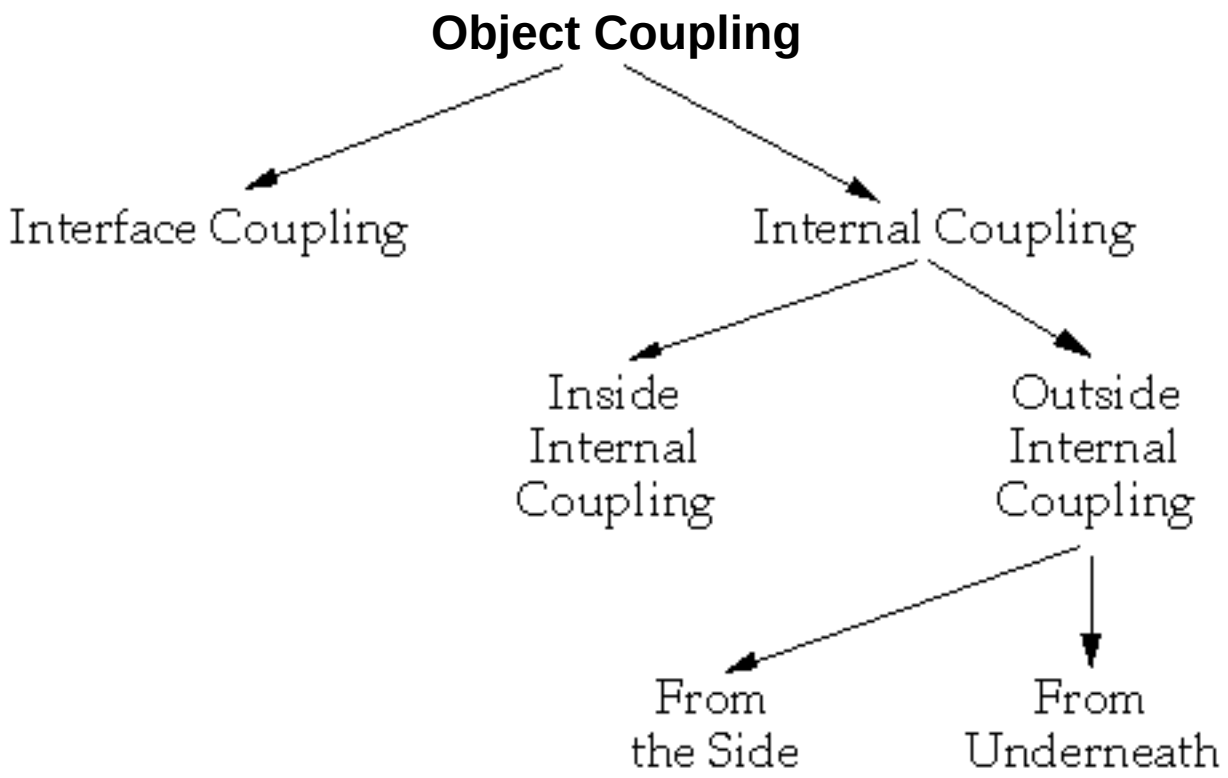
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2001
Doc 3 Object Coupling & Cohesion
Contents

References

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



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

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. The no compile time type checking solution is a common source of flame wars in the net. 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 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

Passive Iterator

Neither Java nor C++ support passive iterators. Smalltalk does support them. In a passive iterator, you pass a method or function to the composite object, and the object then applies the method to all elements in the object. Passive iterators in Smalltalk are very powerful. Passive iterators require very minimal code to use. They require efficient ways to deal with method/functions as parameters. Only one passive iterator can be active on an object at a time.

```
class List {
   Object[] listElements = new Object[ size ];

   public void do( Function userOperation ) {
      for ( int k = 0; k < listElements.length(); k++ )
         userOperation( listElements[ k ] );
   }
}
```

In Main

```
   List grades = new List();
   aFunction = ( item ){ print( item ) };

   grades.do ( aFunction );
```

Active Iterator

Java (Enumeration, Iterator (JDK1.2), ListIterator (JDK1.2), StringCharacterIterator) and C++ (in STL) use active iterators.

```
List grades = new List();

Iterator gradeList = grades.iterator();

while ( gradeList.hasNext() ){
   listItem = gradeList.next();
   print ( listItem );
   }
```

Java Enumeration/Iterator

Methods		
Enumeration	**Iterator**	**ListIterator**
hasMoreElements()	hasNext()	hasNext()
nextElement()	next()	next()
	remove()	remove()
		nextIndex()
		hasPrevious()
		previous()
		previousIndex()
		add()
		set()

Iterators go through elements of a collection.

Iterator and ListIterator are fail-fast

If the underlying collection is changed (elements added or removed) by means other than the iterator, then the next time the iterator is accessed it will throw a java.util.ConcurrentModificationException

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 ) );
```

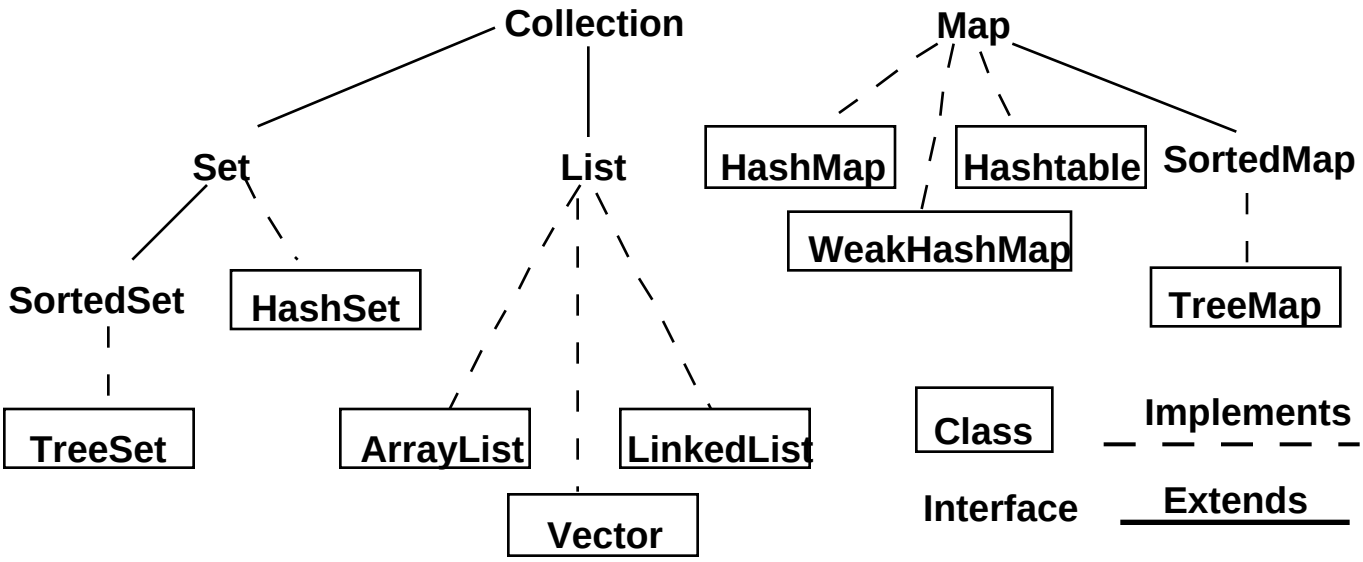
Binary Search Tree

```
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."

Object Cohesion

The degree to which components of a class are tied together

Evaluating cohesion requires:

- Technical knowledge of the application domain
- Some experience in building, modifying, maintaining, testing and managing applications in the appropriate domain
- Technical background in and experience with reusability

Questions to probe cohesiveness of an object

Does the object represent a complete and coherent concept or does it more closely resemble a partial concept, or a random collection of information?

Does the object directly correspond to a "real world entity," physical or logical?

Is the object characterized in very non-specific terms?

Collection of data, statistics, etc.

Do each of the methods in the public interface for the object perform a single coherent function?

If the object (or system of objects) is removed from the context of the immediate application, does it still represent a coherent and complete object-oriented concept?

For objects that are "system of objects"

Does the system represent an object-oriented concept?

Do all the objects directly support, or directly contribute to the support of, the object-oriented concept that the system represents?

Are there missing objects?

Objects in Isolation

Isolation means without considering any hierarchy that may contain the object or class

Does not discuss non-objects:

- Object with only functions
- Objects with only data

Individual Objects

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

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

A **sufficient set of primitive methods** for an object is a minimum set of primitive methods to accomplish all necessary work with on the object

A sufficient set of primitive methods has two major problems:

- Some tasks may be awkward and/or difficult with just a sufficient set of primitive methods

- A sufficient set of primitive methods may not allow us to fully capture the abstraction represented by the object

A **complete set of primitive methods** is a set of primitive methods that both allows us to easily work with the object, and fully captures the abstraction represented by the object.

An object is not as cohesive as it could be if the public interface contains:

- Only primitive methods, but does not fully capture the abstraction represented by the object
- Primitive and composite methods, but does not fully capture the abstraction represented by the object
- A sufficient set of primitive methods with composite methods
- No primitive methods, just composite methods

Note

- Objects with a sufficient set of primitive methods with composite methods is more cohesive than objects with out a sufficient set of primitive methods
- All public methods must directly support the abstraction represented by the object. The methods must make sense when object is removed from the application

Composite Objects

A **composite object** is an object that is conceptually composed of two, or more, other objects, which are externally discernable.

Component objects are those that make up the composite object.

Component objects are **externally discernable** if

- Component objects can be directly queried or changed via methods in the public interface of the composite object and/or
- The externally discernible state of the object is directly affected by the presence or absence of one or more component objects

Ranking of Cohesion of Composite Objects Increasing order of Goodness

- Externally discernible component objects not related
- Some externally discernible component objects are related, the group component objects does not make sense
- The group component objects does not represent a single stable object-oriented concept, but are all bound together some how in an application
- A majaroity of the externally discernible component objects support a single,coherent, object-oriented concept, but at least one does not
- All of the externally discernible component objects support a single,coherent, object-oriented concept, but at least one needed is missing
- All of the externally discernible component objects support a single,coherent, object-oriented concept, and none are missing

Accessing Cohesion of an Individual Object

Accessment of the public methods/public non-methods/component objects

Are all the items appropriate for the given object?

Do we have at least a minimally sufficient set of items?

Do we have extra or application-specific items?