# CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2022
Doc 15 Coupling
Oct 25, 2022

# 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

# KWIC Solutions

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

# 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

```
class Bar {
    ArrayList<People> employees;

    void addEmployees(Foo data) {
        int result = data.computeCompanySize();
        employees = new ArrayList<People>(result);
        employees.add(data.people());
    }

    Money averageSalary() {
        Foo moreData = new Foo();
        moreData.setStuff(employees);
        return moreData.averageSalary();
    }
```

# 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 );
```

# Attempt 1

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

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

||

# Attempt 1

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

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

# Attempt 2

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

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

```
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
        }
}
```

## Attempt 3

```
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 {
        Comparable[] sortedElements = new Object[ properSize ];

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

# Attempt 4

```
interface Comparing {
        public boolean lessThan( Object a, Object b );
        public boolean greaterThan( Object a, Object b );
        public boolean equal( Object a, Object b );
}

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


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

# Attempt 4

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

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


SortedList byName = new SortedList( new ByName() );
SortedList byID = new SortedList( new ById());
```

# Java 8 Solution

```
interface Comparator<T> { int compare(T o1, T o2) }


class SortedList<T> {
        T[] sortedElements = new T[ properSize ];
        Comparator<T> comparer;
        public SortedList(Comparator<T> y) {comparer = y;}

        public void add( Object X ) {
                // coded not shown
                if ( (comparer.compare( sortedElements[ K ], X ) < 0 )
                        // do something
                else
                        // do something else
                }
        }
```

# Java 8 Solution

```
SortedList byName = new SortedList( (a, b) -> a.lastName() < b.name());

SortedList byID = new SortedList( (a, b) -> a.id() < b.id());
```

# 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

# Interface Coupling

More flexible form of Data Coupling

List bar = new ArrayList()
Iterator foo = bar.iterator()

# 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

```
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

```
class BankAccount {
        public void withdrawal(Float amount) {
                balance = balance - amount;
        }
etc.
```

**Is this Control Coupling**

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

What if the Lamp had 50 settings?

## Control Coupling

### Common Object Occurrence

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

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

# Cure – Use Optionals

Use where a value may be absent

Replaces use of null & exceptions

An optional either
  Has a value
  Is nil

Converting String to Int

“123” ——— 123

“2cat” ——— ?

var sample: String?

Defines a string optional
sample is either nil or contains a string

# Cure – Use Optionals

```
let aString = "123"
let possibleInt:Int? = Int(aString)


if let theInt = possibleInt {
    let foo = theInt + 10
} else {
    print("It was not an Int")
}
```

```
func foo() throws -> String


func foo() -> String?
```

Swift

Can convert Exception into optional

# 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"

Make a reference to a specific external object

Make a reference to a specific external object, and to methods in the 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
- Python

# Java Police Verse Python

From Stack Over Flow

Python drops that pretense of security and encourages programmers to be responsible.
In practice, this works very nicely.

# 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

# Patterns that Reduce Coupling

Abstract Factory

Bridge

Chain of Responsibility

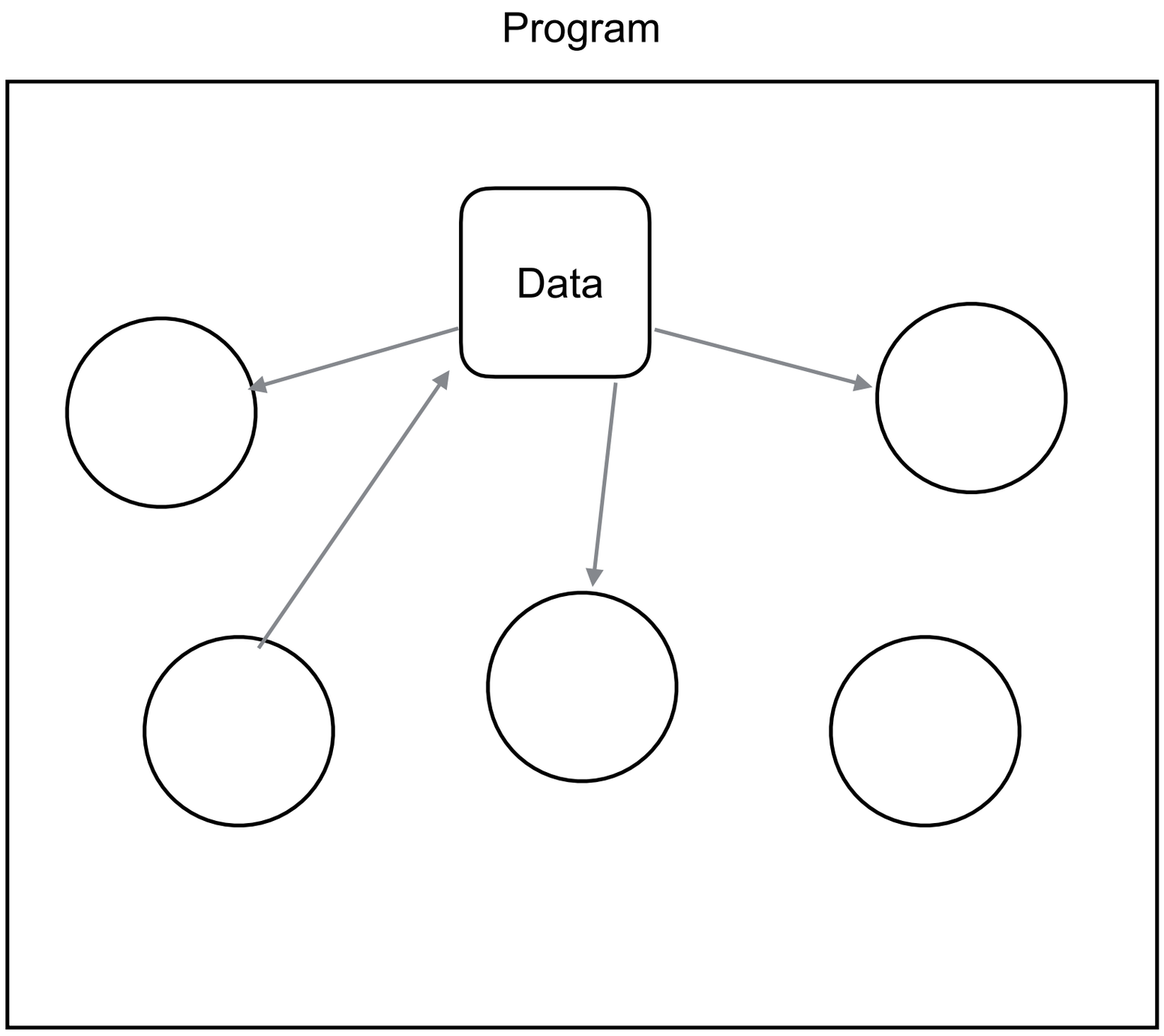
Command

Observer

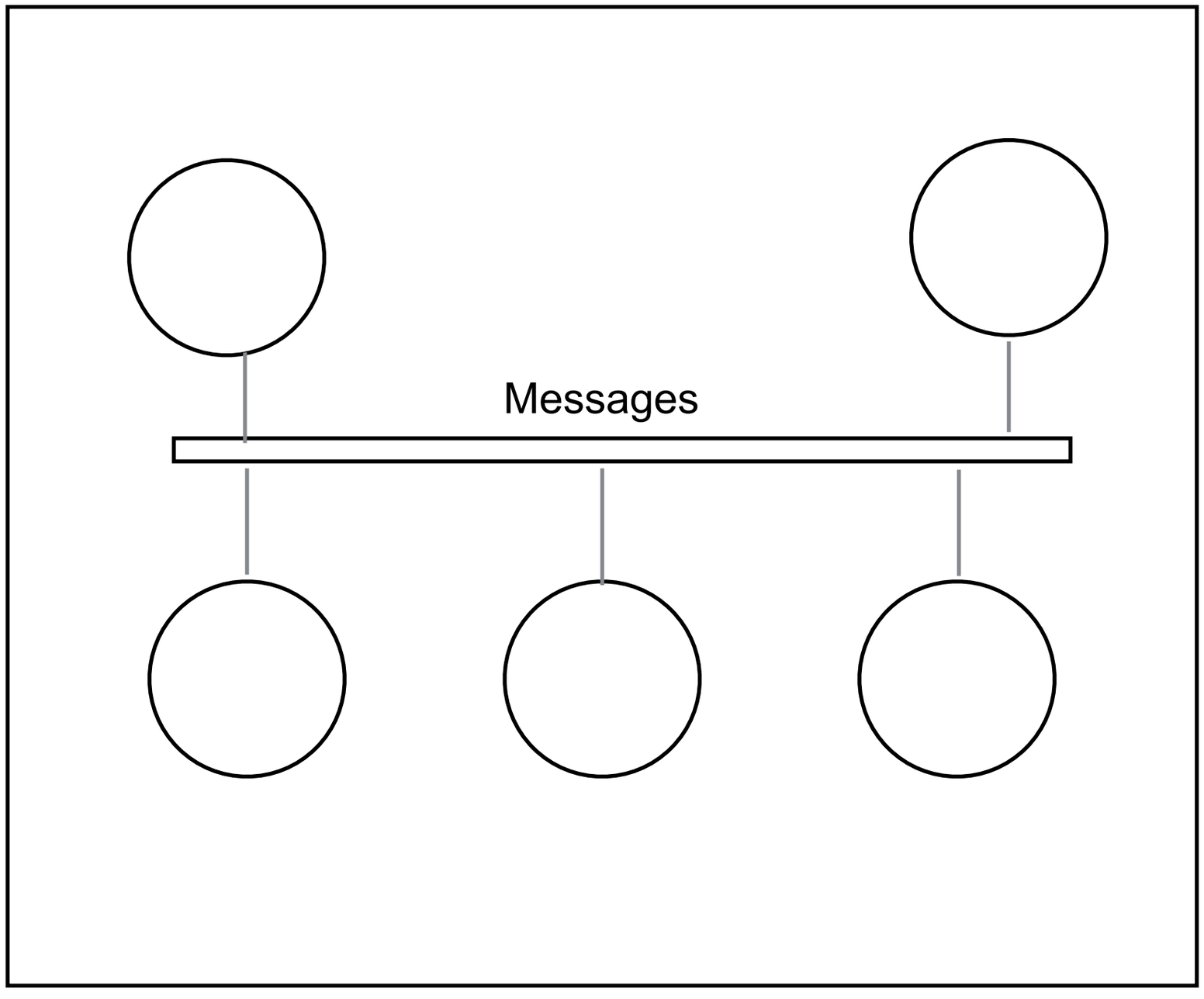
Facade

Mediator

# Reactive Programming & Single Source of Truth



# Events/Messages



# Distributed/Serverless