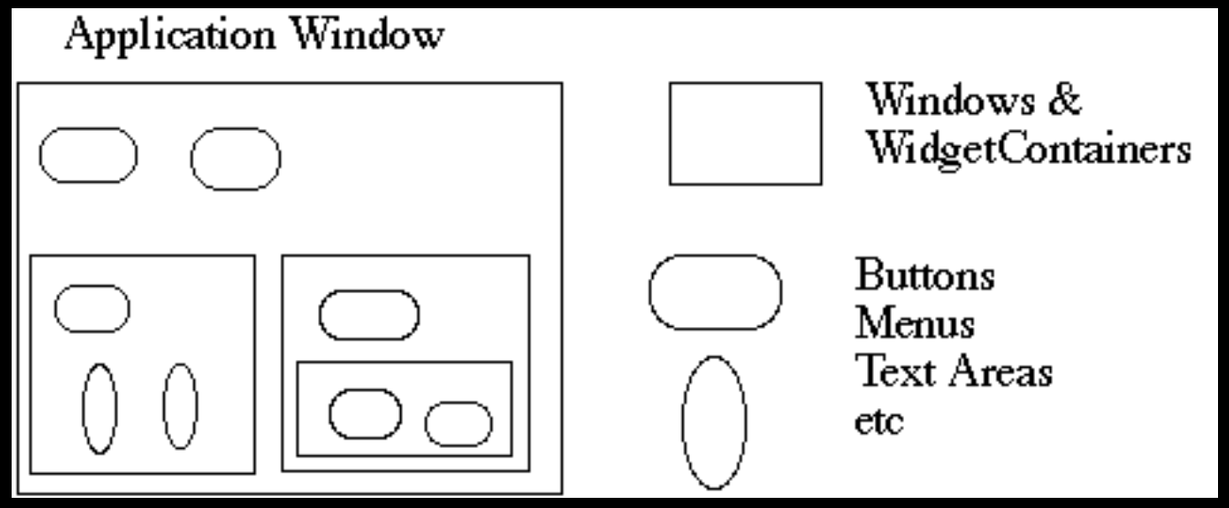
CS 635 Advanced Object-Oriented Design & Programming
Spring Semester, 2018
Doc 9 Composite, Pattern Intro, Coupling
Oct 2, 2018

Composite

Composite Motivation



How does the window hold and deal with the different items it has to manage?

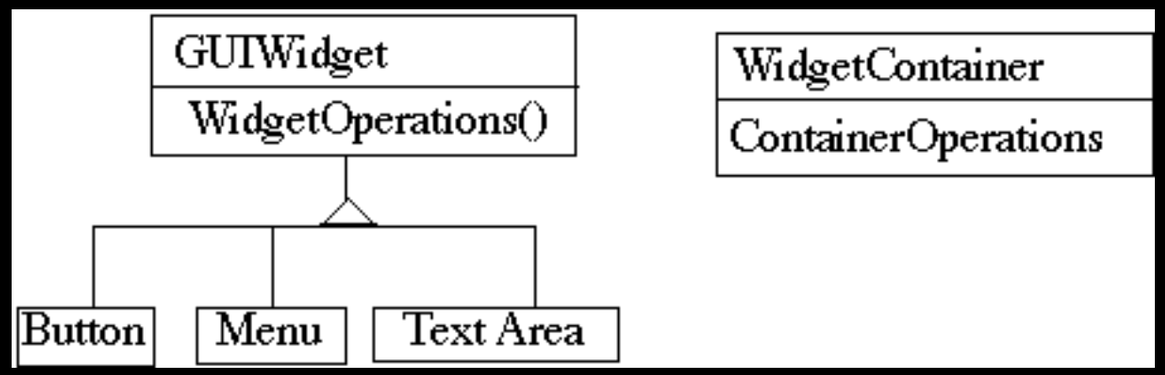
Widgets are different that WidgetContainers

Bad News

```
class Window {
    Buttons[] myButtons;
    Menus[] myMenus;
    TextAreas[] myTextAreas;
    WidgetContainer[] myContainers;

   public void update() {
      if ( myButtons != null )
          for ( int k = 0; k < myButtons.length(); k++ )
              myButtons[k].refresh();
      if ( myMenus != null )
          for ( int k = 0; k < myMenus.length(); k++ )
              myMenus[k].display();
      if ( myTextAreas != null )
          for ( int k = 0; k < myButtons.length(); k++ )
              myTextAreas[k].refresh();
      if ( myContainers != null )
          for ( int k = 0; k < myContainers.length(); k++ )
              myContainers[k].updateElements();
      etc.
}
public void fooOperation(){
    if (myButtons != null)
    etc.
```

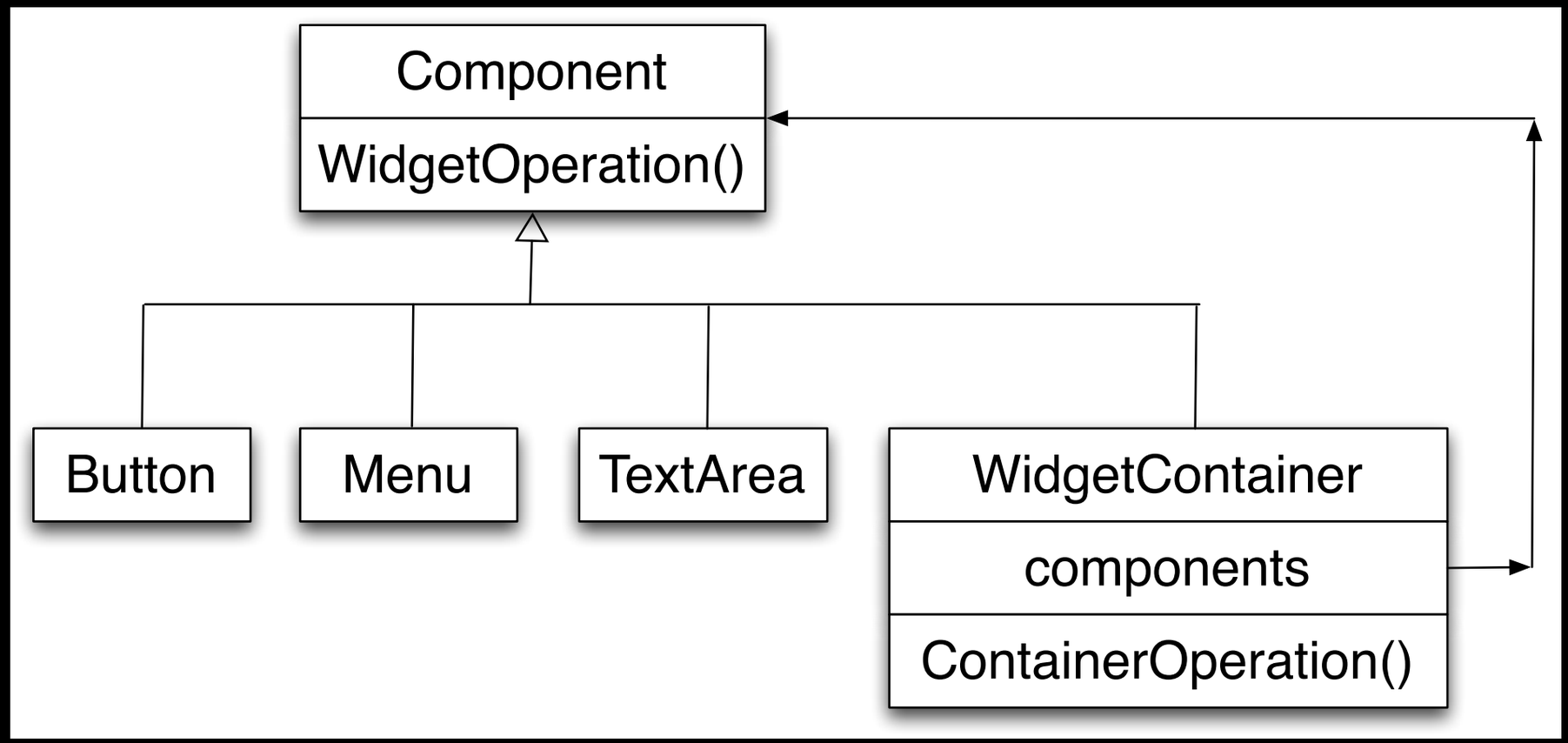
An Improvement




```
class Window {
    GUIWidgets[] myWidgets;
    WidgetContainer[] myContainers;

    public void update(){
      if ( myWidgets != null )
          for ( int k = 0; k < myWidgets.length(); k++ )
              myWidgets[k].update();
      if ( myContainers != null )
          for ( int k = 0; k < myContainers.length(); k++ )
              myContainers[k].updateElements();
      etc.
    }
}
```

Composite Pattern



Composite Pattern

Component implements default behavior for widgets when possible

Button, Menu, etc overrides Component methods when needed

WidgetContainer will have to overrides all widgetOperations

```
class WidgetContainer {
    Component[] myComponents;

    public void update() {
        if ( myComponents != null )
            for ( int k = 0; k < myComponents.length(); k++ )
                myComponents[k].update();
    }
}
```

Issue - WidgetContainer Operations

Should the WidgetContainer operations be declared in Component?

Pro - Transparency

Declaring them in the Component gives all subclasses the same interface

All subclasses can be treated alike. (?)

Con - Safety

Declaring them in WidgetContainer is safer

Adding or removing widgets to non-WidgetContainers is an error

One out is to check the type of the object before using a WidgetContainer operation

Issue - Parent References

```
class WidgetContainer
    {
    Component[] myComponents;

    public void update() {
        if ( myComponents != null )
            for ( int k = 0; k < myComponents.length(); k++ )
                myComponents[k].update();
    }

    public add( Component aComponent ) {
        myComponents.append( aComponent );
        aComponent.setParent( this );
    }
}

class Button extends Component {
    private Component parent;
    public void setParent( Component myParent) {
        parent = myParent;
    }

    etc.
```

More Issues

Should Component implement a list of Components?

The button etc. will have a useless data member

Child ordering is important in some cases

Who should delete components?

Applicability

Use Composite pattern when you want

To represent part-whole hierarchies of objects

Clients to be able to ignore the difference between compositions of objects and individual objects

Pattern Intro

Pattern Beginnings

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

"Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution"

A Pattern Language, Christopher Alexander, 1977

A Place To Wait

The process of waiting has inherent conflicts in it.

Waiting for doctor, airplane etc. requires spending time hanging around doing nothing

Cannot enjoy the time since you do not know when you must leave

Classic "waiting room"
Dreary little room
People staring at each other
Reading a few old magazines
Offers no solution

Fundamental problem
How to spend time "wholeheartedly" and
Still be on hand when doctor, airplane etc arrive

Fuse the waiting with other activity that keeps them in earshot
- Playground beside Pediatrics Clinic
- Horseshoe pit next to terrace where people waited

Allow the person to become still meditative
- A window seat that looks down on a street
- A protected seat in a garden
- A dark place and a glass of beer
- A private seat by a fish tank

A Place To Wait

Therefore:

"In places where people end up waiting create a situation which makes the waiting positive. Fuse the waiting with some other activity - newspaper, coffee, pool tables, horseshoes; something which draws people in who are not simple waiting. And also the opposite: make a place which can draw a person waiting into a reverie; quiet; a positive silence"

Chicken And Egg

Problem

Two concepts are each a prerequisite of the other
To understand A one must understand B
To understand B one must understand A
A "chicken and egg" situation

Constraints and Forces

First explain A then B
Everyone would be confused by the end
Simplify each concept to the point of incorrectness to explain the other one
People don't like being lied to

Solution

Explain A & B correctly by superficially

Iterate your explanations with more detail in each iteration

Patterns for Classroom Education, Dana Anthony, pp. 391-406, Pattern Languages of Program Design 2, Addison We
1996

Design Principle 1

Program to an interface, not an implementation

Use abstract classes (and/or interfaces in Java) to define common interfaces for a set of classes

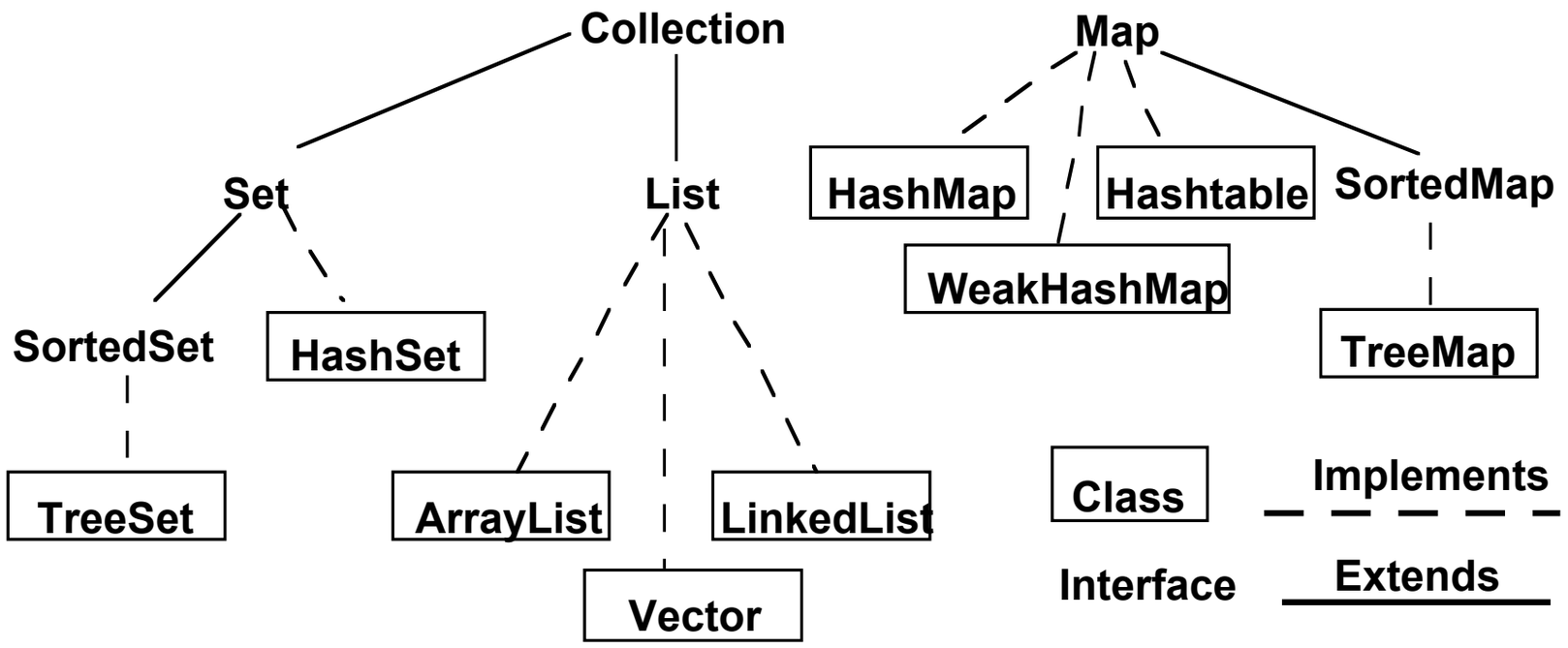
Declare variables to be instances of the abstract class not instances of particular classes

Benefits of programming to an interface

Client classes/objects remain unaware of the classes of objects they use,
as long as the objects adhere to the interface the client expects

Client classes/objects remain unaware of the classes that implement these objects.
Clients only know about the abstract classes (or interfaces) that define the interface.

Programming to an Interface




```
Collection students = new XXX;
students.add( aStudent);
```

students can be any collection type

We can change our mind on what type to use

Interface & Duck Typing

In dynamically typed languages programming to an interface is the norm

Dynamically typed languages tend to lack a way to declare an interface

Design Principle 2

Favor object composition over class inheritance

Composition
- Allows behavior changes at run time
- Helps keep classes encapsulated and focused on one task
- Reduce implementation dependencies

Inheritance

```
class A {
   Foo x
   public int complexOperation() {   blah }
}

class B extends A {
   public void bar() { blah}
}
```

Composition

```
class B {
   A myA;
   public int complexOperation() {
      return myA.complexOperation()
   }

   public void bar() { blah}
}
```

Designing for Change

Algorithmic dependencies
Builder, Iterator, Strategy,
Template Method, Visitor

Inability to alter classes conveniently
Adapter, Decorator, Visitor

Dependence on specific operations
Chain of Responsibility, Command

Dependence on hardware and software platforms
Abstract factory, Bridge

Tight Coupling
Abstract factory, Bridge, Chain of Responsibility,
Command, Facade, Mediator, Observer

Extending functionality by subclassing
Bridge, Chain of Responsibility, Composite,
Decorator, Observer, Strategy

Dependence on object representations or implementations
Abstract factory, Bridge, Memento, Proxy

Extending functionality by subclassing
Bridge, Chain of Responsibility, Composite,
Decorator, Observer, Strategy

Creating an object by specifying a class explicitly
Abstract factory, Factory Method, Prototype

Kent Beck's Rules for Good Style

One and only once

In a program written in good style, everything is said once and only once

Methods with the same logic
Objects with same methods
Systems with similar objects

rule is not satisfied

Lots of little Pieces

"Good code invariably has small methods and small objects"

Small pieces are needed to satisfy "once and only once"

Make sure you communicate the big picture or you get a mess

Rates of change

Don't put two rates of change together

An object should not have a field that changes every second & a field that change once a month

A collection should not have some elements that are added/removed every second and some that are add/removed once a month

An object should not have code that has to change for each piece of hardware and code that has to change for each operating system

Replacing Objects

Good style leads to easily replaceable objects

"When you can extend a system solely by adding new objects without modifying any existing objects, then you have a system that is flexible and cheap to maintain"

Moving Objects

"Another property of systems with good style is that their objects can be easily moved to new contexts"

Coupling

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

Types of Modular Coupling

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

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