CS 130 SOFTWARE ENGINEERING

INFORMATION HIDING:

DESIGN PRINCIPLE

Professor Miryung Kim

UCLA Computer Science

SOFTWARE DESIGN PRINCIPLES

- Information hiding principle
- Low coupling (dependencies) : reduce the dependencies (calls, using fields, using types declared in another package) between modules (classes, packages, etc)
- High cohesion: a single concept is represented is a single place. One class does not do multiple things.
- Separation of concerns: a single concern is easily separated from the rest of concerns.

THINK PAIR SHARE

- What is a module?
  - A self contained piece of code that does one job or several related jobs.
  - Parnas: an **independent work assignment** that can be assigned to a single engineer
- What is a good modularization?
  - Based on a set of assumptions on things that could change, **a good modularization would allow a localized update to a single module.**
  - **"Given alternative designs, which design would allow localized updates?"**

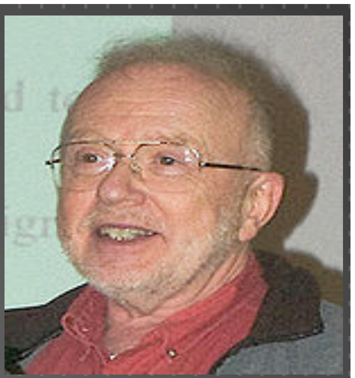
THINK PAIR SHARE

- What is a module?
  - Class in Java
  - Could it be a function or Java method?
  - A directory or a package
  - A library – a jar file, which gets unpacked into a set of packages with binary files.
- What is a good modularization? => This is a somewhat wrong question to ask. What we should be asking is "which design is a better modularization than the other?" =>" **easier to accommodate changes w.r.t. anticipated evolution scenarios"**
  - Compatibility with respect to other modules
  - Amenable to change (easy to change)
  - Easy to debug (comprehensibility)
  - Easy to test
  - Reduce shared resources

MODULARIZATION

- One technique proposed to improve software's **ease of change** is modularization
- A **module** is a work assignment; a **modular structure** is the **decomposition** of a program into modules for parallel work assignments for different teams

INFORMATION HIDING



- A principle for breaking a program into ***modules***
- "Design decisions that are likely to change independently should be secrets of separate modules."
- "The only assumptions that should appear in the ***interfaces*** between modules are those that are considered unlikely to change."

INFORMATION HIDING

- When information hiding is achieved, anticipated changes affect modules in an *isolated* and *independent* way
- Programmers must both identify what is likely to change and then ensure that interfaces between modules *do not reveal any volatile* information.

KWIC

- Input: an ordered set of lines where
  - each line is an ordered set of words
  - each word is an ordered set of characters
- Output: all circular shifts of all lines in alphabetical order

KWIC

- Input: an ordered set of lines where each line is an ordered set of words and each word is an ordered set of characters
  - *My name is Miryung Kim*
  - *Software Evolution*
- All circular shifts of all lines
  - *My name is Miryung Kim*
  - *name is Miryung Kim My*
  - *is Miryung Kim My name*
  - *Miryung Kim My name is*
  - *Kim My name is Miryung*
  - *Software Evolution*
  - *Evolution Software*

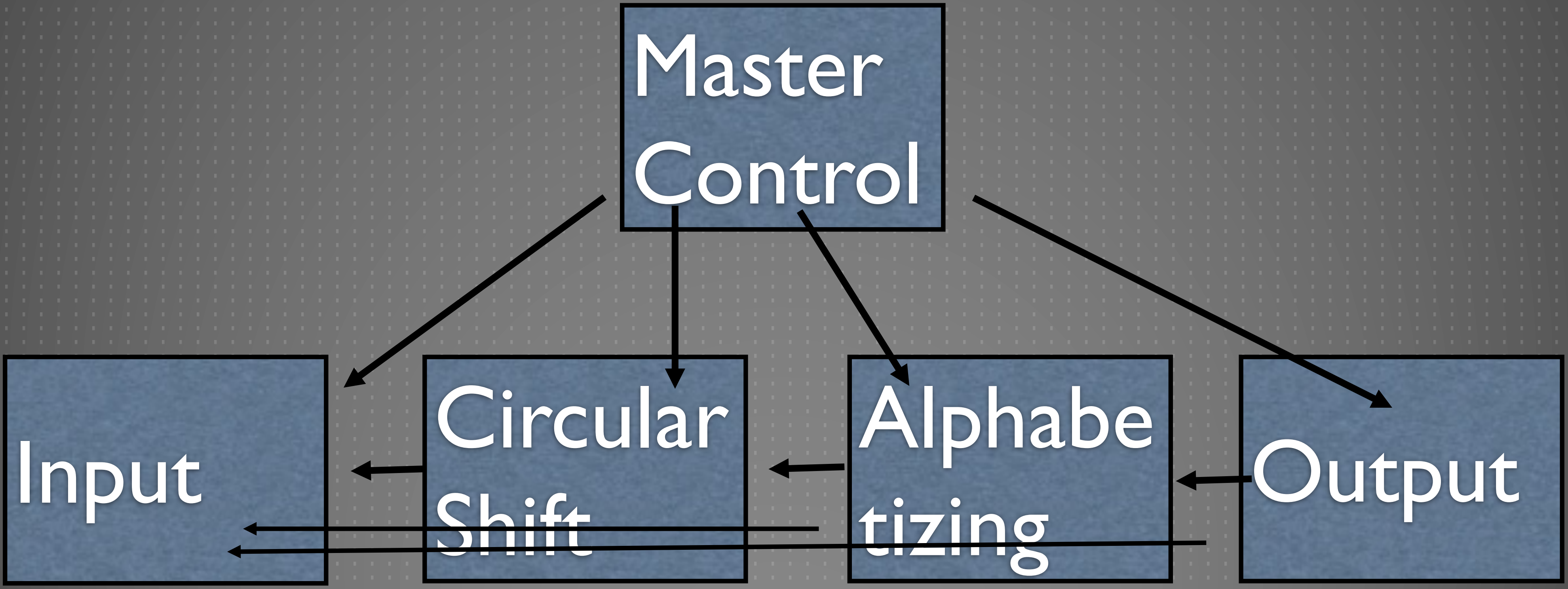
KWIC

- All circular shifts of all lines in alphabetical order
  - *Evolution Software*
  - *Kim My name is Miryung*
  - *Miryung Kim My name is*
  - *My name is Miryung Kim*
  - *Software Evolution*
  - *is Miryung Kim My name*
  - *name is Miryung Kim My*

- **Functional decomposition** (Flowchart approach)
  - Each module corresponds to each step in a flow chart.
  - Data representation is shared knowledge
- **Information Hiding**
  - Each module corresponds to a design decision that is likely to change and that must be hidden from other modules.
  - Interfaces definitions were chosen to reveal as little as possible.

- KWIC Modularization 1: Functional decomposition
- KWIC Modularization 2: Decomposition based on the information hiding principle

MODULARIZATION 1



Module Name		Input	Output
Input	reads in lines and stores them in memory	A set of lines	array of characters (chars) and array that stores the starting address of each line (line-index)
CircularShift	creates circular shifts	arrays produced by Input	two-dimensional array, each column of the array stores the address of the first character in the shift and the original index of the line in Input's line-index array
Alphabetizing	alphabetizes the circular shifts	arrays produced by Input and CircularShift	two-dimensional array (similar to CircularShift's output, except in alphabetical order)
Output	prints the alphabetized circular shifts	arrays produced by Input andAlphabetizing	printout of lines
MasterControl	passes control to the other four modules	lines of input	printout of lines

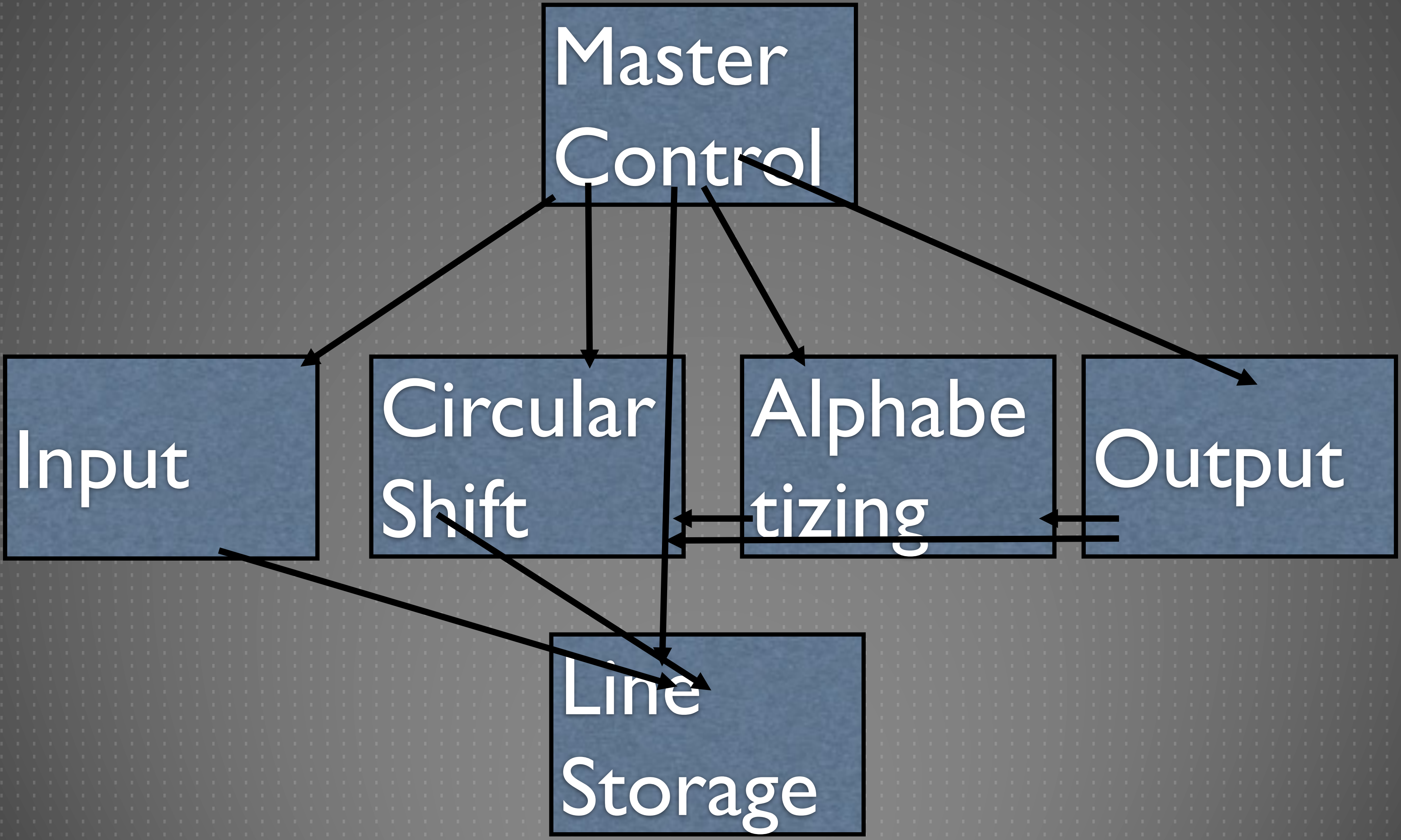
THINK PAIR SHARE

- What do you not like about this ”functional decomposition” design?
- **I am uncomfortable with exposing all internal content of arrays indices, etc. (Bingo! – violation of information hiding)**
- When MainController calls an CircularShift, the results of calling execute should be returned back and stored in MainController, but Alphabetizing directly invokes CircularShift to get the results (the issue of call sequences is not the main issues)
- Idempotency and “pure function” the function call has a side effect that is remained in the module. (function is pure or not)

- If we created a method whose return type is Integer[] instead of int[], is it better?
  - Using primitive types vs. object types is a red herring
- Using a static call instead of constructor followed by a dynamic call, is also not a true issue.
- Using a public getter after private field declaration does not make this code any better.

- Poor API choices:
  - If you change the internal data structure type from int [] to any thing, of course the clients need to change their code because it does not compile.
  - Internal data structure choices/ underlying data layout details are directly revealed to clients and clients must understand the layout of internal data to work on their implementation.

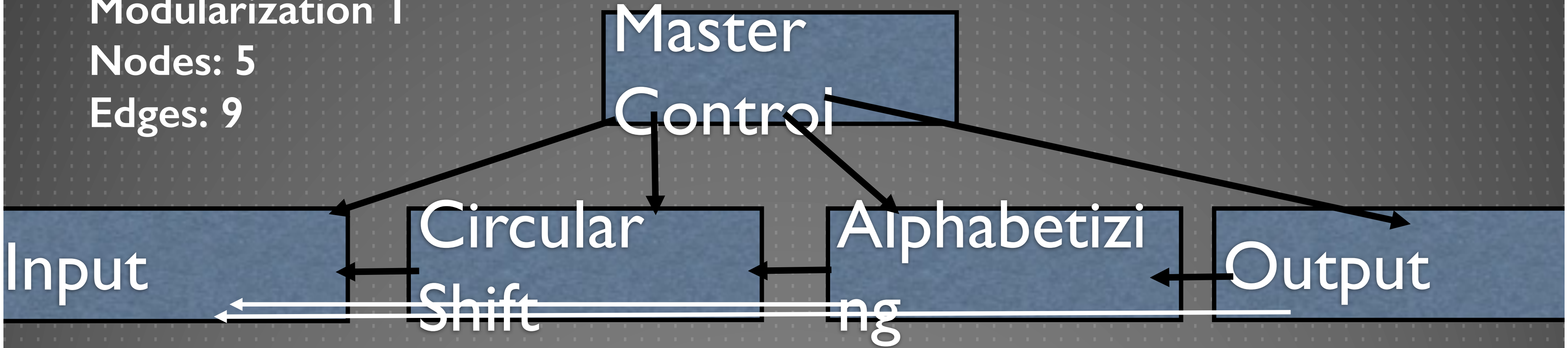
MODULARIZATION 2



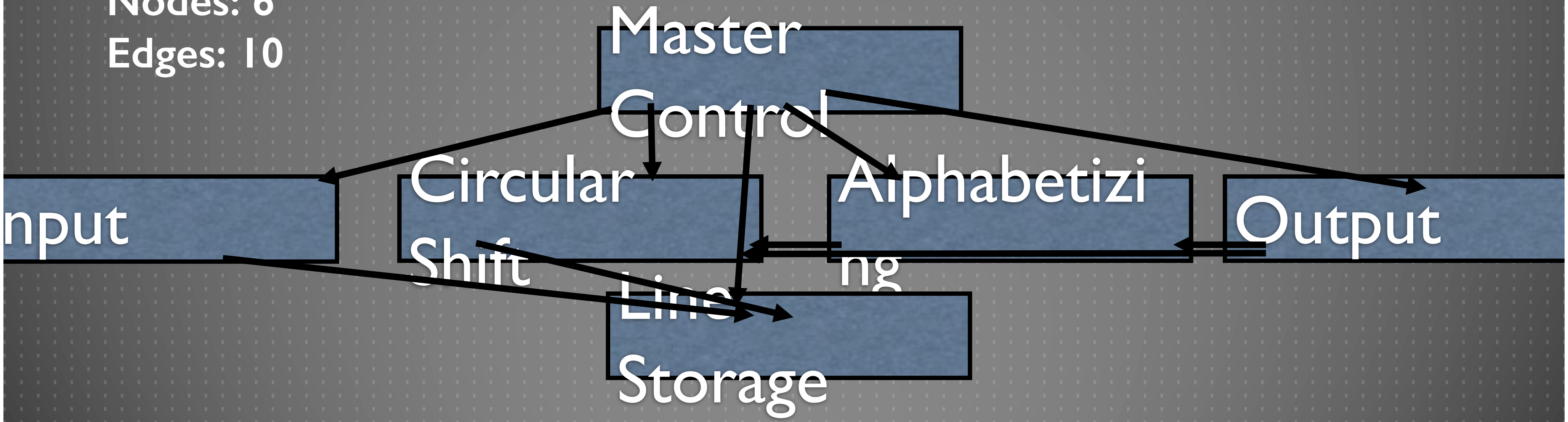
Module name	Role	What it calls	What it offers clients
LineStorage	hides the exact representation used to store lines	none	numerous methods for accessing and setting characters, words, and lines
Input	read input	LineStorage methods	none
CircularShifter	create impression that a list of all circular shifts exists	LineStorage methods	init method (setup) and methods to access characters, words, and lines of circular shifts (e.g., getChar)
Alphabetizer	provide clients with a means to access alphabetized shifts	CircularShifter methods	init method (alpha) and a method that offers an alphabetized ordering for circular shifts (ith)
Output	prints the alphabetized circular shifts	CircularShift and Alphabetizer methods	printout of lines
MasterControl	passes control to Input, CircularShifter, Alphabetizer, and Output	Input, Output, CircularShifter, and Alphabetizer methods	printout of lines

COMPARISON ?

Modularization 1
Nodes: 5
Edges: 9



Modularization 2
Nodes: 6
Edges: 10



- Both are decompositions.
- Both share data representations and access methods
- Is the modularization #1 bad? Why?

- For both designs, MasterControl calls different modules in a certain order. When they are called, certain information is passed around.
- Difference: What kind of information was shared?
  - 1. Information about internal data layout was shared. (all secrets were revealed.)
  - 2. Information about data was shared (line, word, etc), not every secrets were revealed.
- If you added LineStorage, and if it had a single getLine() method that returns char[][], it is as worse as before.

QUESTIONS FROM AUDIENCE DURING THE BREAK

- Couldn’t you just add a new class LineStorage to the first decomposition and that becomes the information hiding based decomposition, right?
  - Yes, the answer is YES.
- So then what is the true difference between functional decomposition and information hiding?
  - Recognize the needs of “LineStorage” and making it as a separate module?
- Information Hiding is not about getters, setters and use of private fields.
  - “Information to be hidden from other modules” is the design decisions that are likely to change.

THINK-PAIR-SHARE: CHANGE SCENARIOS

- What kinds of changes can you anticipate regarding the KWIC?

CHANGE SCENARIOS

InputFormat	Input format is changed
A Single Storage	The decision to have all lines stored in core. For large jobs, it may prove inconvenient or impractical to keep all of the lines in core at any one time
Packing characters	The decision to pack the characters four to a word. In cases where we are working with small amounts of data, it may prove undesirable to pack the characters.
Index for CS	The decision to make an index for the circular shifts rather than actually store them as such.
Search or Partial Alphabetize	The decision to alphabetize the list once, rather than either search for each item when needed or partially alphabetize them

CHANGEABILITY ASSESSMENT: MODULARIZATION 1

Changes	MasterControl	Input	CircularShift	Alphabetizer	Output
InputFormat		✔			
A Single Storage	✔	✔	✔	✔	✔
Packing characters	✔	✔	✔	✔	✔
Index for CS			✔	✔	✔
Search or Partial Alphabetize				✔	✔

CHANGEABILITY ASSESSMENT: MODULARIZATION 2

Changes	MasterControl	Input	CircularShift	Alphabetizer	Output	LineStorage
InputFormat		✔				
A Single Storage						✔
Packing characters						✔
Index for CS			✔			
Search or Partial Alphabetixe				✔		

Modularization 1:Modularization 2:

Changes	Master Control	Input	CircularShift	Alphabetizer	Output
InputFormat		✓			
A Single Storage	✓	✓	✓	✓	✓
Packing characters	✓	✓	✓	✓	✓
Index for CS			✓	✓	✓
Search or Partial Alphabetixe				✓	✓

Changes	MasterControl	Input	CircularShift	Alphabetizer	Output	LineStorage
InputFormat		✓				
A Single Storage						✓
Packing characters						✓
Index for CS			✓			
Search or Partial Alphabetixe				✓		

- Modularization 1: The decision to store line indices and word indices must be communicated among all module developers
- Modularization 2: API names and types stay unchanged. Only the internal implementation of APIs are affected.

- Functional decomposition (Flowchart approach)
  - Each module corresponds to each step in a flow chart.
- Information Hiding
  - Each module corresponds to a design decision that are likely to change and that must be hidden from other modules.
  - Interfaces definitions were chosen to reveal as little as possible.

CLASS ACTIVITY: DESIGN REVIEWS

- Study the source code, MortgageCalculator
- Critique the design with respect to its use of information hiding.

```java
public class MortgageCalculator {
    double payment, principal = 200000;
       // Principle amount of loan is $200,000
       double annualInterest = 0.0575;
       // Interest rate is currently 5.75%
       int years = 30; //*Term of the loan is 30 years

       public static void main (String[] args){
          MortgageCalculator calculator = new MortgageCalculator();
           if (args.length == 3) {

                 double principal = Double.parseDouble(args[0]);
                 double annualInterest = Double.parseDouble(args[1])
                 int years = Integer.parseInt(args[2]);
                 calculator.principal= principal;
                 calculator.annualInterest= annualInterest;
                 calculator.years= years;
                 calculator.print(principal, annualInterest, years);
              }
           }
```

```
        public static double calculatePayment(double principal, 
double annRate, int years){
            double monthlyInt = annRate / 12;
            double monthlyPayment = (principal * monthlyInt)
                        / (1 - Math.pow(1/ (1 + monthlyInt), years * 
12));
            //Shows 1 monthly payment multiplied by 12 to make one 
complete year.
            return format(monthlyPayment, 2);
        }
        public static double format(double amount, int
mortgage) {
                double temp = amount;
                temp = temp * Math.pow(10, mortgage);
                temp = Math.round(temp);
                temp = temp/Math.pow(10, mortgage);
                return temp;
        }
        public void print(double pr, double annRate, int
years){
                double mpayment = calculatePayment(pr, annRate,
years);
                System.out.println("The principal is $" +
(int)pr);
                //Shows the principle amount in $ value.
                System.out.println("The annual interest rate is
" + format(annRate * 100, 2) +"%");
                System.out.println("The term is " + years + "
years");
                //Term is normally in years.
                System.out.println("Your monthly payment is $"
+ mpayment);
                //Shows output of monthly payment.
        }
```

DISCUSSION QUESTION 1.

- What kinds of secrets are hidden by MortgageCalculator?
  - calculatePayment(principal, annRate, years) =>double
    - How you compute payment. E.g., **monthly compounding** cs. Daily compounding

DISCUSSION QUESTION 2

- What kinds of changes can you anticipate? List any new features that you may want to add.

DISCUSSION QUESTION 3

- Critique the current code in terms of readability and comprehensibility.

DISCUSSION QUESTION 4

- Critique the current code in terms of capability to support independent work assignment.

CHANGE SCENARIOS

	payment, annualInterest, years	calculatePayment	format	print
Bi-weekly payment	Add a variable payment option	V		V
Daily compounding of interest	Add an option	V		
Ignore cents		V		

ACTIVITY AT HOME: IMPROVING KWIC DESIGN

- Step 1. Identify two unanticipated evolution scenarios for KWIC, which were not discussed the class today
- Step 2. Discuss how the KWIC modularization 1 and 2 will be impacted to accommodate the evolution scenario

REVIEW QUESTION 1

- What is the definition of a “module”?
- What is the role of public *interface* in software design?
- What is the benefit of software modularization?

REVIEW QUESTION 2

- Critique the following software design using Parnas' information hiding principle.

```
class LegacyLine {
    public void draw(int x1, int y1, int x2, int y2) {
        System.out.println("line from (" + x1 + ',' + y1 + ")
to ("
     + x2 + ',' + y2 + ')');
    }
}
class LegacyRectangle {
    public void draw(int x, int y, int w, int h) {
        System.out.println("rectangle at (" + x + ',' + y + ")
with width "
     + w + " and height " + h);
    }
}
```

```java
public class Demo {
    public static void main(String[] args) {
        Object[] shapes = { new LegacyLine(), new
LegacyRectangle() };
        int x1 = 10, y1 = 20, x2 = 30, y2 = 60;
        for (int i = 0; i < shapes.length; ++i) {
          if
(shapes[i].getClass().getName().equals("LegacyLine"))
            (LegacyLine)shapes[i].draw(x1, y1, x2, y2);
          else if
(shapes[i].getClass().getName().equals("LegacyRectangle"))
            (LegacyRectangle)shapes[i].draw(Math.min(x1, x2),
        Math.min(y1, y2), Math.abs(x2 - x1), Math.abs(y2 -
y1));
        }
    }
}
```

REVIEW QUESTION 3

- Discuss the advantage of the following code using the IH principle.

```
interface Shape {
  void draw(int x1, int y1, int x2, int y2);
}

class Line implements Shape {
    private LegacyLine ll = new LegacyLine();
    public void draw(int x1, int y1, int x2, int y2) {
        ll.draw(x1, y1, x2, y2);
    }
}

class Rectangle implements Shape {
    private LegacyRectangle lr = new LegacyRectangle();
    public void draw(int x1, int y1, int x2, int y2) {
        lr.draw(Math.min(x1, x2), Math.min(y1, y2),
     Math.abs(x2 - x1), Math.abs(y2 - y1));
    }
}
```

```
public class Demo {
    public static void main(String[] args) {
        ArrayList<Shape> shapes = new ArrayList<Shape>();
        shapes.add(new Line());
        shapes.add(new Rectangle());

        int x1 = 10, y1 = 20, x2 = 30, y2 = 60;
        for (Shape s : shapes)
          s.draw(x1, y1, x2, y2);
    }
}
```

RECAP

- Information hiding principle is an analysis of how changes will affect existing code and assessment of changeability.

PREVIEW

- Read Head First Design Patterns Chapter 1-3

QUESTIONS?