

Object Oriented Programming Design Basics

CMSC 202

Topics

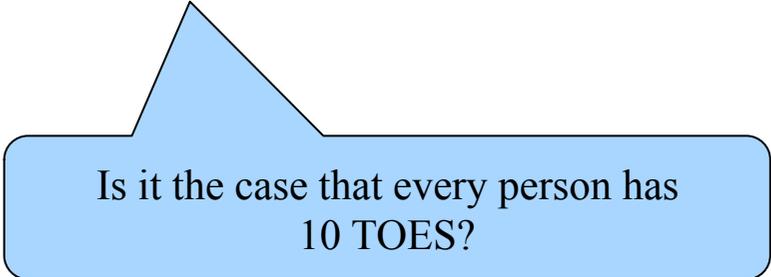
- Static Methods and Members
 - Appropriate Uses
 - Eclipse Debugging
- Encapsulation
 - Misuse of Accessors and Mutators
 - Immutable Objects and Constructors
- Composition
 - Fluent Interfaces
 - Method Chaining
 - Coupling
 - Delegation

Static Variables

- Remember, Static variables belong to the class.
 - All instances of the class have the static variable as their own.
 - However, all instances share the same static variable.
 - Can we use static variables to ensure all instance of a class will have the same attribute value?
 - Can we use static variables to represent general/universal attributes (i.e. Every person has 10 toes, 10 fingers,...)

Number of Toes

```
public class Person {  
    private static int numToes;  
  
    public Person() {  
        numToes = 10;  
    }  
  
    public loseToe() {  
        numToes--;  
    }  
  
    public static void main(String[] args) {  
        Person ted = new Person();  
        Person lilly = new Person();  
        Person peg = new Person();  
        peg.loseToe();  
    }  
}
```



Is it the case that every person has
10 TOES?

- Declaring numToes to be a static variable
 - What happens when peg, lilly, or ted were to lose a toe?
Everyone loses a toe.
- Generally:
 - Attributes are local to an instance, even when they appear to be true about an entire set of instance.
 - Static variables are commonly used to count instances of a class and control the state of all instances.

Static Constants

- Magic Numbers
 - "are a special constant used for some specific purpose",
 - whose value or presence is inexplicable without some additional knowledge of the implementation,
 - and commonly mislabeled **public**.
 - Implementation details should be hidden from a class' users.

```
private int[] grades = new int[5];
private static final int A = 0;
private static final int B = 1;

// magic numbers as array indexes
grades[0] = 5; // number of A in the class
grades[1] = 2; // number of C in the class

grades[A] = 5;
grades[B] = 2;
```

Static Methods

- Static methods:
 - are members of all instances of the class,
 - are mostly used as utility functions,
 - do not require a calling object,
 - can only use/call other static members/methods,
 - are most commonly debugged incorrectly.

Mistaken Static Methods

```
public class Examples {  
    public int getVariable(){  
        return variable;  
    }  
  
    private int variable = 0;  
  
    public static void main(String[] args){  
        getVariable();  
    }  
}
```

Eclipse suggests a solution
to change the modifier to
'static'

Changing the method
to static will not
still not solve the problem.
Eclipse will then suggest
variable be changed to static.
Why is this the case?

```
public class Examples {  
    public int getVariable(){  
        return variable;  
    }  
  
    private int variable = 0;  
  
    public static void main(String[] args){  
        getVariable();  
    }  
}
```

- Change modifier of 'getVariable()' to 'static'
- Assign statement to new local variable (Ctrl+2, L)
- Assign statement to new field (Ctrl+2, F)
- Rename in file (Ctrl+2, R)
- Rename in workspace (Alt+Shift+R)

```
... public static int getVariable(){  
    return variable;  
    ...  
}
```

Encapsulation

- We defined ***Encapsulation*** as a means to hide the implementation details of a class from the class user.
 - Visibility modifiers allow class creators to hide variables and functions.
- We often add accessors/mutators to classes providing users a means to access/modify the state of an object.
 - We even go on to say that this is still encapsulation, because we decide what services are provided and how they must be used.
 - Why not show it all?

Procedural or OO?

```
public class Car {
    // other instance variables...
    private int longitude = 0;
    private int latitude = 0;

    public int getLongitude(){ return longitude; }
    public int getLatitude(){ return latitude; }
    public void setLongitude(int long) { longitude = long; }
    public void setLatitude(int lat){ latitude = lat; }

    public static void main(String[] args){
        Car car = new Car("ABC123",1967,"Ford", "Mustang");
        // set initial location
        car.setLatitude(39);
        car.setLongitude(-76);
        System.out.println("Latitude: " + car.getLatitude() +
                           " Longitude: " + car.getLongitude());

        // drive to CCBC
        car.setLatitude(39);
        car.setLongitude(-75);
        System.out.println("Latitude: " + car.getLatitude() +
                           " Longitude: " + car.getLongitude());
    }
}
```

Encapsulation Problems

- When new OO programmers create an initial class, many times the class' interface is completely overlooked.
 - “I need to get this thing to work!”
 - We have all said this at some point, leading to poorly written OO code.
 - Let us think for a moment on how a Car actually changes *its* location.
 - A car typically has to be driven to another location. It does not instantaneous appear at its destination.
 - Why did we provide a public mutator to set the Latitude/Longitude?
 - Set the initial location of the Car? A constructor provides this functionality.
 - A public method drive(new latitude, new longitude) would suffice instead of providing two mutators.

The Public Interface

- A class' public interface is the set of all public methods and variables.
 - Let us look at two implementations of a similar behavior.

```
private boolean doorLock = false;

public void lockDoors() { doorLock = true; }
public void unlockDoors() { doorLock = false; }
```

— or —

```
private boolean doorLock = false;

public void setDoorLock(boolean lock) { doorLock = lock; }
```

- The first example provides two methods to simulate two behaviors of a Car door locking.
- The second uses a traditional mutator to accomplish both.
 - Is one better than the other?

Encapsulation: Things to Avoid

- Providing total access/control to all instance variables
 - Provide a well documented interface that allows users to modify the state indirectly (i.e. through class services)
 - Provide constructors that allow a user to initially set the state

```
public Car(String vin, ..., int latitude, int longitude){ ... }
```
 - Provide users accessors to a few variables they may require
 - Provide users services that act as mutators

```
public void moveCar(int latitude, int longitude){ ... }
```

Mutate or Garbage?

- Immutable Objects are objects whose state cannot change after it is constructed.
- How can we mutate an Immutable Object?
 - Create a new one.
 - The impact of object creation is often overestimated, and can be offset by some of the efficiencies associated with immutable objects. These include decreased overhead due to garbage collection, and the elimination of code needed to protect mutable objects from corruption.
- Again, not every class needs mutators to modify the state of an object.
 - It is our job to limit any class' public interface to the necessary variables and methods.

Fluent Interfaces

- A *Fluent Interface* is a way of implementing an OO API in a way that is more readable and reusable.
 - It is implemented by using *method chaining* to relay the instruction context of a subsequent call.
 - A phrase coined by Eric Evans and Martin Fowler

Method Chaining

- Method Chaining is when you invoke a method of a class and then invoke another method on the returned object and so on...

```
object.doSomething().doSomethingElse().doAnotherThing();
```

- This provides a more fluid feel while coding.
 - Suppose we want to get the first 3 characters of a Car's vin.

```
defaultCar.getVin().substring(0,3);
```

There are a few things you should know...

Coupling

- Coupling is the degree to which software modules rely upon one another.
 - If module A is coupled to module B, module A has a **dependency** on module B.
 - For example, class A is defined as such...
 - A is coupled to B,
 - A depends on B,
 - and B changes, A may need to change.

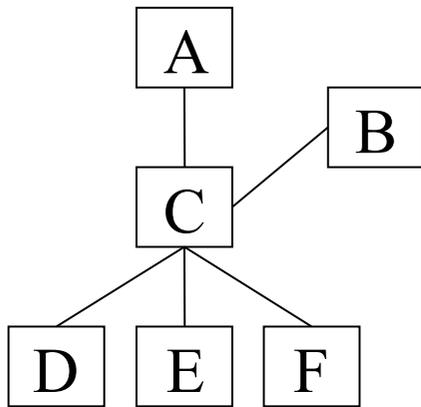
```
public class A {  
    private B b;  
    ...  
}
```

Class
A

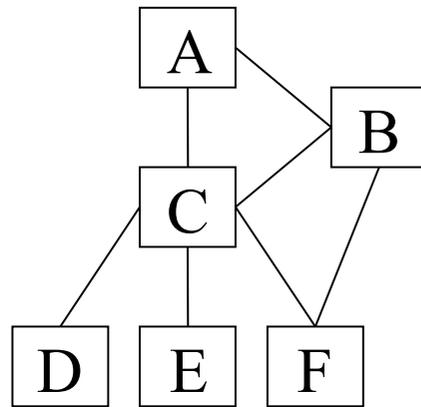
Class
B

More Coupling

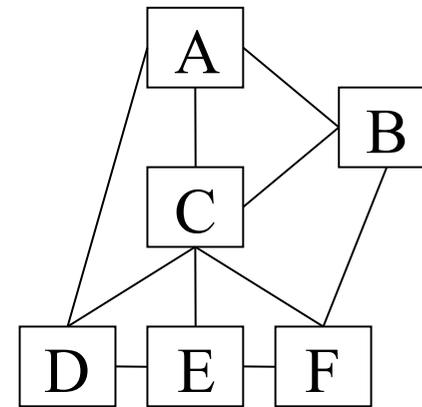
- Coupling strength is based on the
 - Quantity of module coupling points,
 - And the complexity of the coupling points.



Low coupling



Higher coupling



Very high coupling

For each example above, what are the concerns when the module C must be modified or replaced?

Evaluating Coupling

A Method

```
public void someMethod(int flag) {  
    ...  
  
    if (flag == 1)  
        ...  
    else if (flag == 2)  
        ...  
    else if (flag == 3)  
        ...  
    else ...  
  
    ...  
}
```

- This code is very strongly coupled to any other internal or external code that calls it.
- Yes, there is only one coupling point between the caller and someMethod (). But it is very strong (complex).
- The calling code must be aware of the meaning of all flag values.
- someMethod must be careful if it adds flag values, deletes flag values, or changes the meaning of any flag values.

Relative to coupling, what's a better way to implement this code?

More Coupling

- Want *weak (loose) coupling*
- Cannot have zero coupling, so our goals are to
 - minimize coupling,
 - weaken (loosen) coupling, and
 - most importantly, to control coupling.
 - Every coupling point is intentional.
 - Every coupling point has a well-defined interface.

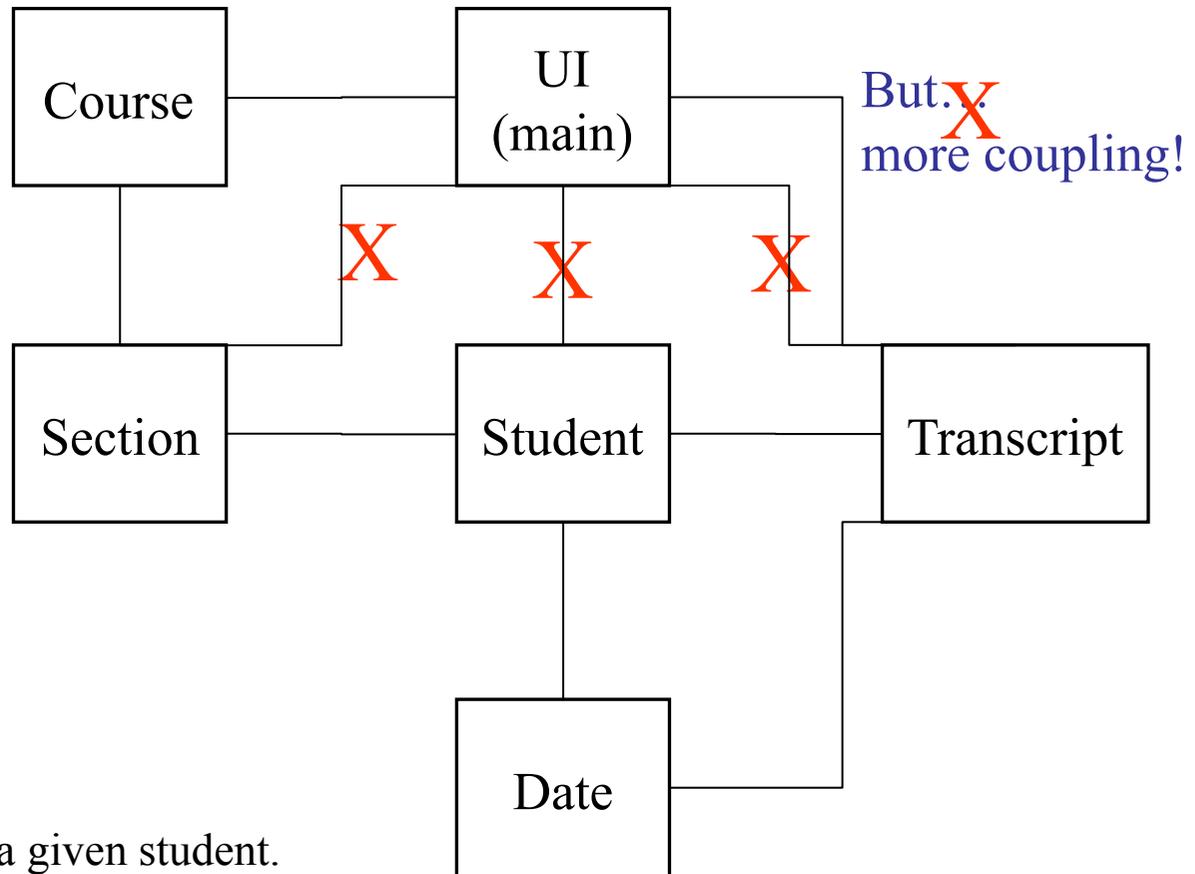
So, What's a Module?

- Any “chunk” of a software system or program
 - program
 - function
 - method
 - class
 - package
 - cluster of functions
 - cluster of classes
 - other ...
- Coupling must be controlled *at every level* of a software system or program.

Decoupling Method Chains

Problem: Create a program to track students registered for all sections of a course.

How do we remove the chains?



Suppose we want the GPA for a given student.
From the UI class, we could:

```
course.XgetSection(sectionId).getStudent.X(studentId).getTranscript().getGPA()
```

```
course.getStudentGPA(studentID)
```

Delegation

- Composition is known as classes using other classes to reuse code. Effective composition relies heavily on Delegation.
- Where should the work be done if not here? (aka The Fluent Interface approach to solving any problem)
 - OO design gurus believe that a class should do the work of its type.
 - A section is made up of students, so let section delegate the work of finding a student's GPA to the student class.
 - The String class handles the work of String objects. All users of Strings should delegate the String work to the String class.
 - Each class should delegate the work it is trying to do to the class that knows how to do it.

Final Thoughts

- A well designed interface solves a great many problems.
 - A little time now saves a larger amount of time later.