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 instances 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 now 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 instances.
  - 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();
    }

}
```

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

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

```
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;
...
```

Press 'Tab' from proposal table or click for focus

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.setLatitude(-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 ***<u>its</u>*** location.
    - A car typically has to be driven to another location. It does not instantaneously appear at its destination.
    - Why did we provide public mutators to set the latitude and longitude?
      - Set the initial location of the car? A constructor provides this functionality.
      - A public method

```
drive(newLatitude, newLongitude)
```

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 set the object's initial state.

```
public Car(String vin, ..., int latitude, int longitude){ ... }
```

    - Provide users accessors to a few attributes they may require.
    - Provide users services that act as mutators.

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

Mutate or Garbage?

- An ***immutable object*** is an object 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.

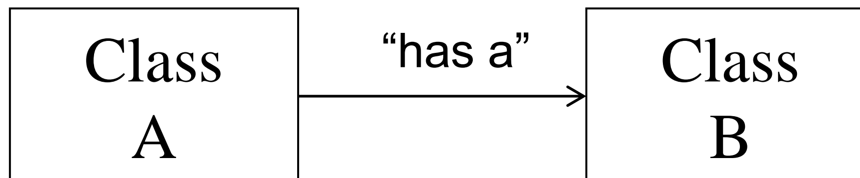
```
defaultCar.getVin().substring(0,3);
```

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 if B changes, A may need to change.

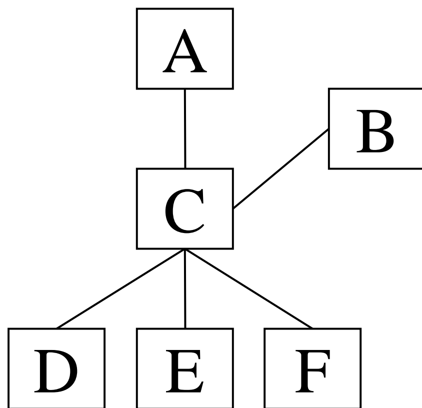
```
public class A {
    private B b;
    ...
}
```



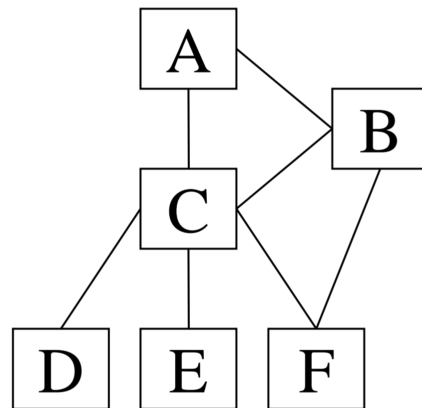
Question:
Is B coupled to A?

More Coupling

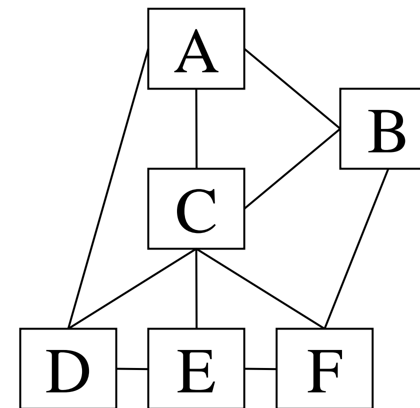
- ***Coupling strength*** is based on the
  - quantity of module ***coupling points***, and
  - the complexity of the coupling points.



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.

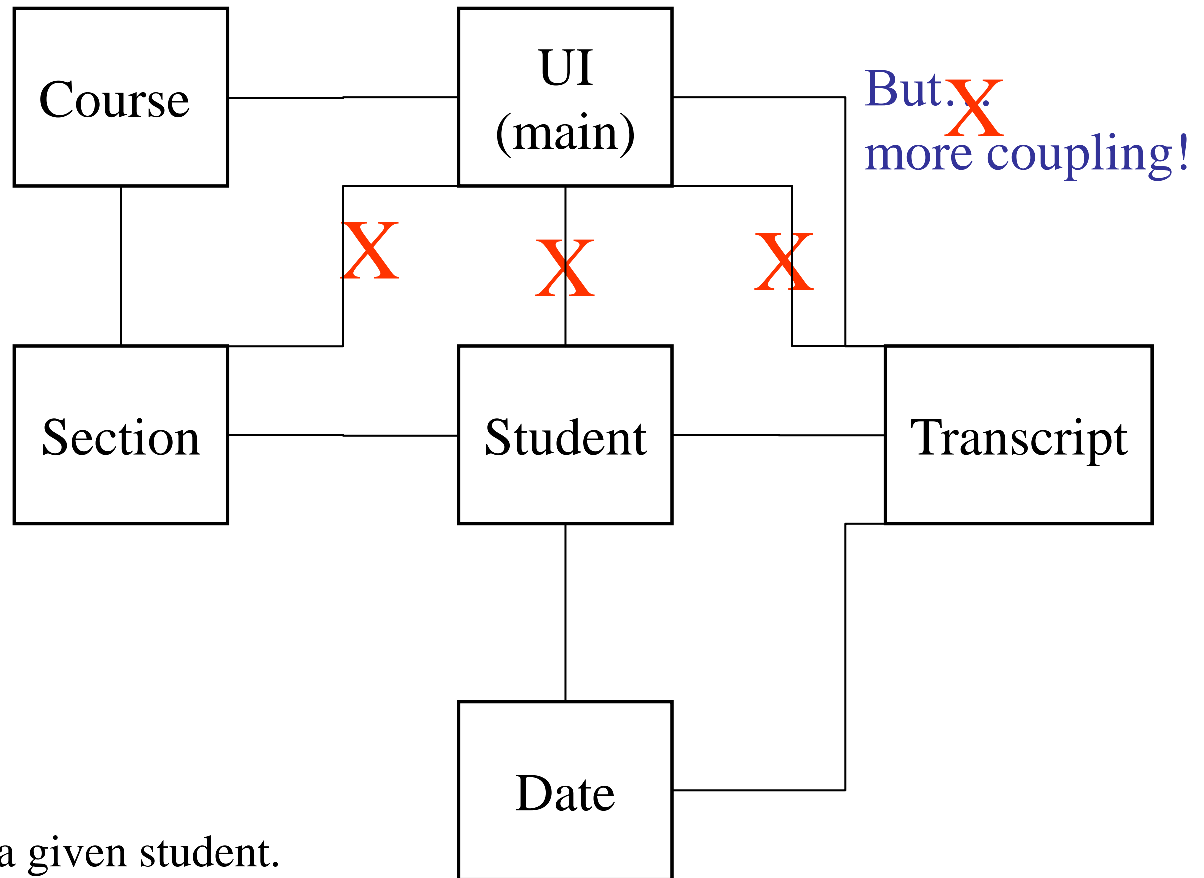
So, What's a Module?

- 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.getSection(sectionId).getStudent(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 ***<u>its</u>*** 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 Thought

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