Good Programming Practices

Things We Occasionally Forget To Do

Things we noticed

- High level stuff
  - Cohesion && Coupling issues
  - Problems separating concerns
  - Model-View-Controller pattern
- Code level stuff
  - Source folders
  - Packages
  - Comments
    - Javadoc
    - Inline
  - Testing
    - Thorough testing
    - Appropriate test names
  - Readable code

Cohesion && Coupling

- Remember: we want our projects to have high cohesion and low coupling
- What this means:
  - Given two lines of code, A and B, they are **coupled** when B must change behavior only because A changed.
  - They are **cohesive** when a change to A allows B to change so that both add new value.

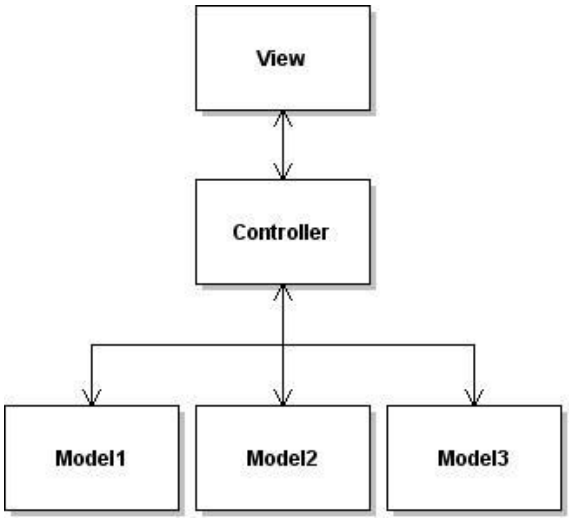
Separation of concerns

- What this means:
  - This is the process of separating a program into distinct features that overlap in functionality as little as possible.
- Consequently:
  - As little program code as possible needs to be in your view (GUI/console/whatever) class
  - This can always be fixed through refactoring, but it's a good idea to just not do it in the first place

Model-View-Controller (1)

- This is a very basic and very important pattern in software engineering
- Helps with separation of concerns
- Helps with coupling && cohesion (to a degree)

Model-View-Controller (2)



Model-View-Controller (3)

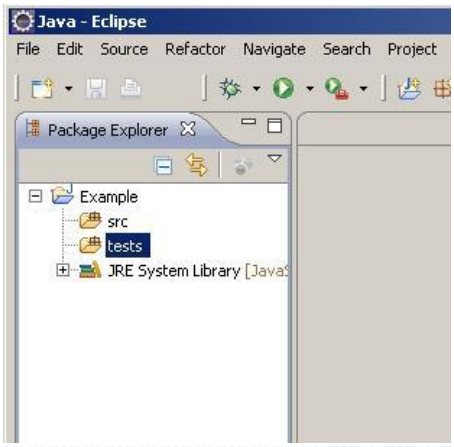
- We have discussed this extensively both this and last semester
- If you have further questions please use Google wisely or talk to Josh or Dr. Lloyd

End of high level stuff

- Now on to the code level stuff…

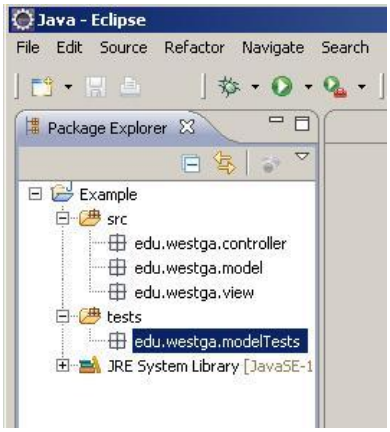
Source Folders

- Within your project you should have a number of source folders:
  - src
  - tests



Packages

- Packages within your project should have descriptive names following a reverse URL convention
- Examples:
  - edu.westga.model
  - edu.westga.modelTests
  - edu.westga.controller
  - edu.westga.view



Inline Comments

- You should always add inline comments if what the code is doing is not explicitly clear

```
 */
public class ModelOne {

    public ModelOne() {
        // Anything that is not explicitly clear in your code
        // should be accompanied by an inline comment
    }
}
```

javadoc comments

javadoc comments are similar to the XML comments from C# in that they are visible in others classes when an object is used:

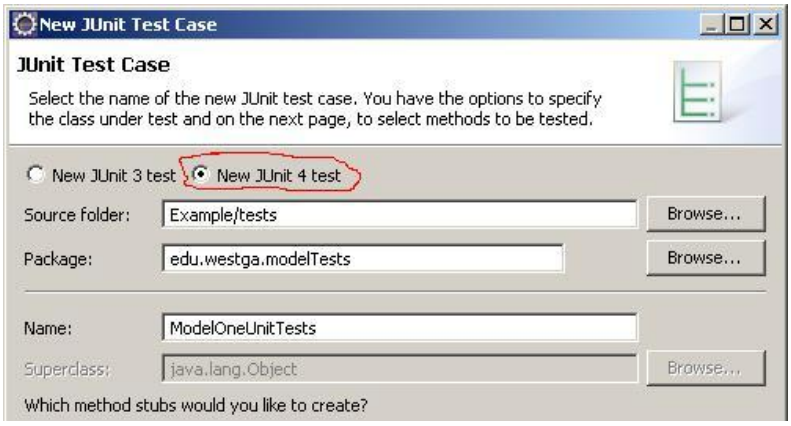
```
public void doSomethingElse() {
    this.theM1.thisMethodDoesSomethingtoSomething(this.testInt);
}
```

Testing: Thoroughness

- Always make sure to thoroughly test your classes and methods therein
  - Remember to test boundaries where applicable
  - If a method can only return 2 – 3 possible results then test for all possibilities

Testing: JUnit Version

- Make sure to always use JUnit 4



Testing: Test Names (1)

- It is very important to have proper test names
- This is self-documenting code
- Eliminates the need for comments in test cases

- Good test name:
  shouldDoSomethingWhenSomethingElseHappens

- Bad test name:
  testX

Testing: Test Names (2)

```
@Before
public void setUp() throws Exception {
    try {
        this.theM1 = new ModelOne();
    } catch(Exception e) {
        // nothing to really do here
    }
}

@Test
public void shouldGetFiveWhenFourIsPassedInToThisMethodDoesSomethingtoSomething() {
    int expected = 20;
    int testVar = 4;
    int actual = this.theM1.thisMethodDoesSomethingtoSomething(testVar);

    assertEquals(actual, expected);
}
```

Readable Code

- It is always a good practice to chop your code into sections to make it more readable

Bad

```
 */
public class ModelOne {

    int something;
    public ModelOne() {
        // Anything that is not explicitly clear in your code
        // should be accompanied by an inline comment
        this.something = 5;
    }
    public int thisMethodDoesSomethingtoSomething(int x) {
        return this.something = this.something * x;
    }
}
```

Good

```
public class ModelOne {
    // data member that will always
    // be initialized to be 5
    int something;

    /**
     * default constructor
     *
     * requires:    nothing
     * ensures:     creation of a new object of this type
     */
    public ModelOne() {
        // Anything that is not explicitly clear in your code
        // should be accompanied by an inline comment
        this.something = 5;
    }

    /**
     * multiplies the data member "something" by the passed in value "x"
     * and then returns that value
     *
     * requires:    x != null && x != 0
     *
     * @param x:    the value to multiply the data member "something" by
     * @return      "something" multiplied by the passed in value "x"
     */
    public int thisMethodDoesSomethingtoSomething(int x) {
        return this.something = this.something * x;
    }
}
```

Best

```
public class ModelOne {
    // ****************************** data members ******************************

    // data member that will always
    // be initialized to be 5
    int something;

    // ****************************** constructor(s) ******************************

    /**
     * default constructor
     *
     * requires:    nothing
     * ensures:     creation of a new object of this type
     */
    public ModelOne() {
        // Anything that is not explicitly clear in your code
        // should be accompanied by an inline comment
        this.something = 5;
    }

    // ****************************** public methods ******************************

    /**
     * multiplies the data member "something" by the passed in value "x"
     * and then returns that value
     *
     * requires:    x != null && x != 0
     *
     * @param x:    the value to multiply the data member "something" by
     * @return      "something" multiplied by the passed in value "x"
     */
    public int thisMethodDoesSomethingtoSomething(int x) {
        return this.something = this.something * x;
    }
}
```

???? || ?!?!

- Questions?