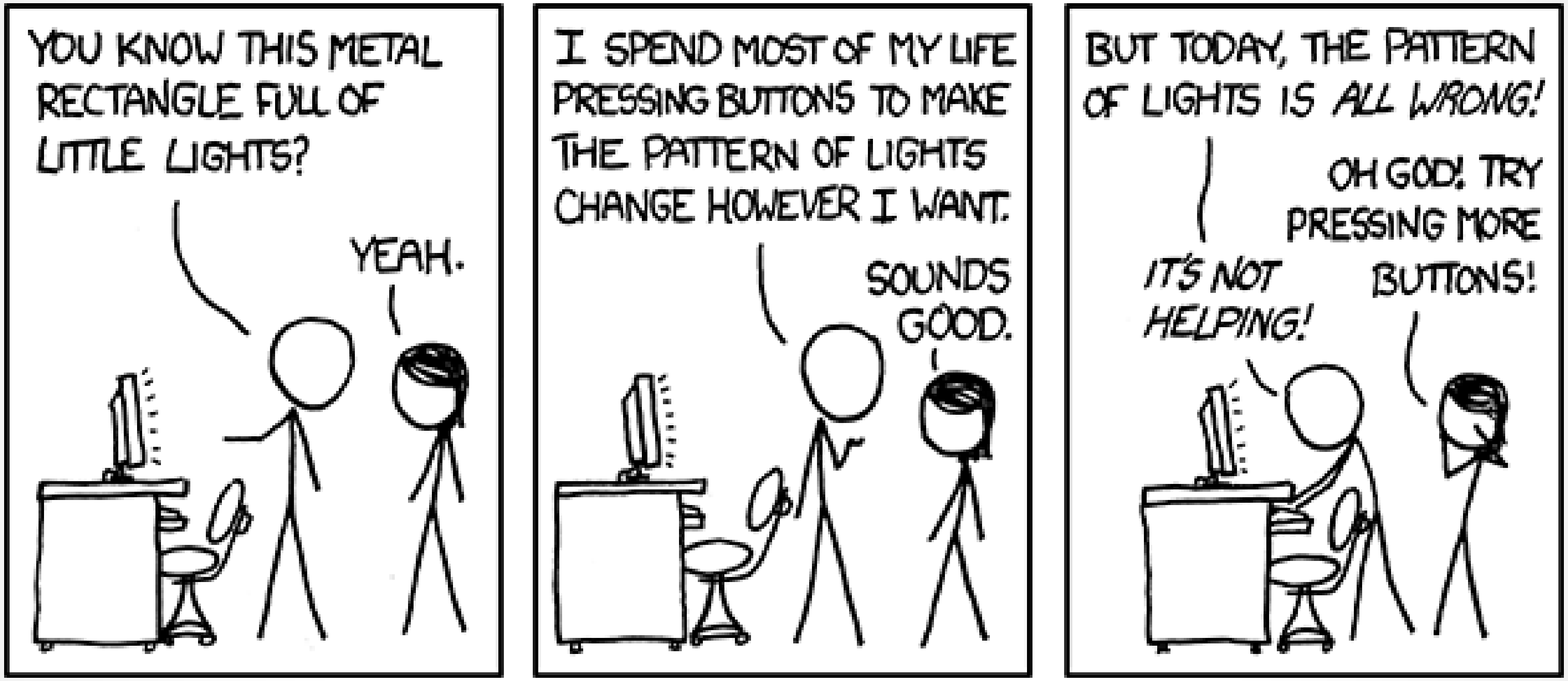
YOU KNOW THIS METAL RECTANGLE FULL OF LITTLE LIGHTS?
YEAH.
I SPEND MOST OF MY LIFE PRESSING BUTTONS TO MAKE THE PATTERN OF LIGHTS CHANGE HOWEVER I WANT.
SOUNDS GOOD.
BUT TODAY, THE PATTERN OF LIGHTS IS ALL WRONG!
OH GOD! TRY PRESSING MORE BUTTONS!
IT'S NOT HELPING!

Section 10: Final Review

Slides by Alex Mariakakis

Fall 2012 Q8

More generic questions. Suppose we have the following classes defined:

```
class Shape { … }
class Circle extends Shape { … }
class Rectangle extends Shape { … }
```

Now suppose we have a program that contains the following objects and list:

```
Object o;
Shape s;
Circle c;
Rectangle r;
```

Fall 2012 Q8

`List<? extends Shape> les;`

	legal	illegal
`les.add(s);`		X
`les.add(c);`		X
`c = les.get(0);`		X
`s = les.get(0);`	X	
`o = les.get(0);`	X	

`List<? super Shape> lss;`

	legal	illegal
`lss.add(s);`	X	
`lss.add(c);`	X	
`c = lss.get(0);`		X
`s = lss.get(0);`		X
`o = lss.get(0);`	X	

Fall 2012 Q12

We discussed several patterns during the quarter. For reference, here is a list of some of them:

- *Creational: Factory, Singleton, Builder, Prototype, Interning, Flyweight*
- *Structural: Adapter, Composite, Decorator, Flyweight, Proxy*
- *Behavioral: Interpreter, Observer, Iterator, Strategy, Model-View-Controller, Visitor*

For each of the following situations, write down the name of the design pattern that would be an appropriate solution to the given design problem. You do not need to justify your answer – just pick the best one. If more than one pattern seems appropriate, pick the best match or most specific one.

Fall 2012 Q12

a) You have a set of objects that is unlikely to change, but the set of operations on them is likely to change. You want to keep the code for each operation together in a single module.

Visitor/Procedural

b) You have a class with a many optional parameters, and want to avoid creating a large number of constructors.

Builder (partial credit for Factory)

c) You have a Point class whose interface uses polar coordinates and you want to provide a wrapper so it can be used to implement a version of Point that uses rectangular coordinates.

Adapter

Fall 2012 Q12

d) You want to wrap a library class that manages data connections so that it logs connection attempts as they occur.

Decorator (partial credit for Proxy)

e) You want to save memory costs of creating many duplicate objects of a class which has only a few distinct abstract values.

Interning (partial credit for Factory, Flyweight)

f) When new objects of your class are created the new objects should actually have a more appropriate subtype.

Factory

Fall 2013 Q2

```
/** A tic-tac-toe game */
public class TicTacToe {
 // instance variables omitted

 /** Initialize a new game object */
 public TicTacToe() { }

 /** Print the state of the game board on System.out */
 public void printGame() { }

 /** Read the player's next move from the keyboard
 * and update the game to reflect that move. */
 public void getPlayerMove() { }

 /** Calculate the computer's next move and print it on
 * System.out */
 public void computerMove() { }

 /** Calculate the computer and user's current scores
 * and print them on System.out */
 public void printScores() { }

 /** Reset the game back to the same initial state it had
 * when it was created */
 public void resetGame() { }
}
```

Fall 2013 Q2

What is (are) the major design flaw(s) in the above design? A brief couple of sentences should be enough to get the point across.

The major design flaw is that the game logic and user interface have been jumbled together in a single class. These need to be separated, particularly if we ever hope to be able to replace the user interface with a different one without a major rewrite.

Fall 2013 Q2

Describe how you would refactor (change) the initial design to improve it. Briefly sketch the class(es) and methods in your design, and what each of them do. You do not need to provide very much detail, but there should be enough so we can tell how you've rearranged the design, what the major pieces are, and how they interact.

Fall 2013 Q2

Describe how you would refactor (change) the initial design to improve it. Briefly sketch the class(es) and methods in your design, and what each of them do. You do not need to provide very much detail, but there should be enough so we can tell how you've rearranged the design, what the major pieces are, and how they interact.

The important thing that the application should be split using the MVC pattern to separate the game logic from the interface

Model: game logic. Operations would include the following from the original code:

- **Create a new game model**
- **Record a new player move**
- **Calculate a new move for the computer**
- **Return the current user and computer scores to the caller**
- **Return the current state of the game board to the caller**
- **Reset the game board to start a new game**

Fall 2013 Q2

Describe how you would refactor (change) the initial design to improve it. Briefly sketch the class(es) and methods in your design, and what each of them do. You do not need to provide very much detail, but there should be enough so we can tell how you've rearranged the design, what the major pieces are, and how they interact.

Viewer/Controller (likely a single module in a console text-based game):

- **Process initialize and reset commands from the user**
- **Read player moves and call the model to update the game state**
- **Display computer moves**
- **Display current contents of the game board and current scores**

The viewer/controller and the model would likely use some form of the Observer pattern to communicate.

Fall 2013 Q3

We would like to modify this code to change it into a generic method that works with any sorted array whose elements have type Comparable<T> (i.e., the elements can be ordered using the compareTo method). Mark the code above to show the changes needed to turn this into a generic method.

```
/** Search a sorted array of integers for a given value
 * @param a Array to be searched
 * @param x Value to search for in array a
 * @return If x is found in a, return largest i such
 * that a[i]==x. Return -1 if x not found in a
 * @requires a!=null & a is sorted in non-decreasing
 * order (i.e., a[0]<=a[1]<=...<=a[a.length-1])
 */
```

Fall 2013 Q3

```
public static int bsearch(int[] a, int x) {
      int L = -1;
      int R = a.length;
      // inv: a[0..L] <= x && a[R..a.length-1] > x &&
      // a[L+1..R-1] not examined
      while (L+1 != R) {
            int mid = (L+R)/2;
            if (a[mid] <= x)
                  L = mid;
            else // a[mid] > x
                  R = mid;
      }
      if (L >= 0 && a[L] == x)
            return L;
      else
            return -1;
}
```

Fall 2013 Q3

```
public static <T extends Comparable<T>> int bsearch(T[] a, T x) {
      int L = -1;
      int R = a.length;
      // inv: a[0..L] <= x && a[R..a.length-1] > x &&
      // a[L+1..R-1] not examined
      while (L+1 != R) {
            int mid = (L+R)/2;
            if (a[mid].compareTo(x) <= 0)
                  L = mid;
            else // a[mid] > x
                  R = mid;
      }
      if (L >= 0 && a[L].compareTo(x) == 0)
            return L;
      else
            return -1;
}
```

Fall 2013 Q3

```
/** Search a sorted array ~~of integers~~ for a given value
 * @param a Array to be searched
 * @param x Value to search for in array a
 * @return If x is found in a, return largest i such
 * that a[i]==x. Return -1 if x not found in a
 * @requires a!=null & a is sorted in non-decreasing
 * order (i.e., a[0]<=a[1]<=...<=a[a.length-1])
 */
```

Fall 2013 Q4

There's something fishy about this question. Suppose we have the following class hierarchy:

```
class Creature extends Object { }
class Fish extends Creature {
     /** Return the weight of this Fish */
     public float getWeight() { return ...; }
}
class Salmon extends Fish { }
class Haddock extends Fish { }
class Tuna extends Fish { }
```

Class `Creature` *does not have a* `getWeight` *method. Class* `Fish` *implements that method and all of its subclasses inherit it.*

Fall 2013 Q4

Write a `static` *method* `collectionWeight` *whose parameter can be any Java* `Collection` *containing* `Fish` *objects (including objects of any* `Fish` *subclass(es)). Method* `collectionWeight` *should return the total weight of all the* `Fish` *in the Collection, using* `getWeight` *to determine the weight of each individual* `Fish`*.*

Hints: Method `collectionWeight` *will need a proper generic type for its parameter. Java* `Collections` *are things like* `Lists` *and* `Sets` *that implement the* `Iterable` *interface. They do not include things like* `Maps`*, which are not simple collections of objects.*

Fall 2013 Q4

Using wildcards:

```
public static float collectionWeight(Collection<? extends Fish> f) {
     float totalWeight = 0;
     for (Fish fish: f)
          totalWeight += fish.getWeight();
     return totalWeight;
}
```

Or without wildcards:

```
public static <T extends Fish> float collectionWeight(Collection<T> f)
```

Fall 2013 Q9

A large project can be built either bottom-up starting with modules that do not depend on others and that provide infrastructure for modules higher up, or top-down, starting at the top layer and adding lower-level modules as the project progresses.

Give one advantage of a top-down development strategy over a bottom-up one. (But be brief about it!)

Problems with global design and implementation decisions can be found earlier in the project, when they are (much) less expensive and easier to fix. There is usually better visibility. Clients can see that progress is being made and partially-built systems are easier to demonstrate and try earlier.

Give one advantage of a bottom-up development strategy over a top-down one. (And be brief about this, too!)

Efficiency and feasibility problems tend to get uncovered earlier. For example, it is easier to discover that the hardware or infrastructure won't be able to provide the required performance or satisfy other required constraints. Bottom-up can require building less non-deliverable scaffolding code, although this is not necessarily a major advantage.

Final Topics

- First half of the quarter – code reasoning, specifications, ADTs, AFs and RIs, testing, equality and hashing
- Exceptions and Assertions
- Subtypes and Subclasses
- Generics
- Debugging
- Design Patterns
- GUIs
- System Integration

** Disclaimer: This list is <u>not</u> comprehensive!!!