OO Overkill
When Simple is Better than Not

Owen Astrachan

ola@cs.duke.edu

http://www.cs.duke.edu/~ola

NSF Career 9702550

Fundamental Laws

- **First do no harm**
  - **Hippocratic Oath**
- **A robot may not injure a human being, or, through inaction, allow a human being to come to harm**
  - **First law of Robots (Asimov)**
- **You know you have achieved perfection in design, not when you have nothing more to add, but when you have nothing more to take away.**
  - **Saint-Exupery**
- **Do the simplest thing that could possibly work**
  - **First Design Principle of Extreme Programming**

Where are we going? (when do we get there?)

- **Object oriented programming is here to stay (for N years)**
  - **What aspects of OOP and OO Design belong in FYI?**
  - **First Year Instruction must be about trade-offs**
    - **CS2 especially should be about algorithmic and design tradeoffs**

- **How do we (should we) incorporate design patterns in FYI?**
  - **Solving problems is important, patterns are intended to solve problems**
  - **We shouldn't teach patterns, we should teach when they're applicable, what the forces are that merit application, *and the tradeoffs in using them***

- **What's the right programming and design methodology in FYI?**
  - **It's not PSP, it's XP**

- **Teaching and curriculum design can reflect XP too**
  - **Be ready for change, hard to get it right from the start**

Tension in Teaching OO Concepts

- **Left to their own devices and designs, students cannot write correct and well-designed programs**
  - **Solution: Frameworks, Apprentice-Learning, add to existing code, *implement a design-provided***
  - **Solution: Good design comes from experience, experience comes from bad design**

- **Students relish creating programs from scratch**
  - **Is it ok to use an API from JDK 1.x, from the book, from the course, from the assignment?**
  - **There's no time to create interesting programs from scratch**

- ***OO design patterns and skills don't necessarily scale down***

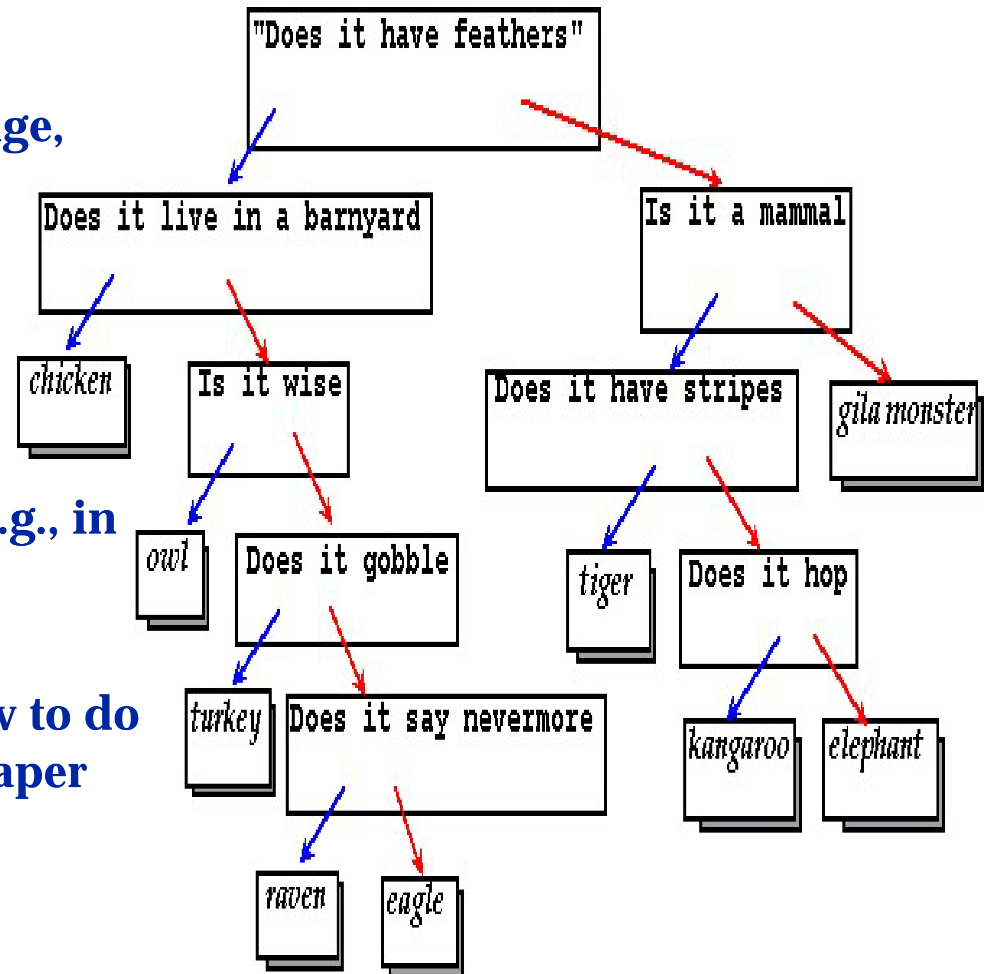
Relevant Tenets of Extreme Programming

- **What parts of embracing change can we embrace in FYI?**
  - **Evolutionary design, small releases, iterative enhancement**
  - **Simple design, don't build for the future (will you need it?)**
  - **Lots of testing, testing, testing**
  - **Refactoring: change design, not functionality**

- **What may be hard to embrace in FYI?**
  - **Code the test first**
  - **Pair Programming**
  - **Business aspects: meetings, customers, …**
- **Links**
  - http://www.xprogramming.com/what_is_xp.htm
  - http://www.extremeprogramming.org/rules.html
  - http://c2.com/cgi/wiki?ExtremeProgrammingRoadmap
  - http://www.martinfowler.com/articles/designDead.html

Twenty-Questions meets binary trees

- Build a "game tree", ask questions, add to knowledge, play again (later)
- Preliminary to RSG Nifty Assignment program
- Procedural version used, e.g., in book by Main and Savitch
- Used as an example of how to do it better in 1998 patterns paper

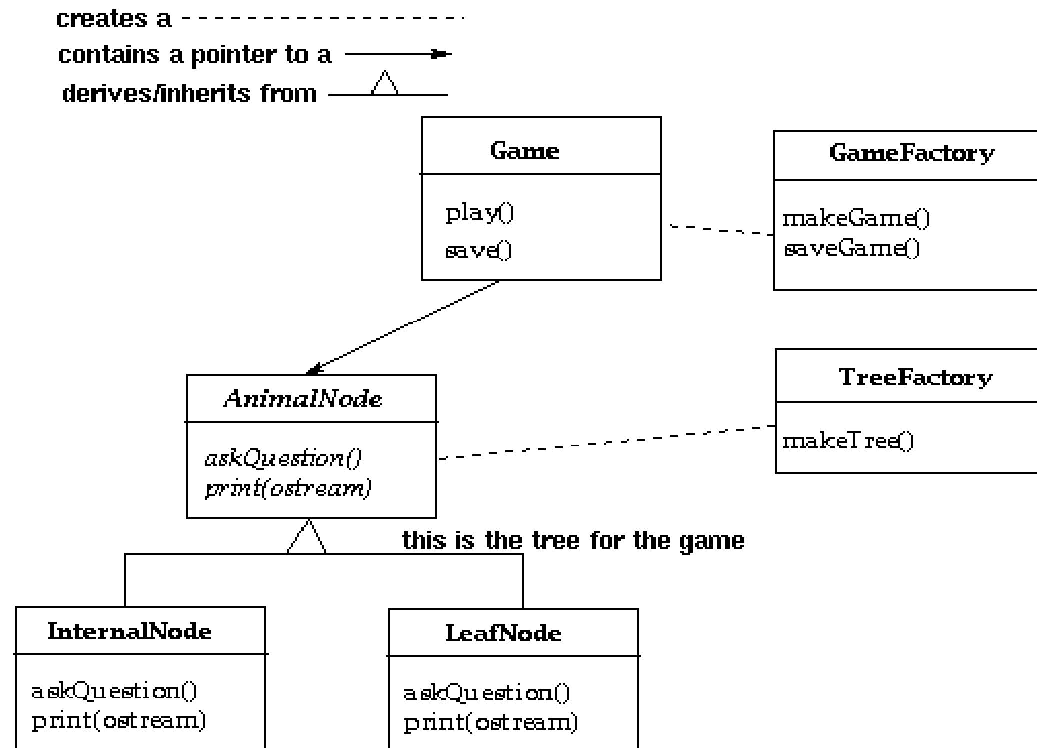


Goals of Twenty Questions assignment?

- Familiarity with trees
  - Reading, writing, traversing, changing structure
    - Preorder (read/write), postorder (cleanup)
  - Reinforce concepts with coding practice
- Interesting and (somewhat) intriguing assignment
  - Satisfaction higher when no guidance given
    - Student satisfaction not always a valid metric, but satisfaction impacts understanding and internalizing
  - Student constructed games shareable with classmates
- Provides context/hook for later refactoring
  - Revisit this in OO design course; CS2 introduces ideas and coding framework for later courses

Twenty Questions: the good, bad, and ugly

- **Factory classes**
  - Singleton?
    - Do it right
    - Do it simply
- **Adding knowledge**
  - Internal->Leaf
    - Accessor
    - Friend
    - State Pattern



- **Internalize patterns and design?**
  - Not used in subsequent programs
  - Acknowledged in later classes as "connect the dots" programming

Current version of Twenty Questions

- **Provide code that plays one game by reading the file**
  - **No tree explicitly constructed, cannot add knowledge**
  - **Recursive nature of file reading mirrors tree construction**

- **Provide no other classes/guidance: course covers trees**
  - **Use plain-old-data, public data/struct approach to tree**
  - **Code from book and in class constructs trees, prints them, copies, finds height, … all using plain-old-data approach**

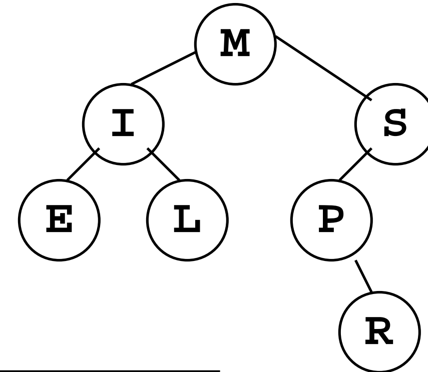
- **Revisit program in later course (ideally same course, but …)**
  - **Discuss factory, inheritance hierarchy, other OO concepts**
  - **Show how before/after approach and refactoring leads to more extendable program, *but why do that first*?**

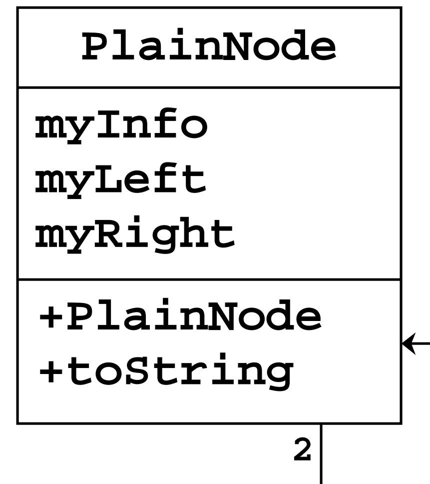
Trees: As simple as possible or too simple?

- Forces: introduce trees in a sequence of courses that rely on OO/procedural languages (OO + CS2 = ???), *simplicity first*
  - NO: a course that follows a functional programming course
  - YES: to be followed by OO/software design course
- Plain-old-data (Pascal, C, …) : doesn't extend to abstract syntax trees, but it's simple, understandable
  - What about "do it right from the start"?
    - Is an inheritance hierarchy for two node types right?
    - Is visitor warranted here?
- Distributed computing/control is hard
  - Two cases for recursion in one function vs in two classes
  - No study on this, but intuition and experience say harder

Trees: the old approach (but still new)

```
public class TreeFunctions
{
  public static int count(PlainNode root)
  {
    if (root == null) return 0;
    return 1 + count(root.myLeft) +
               count(root.myRight);
  }
}
public class Printer
{
   public static void
          inOrder(PlainNode root)
   {
     if (root == null) return;
     inOrder(root.myLeft);
     System.out.println(root.myInfo);
     inOrder(root.myRight);
   }
}
```





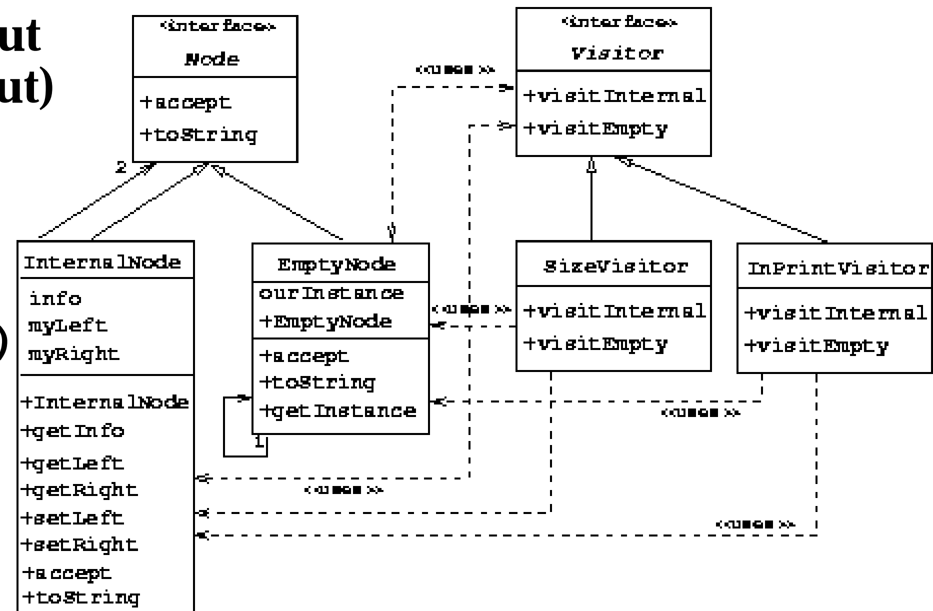
java.util.TreeMap, understanding the source

```
static class Entry {
   Object key;
   Object value;
   Entry left = null;
   Entry right = null;
   Entry parent;
   boolean color = BLACK;
  …
}
public boolean containsValue(Object value) {
   return (value==null ? valueSearchNull(root)
                       : valueSearchNonNull(root, value));
}
private boolean valueSearchNonNull(Entry n, Object value) {
  if (value.equals(n.value))  return true;
  return
     (n.left  != null && valueSearchNonNull(n.left,value))
  || (n.right != null && valueSearchNonNull(n.right,value));
}
```

Can search for a null key, special case

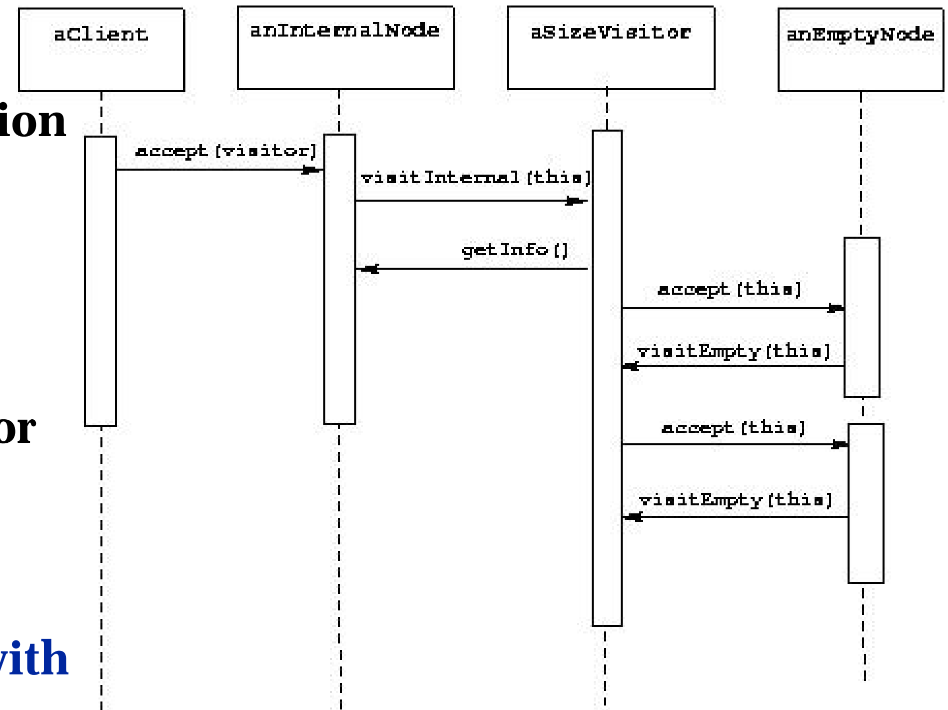
Trees + Null-object + visitor = CS2 OO Overkill

- A Node is either
  - Empty or Internal
  - Leaf (not shown here, but is in code online/handout)
- Empty Node
  - Singleton
  - Response to `getLeft()`
    - Exception
    - No-op
- Visitor
  - Encapsulate new operations over structures
  - Structure built from static set of types (e.g., nodes)



Trees + Null-object + visitor = CS2 OO Overkill

- **Visitor simulates double-dispatch**
  - **Polymorphism on operation and element**
- **Distributed *recursion***
  - **Control not centralized**
  - **Demonstrably difficult for students**
- **Writing copy/clone is trivial with plain-old data, harder with visitor**
  - **Is difficulty relevant?**
  - **Why do we study trees?**
    - **Different goals lead to different solutions**



OO Overkill in a CS2 course

```
public class InternalNode extends Node {
    public Object accept(Visitor v, Object o) {
        v.visitInternal(this,o);
    }
}
public class SizeVisitor extends Visitor {
 public Object visitInternal(InternalNode node, Object o) {
   Integer lcount = (Integer) node.getLeft().accept(this,o);
   Integer rcount = (Integer) node.getRight().accept(this,o);
   return new Integer(1 + lcount.intValue() +
                      rcount.intValue());
 }
 public Object visitEmpty(EmptyNode node, Object o){
   return ourZero;
 }
 private static Integer ourZero = new Integer(0);
}
System.out.println("# nodes = " +
               root.accept(SizeVisitor.getInstance(),null));
```

Hangman: two case studies of OO in FYI

- Contrasting goals and methodologies of studying design patterns using a Java program to play Hangman
  - Duke: http://www.cs.duke.edu/csed/patterns/oofirstyear
  - Oberlin: http://exciton.cs.oberlin.edu/cs151/labs/lab01/
  - Rice: http://www.owlnet.rice.edu/~comp212/01-spring/assignments/hangman/

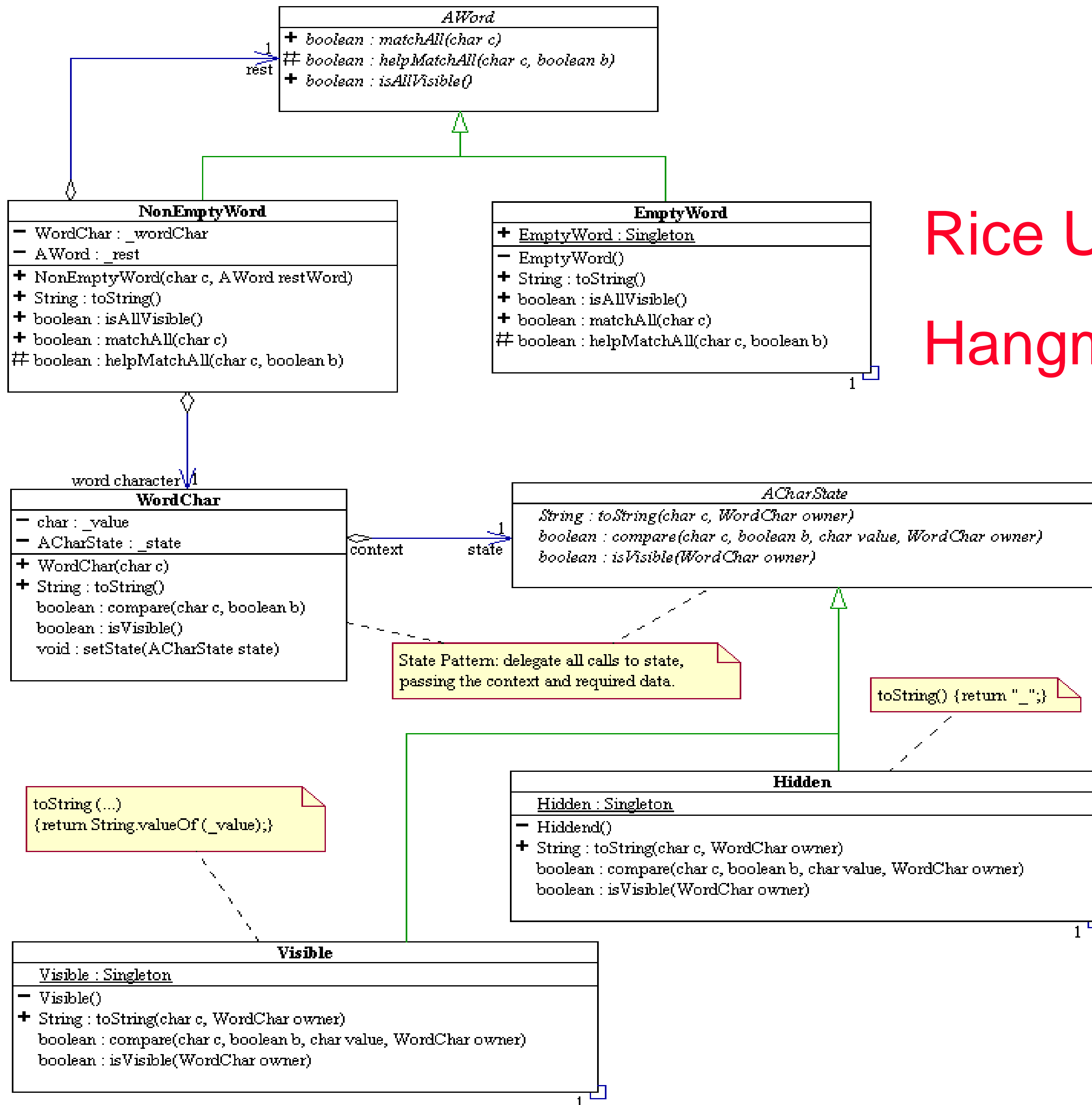
- Different goals and different audiences
  - Duke: 3rd course (or later): software design, and CS educators
  - Oberlin/Rice: 2nd course: data structures(?) and CS educators

- Before and after: study a program that doesn't use design patterns and/or object orientation, refactor into a program that does
  - Why refactor? Must be to solve a problem
  - My goal: simplest program possible to be flexible, mirror (naïve?) view of what the objects are

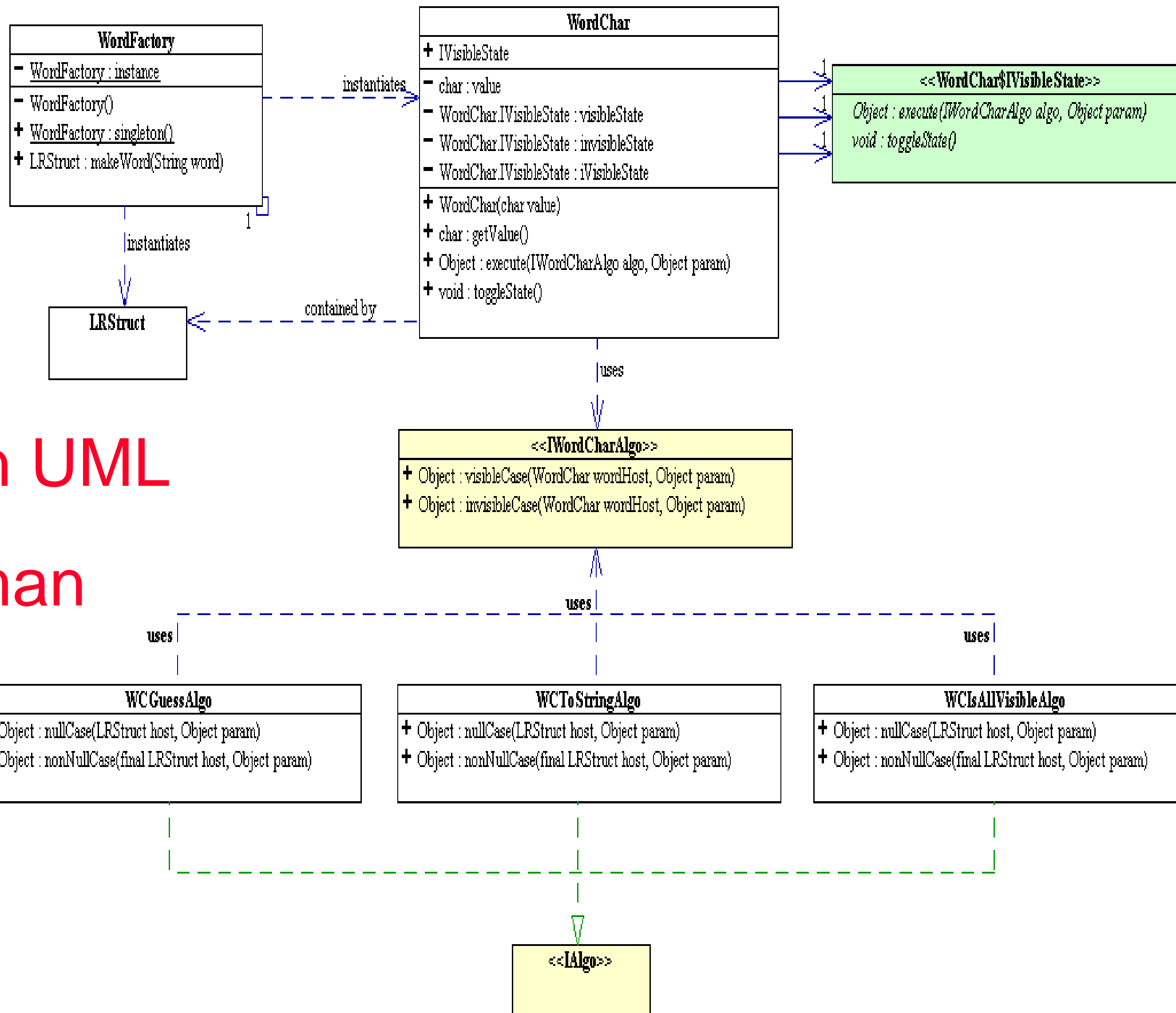
Contributions and Differences

- Designers wear different glasses
  - Visitor, Composite, MVC glasses (Rice, Oberlin)
  - Wrapper/Adapter, MVC, XP glasses (Duke)
- Each provides a software cadaver for study
  - Usable in different contexts, e.g., study details of language in first pass, design details in second (Duke at ESCC)
- What problems do design patterns solve?
  - Why is aWord a composite, why is Word an adapter?
    - Hook for string traversal vs. hook for string
- Well-implemented uses of patterns that can be used in before/after mode are worthy of study

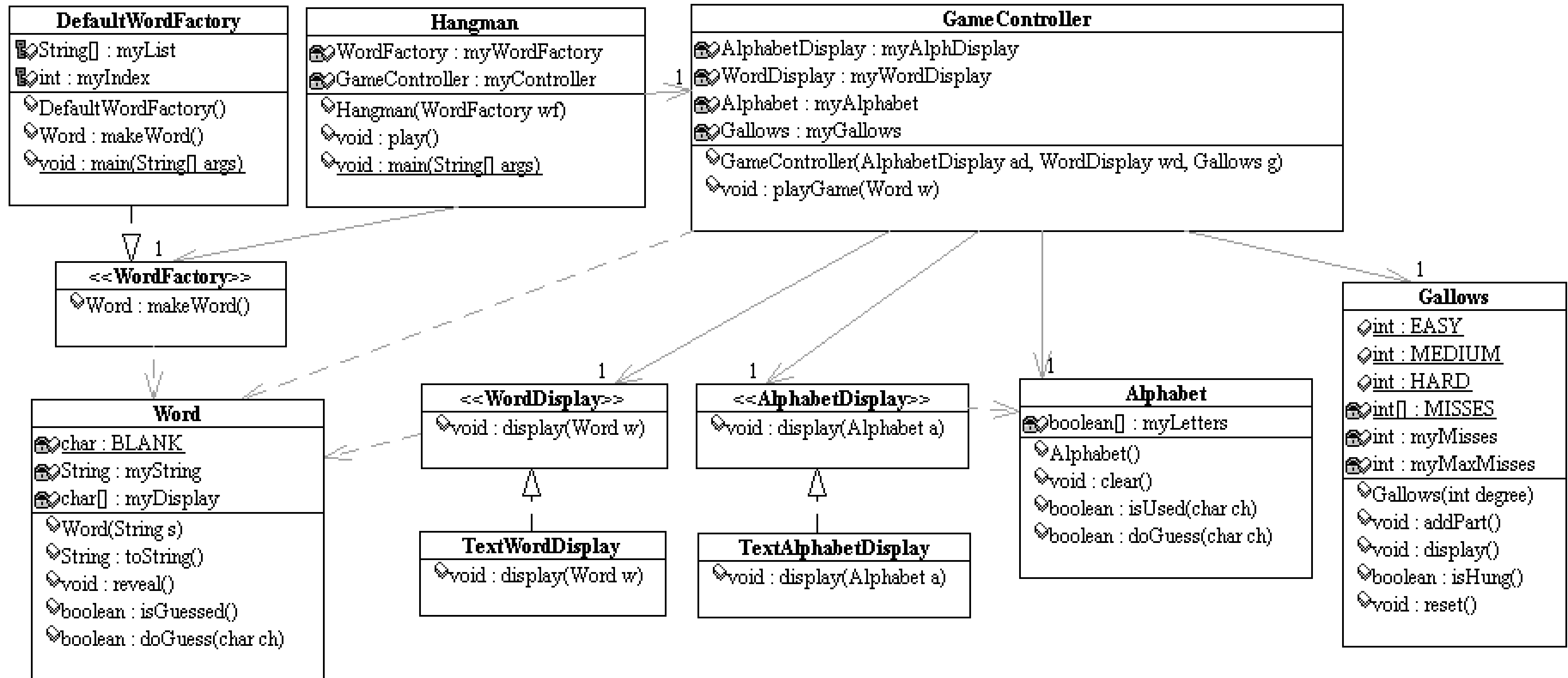
Rice UML Hangman



Oberlin UML Hangman



Duke UML for Hangman



Summary

- Design patterns are as important in FYI (and later) as lists, recursion, stacks, trees, maps, …
  - There's no room for new material, what goes?
- Embracing some of the XP change provides a context for studying and using design patterns
  - Simplicity and refactoring are embraceable early
- Not all design patterns and concepts scale down
  - Be aware of potential for OO Overkill, simplicity first
- We need more software cadavers with code and questions
  - Before and after are essential elements in using patterns