CollegeBoard

The
Teaching
Series

Special Focus in Computer Science

Object-Oriented Design

The College Board: Connecting Students to College Success

The College Board is a not-for-profit membership association whose mission is to connect students to college success and opportunity. Founded in 1900, the association is composed of more than 5,000 schools, colleges, universities, and other educational organizations. Each year, the College Board serves seven million students and their parents, 23,000 high schools, and 3,500 colleges through major programs and services in college admissions, guidance, assessment, financial aid, enrollment, and teaching and learning. Among its best-known programs are the SAT®, the PSAT/NMSQT®, and the Advanced Placement Program® (AP®). The College Board is committed to the principles of excellence and equity, and that commitment is embodied in all of its programs, services, activities, and concerns.

Equity Policy Statement

The College Board believes that all students should be prepared for and have an opportunity to participate successfully in college, and that equitable access to higher education must be a guiding principle for teachers, counselors, administrators, and policymakers. As part of this, all students should be given appropriate guidance about college admissions, and provided the full support necessary to ensure college admission and success. All students should be encouraged to accept the challenge of a rigorous academic curriculum through enrollment in college preparatory programs and AP courses. Schools should make every effort to ensure that AP and other college-level classes reflect the diversity of the student population. The College Board encourages the elimination of barriers that limit access to demanding courses for all students, particularly those from traditionally underrepresented ethnic, racial, and socioeconomic groups.

For more information about equity and access in principle and practice, please send an email to apequity@collegeboard.org.

Visit the College Board on the Web: www.collegeboard.com.
AP Central is the official online home for the AP Program and Pre-AP: apcentral.collegeboard.com.

Table of Contents

Important Note:

The following materials are organized around a particular theme that reflects important topics in AP Computer Science. They are intended to provide teachers with professional development ideas and resources relating to that theme. However, the chosen theme cannot, and should not, be taken as any indication that a particular topic will appear on the AP Exam.

Introduction from the Editor

Fran Trees
Drew University
Madison, New Jersey

I remember sitting in my Spanish class having the teacher ask me a question in Spanish. Mentally, I would parse the question, translate each word into English, formulate my answer in English, translate my answer (usually word by word) into Spanish, and then verbalize my Spanish response (which was occasionally understood by my teacher). The most meaningful advice given to me at that time was, "My dear, you must learn to THINK in Spanish."

With that said. . .

The transition to Java finds AP Computer Science (AP CS) teachers in various stages of "living the language." Java was the first and only language some novice teachers ever studied. These teachers are able to "think in Java." For most of us, this is not the case. What most of us have done during this transition is looked at materials we have worked with in the past and attempted to translate them into Java. Because the object-oriented paradigm is now a focus of AP CS, and successful translation from C++ is extremely difficult to accomplish with this as a goal, we really shouldn't continue to attempt this translation. Our goal now should be to learn to THINK in Java and live and breathe in the object-oriented world. To help you achieve this goal, our theme this year is object-oriented design. It is hoped that the materials contained in this section help you in the classroom and, more importantly, provide ideas and directions that will assist you in the development of your own materials. We have described teaching strategies that use role playing and unit testing; included sample design problems, team projects, lab assignments, and worksheets; and finally, provided you with a list of Web-based resources that point to object-oriented materials. I would like to thank the contributors for their hard work and continued commitment to our AP CS family.

Immersing AP CS Students in Object-Oriented Design Using Role-Playing

Robb Cutler
The Harker School
San Jose, California

Introduction

Roughly one-third to one-half of my AP CS-AB students have no programming or computer science background, and many of the remaining students have no or limited understanding of good object-oriented design principles. I introduce these concepts to my students using some simple exercises whereby for two weeks at the beginning of the year, they become actors immersed in the world of objects.

Initial Exercises

We start with the "first-day" role plays written by David Levine of St. Bonaventure University. These consist of some simple "classes," which can be acted out by the students.

For example, an "acrobat" can do three things: Clap his or her hands a given number of times, perform knee bends (squatting and then standing up) a given number of times, and report how many exercises have been completed.

The methods can be given to students as instructions on a sheet of paper or written on the whiteboard at the front of the classroom. For an acrobat, the methods are:

*When you are asked to **clap**, you will be given a number. Clap your hands that many times.*

*When you are asked to perform **knee bends**, you will be given a number. Stand up and sit down that many times. Note that if you are told "two," then you will stand up twice AND sit down twice.*

*When you are asked to **count**, you will reply (verbally) with the total number of exercises you have done. Note that clapping four times counts as four exercises and performing two knee bends counts as two. If you have done these things (and only these things), your reply should be "six."*

As the teacher, you instantiate an acrobat by asking a student—in this case, a girl named Maya—to construct herself as an acrobat.

Maya, construct yourself as an acrobat.

To call a method, say the student's name and invoke the method with any appropriate parameters. For instance:

Maya, clap three.

Maya, knee bend two.

Maya, count.

Maya, clap one.

Maya, count.

As you ask her to do these things, Maya will clap three times, knee bend twice, say "five," clap once, and say "six."

You can instantiate more than one acrobat object by asking other students to also construct themselves as acrobats.

Sean, construct yourself as an acrobat.

Killy, construct yourself as an acrobat.

You can then invoke methods for different acrobats.

Sean, clap four.

Killy, clap two.

Sean, count.

This helps students differentiate between the acrobat class (the instructions to an acrobat) and an acrobat object (the student). You can emphasize this by giving some "illegal" instructions such as:

Acrobat, clap four. (Only acrobat <u>objects</u> can clap.)

Maya, clap. (Clapping requires a count.)

Clap three. (No object specified.)

Be sure to explain why these instructions are improper for the students.

After 15 minutes of this role play, my students rarely make the otherwise common mistake of calling a method with the class name rather than with the object name. When they do, a questioning "Acrobat clap?" from me anytime during the rest of the year brings a smile to the student's face and instant recognition of the problem.

You can also start associating the instructions with Java code. Write on the board the line

```
Acrobat sean = new Acrobat();
```

as you tell Sean to construct himself as an acrobat. Write

```
sean.clap(4);
```

when you instruct him to clap four times.

Students very quickly learn the Java syntax for instantiating objects and for calling methods in a way that is not only fun but also easily remembered.

Other classes in the first-day role plays include:

- a calculator that can add, subtract, and multiply pairs of numbers
- a lazy calculator that can add either two or four numbers (overloading the add method) and does so by creating a calculator object and asking it to do the work
- a die that returns a random roll, keeps track of the number of rolls, and can reset the roll counter when asked
- a blackboard that draws shapes on the board

Naturally, part of the fun is coming up with your own classes. You can also extend the role plays to give students an understanding of concepts such as inheritance, overriding, overloading, interfaces, and abstract classes.

Designing an ATM Machine

After one class period to get used to the idea of objects and role plays, I ask my students to design a role play for an ATM machine.

I suggest that they have certain classes (such as a display, a cash dispenser, a card reader, and a bank account), and then I ask them to decide what other classes and methods they need.

My goal is to guide them in their design without doing the work for them. I try to let the students make design decisions whenever possible. My only input is to gently lead them away from catastrophic mistakes while letting them have freedom in their design. For instance, they can decide whether (a) each button on the ATM keypad is a separate object or (b) the buttons are just part of the keypad object as a whole.

Once they have decided which classes and methods they need and have written the scripts for the methods, they act out their role play. In doing so, they often find "bugs" in their methods as they realize that they have forgotten to, for example, have the card reader eject the customer's card!

The result of this exercise is that students begin to think in object-oriented ways. In addition, they learn the importance of top-down design and of designing before coding. Naturally, other large object-oriented projects would work as well. Other design ideas I've heard about (but not tried) include a vending machine, a building with multiple elevators, and a calculator.

The Marine Biology Simulation Case Study

Part of the AP CS curriculum includes learning and being able to modify the Marine Biology Simulation Case Study (MBSCS), a large and reasonably complex simulation of fish interacting in a marine environment. Most students, even after several months of programming, find the many pages of code somewhat daunting, to say the least.

Using the MBSCS role plays initially developed by Steve Andrianoff and David Levine and modified by me, I ask my students to become fish (regular *Fish*, *SlowFish*, and *DarterFish*), a *BoundedEnv*, a *Simulation*, and even a random number generator. We spend several class periods performing the case study.

As with the first-day role plays, personifying the objects in the simulation makes the code seem much more manageable to the students and brings a deeper and longer-lasting understanding of the case study. Six months after we did the performance, students still remembered who played the part of the *SlowFish* or the *BoundedEnv* and, more importantly, they remembered how the objects interacted with each other.

Clearly, acting out the case study is no substitute for an in-depth review of the code and hands-on practice modifying and extending the case study. Starting with a role play, however, gives students a good introduction to a large and complex simulation and allows them to experience the interactions among the various objects.

Summary

By the end of two weeks of role-playing, my students—without typing a single line of Java code—have accomplished four major goals.

First, they have acquired a good intuitive sense of object-oriented design. They have learned about classes, objects, methods, constructors, parameters, instance variables, local variables, overloading, and inheritance. They understand the difference between a class and an object, between instance variables and local variables, and between methods and constructors. They can instantiate objects and know that they need an object name in order to call a method.

Second, my students have designed, written, and executed (through role plays) a large and complex program: the ATM machine. In doing so, they have learned good object-oriented and top-down design principles and debugging strategies. They have also learned a good deal of Java syntax as well.

Third, they have immersed themselves in the MBSCS and learned how it works before looking at any code. When they get to the actual code, they are very comfortable with it. Finally, my students have learned an invaluable resource, which they now have at their disposal throughout the course: the ability to role-play their code. It is not uncommon to hear students in my class talking to themselves throughout the year as they personify the objects in their code.

Role-playing is one more way to make programming and computer science more accessible to students, teach them good skills, and, most importantly, make class much more fun!

Resources

The first-day role plays and the MBSCS role plays can be found at
web.sbu.edu/cs/dlevine/RolePlay/roleplay.html.

Chris Nevison of Colgate University has developed some other role plays suitable for AP
CS students. These can be found at
cs.colgate.edu/APCS/Java/RolePlays/JavaRolePlays.htm.

Automated Unit Testing with JUnit

A tool for unit testing, design, lab grading,
student motivation, reasoning, and stress reduction

Dave Wittry
Troy High School
Fullerton, California

The use of JUnit to aid in the design of a class and the creation of error-free code has been well discussed in other works. While this paper discusses design and error-free code to some degree, its main focus is on the most useful and motivational aspects of JUnit for the student and teacher in a high school computer science classroom. The approach is, therefore, practical. The following pages simply detail the understanding and advice of one teacher after a year's use of JUnit in the classroom. Accompanying the discussion of JUnit below is an explanation of a complementary and free lab grading tool, Jamtester.

JUnit from the Teacher's Perspective

What is JUnit?

JUnit (www.junit.org) is open-source software, an API framework, used to automate unit and regression testing. In short, it is software-exercising software. Martin Fowler says of it, "Never in the field of software development was so much owed by so many to so few lines of code."

How does JUnit work?

Each Java class that you write will correspond to a JUnit Java class that will test the methods of your class.

Here's an example:

```java
// your students are trying to write/test this class
public class Student {
      private int exam1, exam2;

      public Student(int ex1, int ex2) {
            exam1 = ex1;
            exam2 = ex2;
      }

      public int bestScore() {
            if (exam1 > exam2) return exam1;
            return exam2;
      }

      public int worstScore() {
            return exam1; // intentional "failure".
      }                  // student needs to fix logic
}
```

this tests this

```java
// this is the test class
import junit.framework.*;

public class StudentTest extends TestCase {

      // here is one unit test method
      public void testBestScore() {
            Student stud1 = new Student(90, 85);
            int answer = stud1.bestScore();

            assertEquals(answer, 90); // compare to known answer
      }

      // here is one unit test method
      public void testWorstScore() {
            Student stud1 = new Student(90, 85);
            int answer = stud1.worstScore();

            assertEquals(answer, 85); // compare to known answer
}
      // more unit tests not shown…

      // will generate results of tests (text output)
      public static void main(String[] args) {
            TestSuite suite = new TestSuite(StudentTest.class);
            junit.textui.TestRunner.run(suite);
      }
}
```

Classes and methods from the JUnit framework are in **bold**. The above generates the following text output: "Tests run: 2, Failures: 1, Errors: 0," however, with a bit more detailed information.

As far as the `testBestScore()` method above,
- an object to be tested was instantiated,
- a method on the object was invoked,
- and the returned answer was compared with the known answer by using JUnit's assertXXX most common method, assertEquals.

That's it!

If the test fails, the student knows to edit her code and rerun the test suite.

How and when should I introduce and use JUnit in the classroom?

At the beginning of the school year, you routinely spell out the methods to write for each Java class; this is the perfect time to give students a JUnit test file for their class, complete with a full suite of test cases.

Students will benefit by this early introduction. They will not need to devise the test cases themselves, nor will they need to worry about writing a main program to test their classes. Instead, they will be able to focus on writing their classes, and they will naturally over time gain a comprehensive understanding of JUnit, including arranging syntax and developing a complete set of test cases. They will also know when they have finished the lab, that is, when all tests are successful.

You, as a teacher, will also benefit. Since you do not begin the year by teaching "main" with all of its syntax and keywords, you can focus immediately on teaching objects and inheritance. This way, you can, from the start, model and communicate good design methods through the use of test cases by guiding students through a systematic way of testing their code. Moreover, you will feel confident that your students will be writing correct method headings, and your students will feel confident as they earn perfect scores on their labs.

How should I use JUnit to help students test their classes?

Over time you will give your students JUnit test files which are less and less complete. In other words, the test suites will intentionally be missing some of the test cases. This way, the students will devise and write test cases on their own, and they will, at the same time, develop the analytical skills necessary to recognize that they have produced a complete set of test cases.

Eventually, you will stop giving them JUnit test files altogether. At this point, an amazing thing will happen: the process will become completely student-centered. Without being prompted, students will create their own JUnit test files. (See also "JUnit from the Student's Perspective" below to learn about their motivation.) In this manner, you will have systematically and gradually taught them to analyze, design, test, and function independently of you, the teacher.

How can I *possibly* fit one more topic into my course?

I wondered this, too, but the answer is simple: the students learn JUnit on their own. Since it is easy to teach and can be introduced slowly, it consumes very little instructional time. The payoff is extraordinarily high in terms of student understanding and performance.

How can JUnit aid in regression testing?

JUnit **is** regression testing! But it is regression testing that is efficient and precise. It can perform and repeat tests at any time.

Think of it this way: how many times does a student, or professional for that matter, test a method until it *seems* to work, and then destabilize the method already tested by making changes to code?

Often. *Too* often, in fact.

JUnit is a tool that enables *all* tests to be run at *any* time. Thus, students will know immediately and at will that a tested method has begun to behave badly.

Why are *ad hoc* testing techniques flawed?

When using *ad hoc* testing techniques such as embedded print statements and customized main programs, students are not naturally inclined to rerun or, more importantly, recreate tests on their own. Who would? Such techniques are inefficient, tedious, time-consuming, and error-prone.

As you know, students wait until they have *completely* finished the creation process before they test their projects even *once*. This approach illustrates flawed reasoning and compounds the problems that they will encounter as they attempt to remove errors in their code. As you also know, coding and testing should not be treated as disparate and independent functions. They are integral functions, complementary to one another. JUnit reconnects coding and testing in a manner that causes the students to embrace logical thinking and good coding practices from the outset.

How does JUnit help students design cohesive methods?

As they master JUnit testing techniques, students will also be preparing to design cohesive methods, that is, methods that perform a single well-defined task. Importantly, cohesive methods are hallmarks of a well-designed class.

Using JUnit early in the year, you will lecture about the proper design of cohesive methods, among other topics, and you will reinforce your lectures by model. As the year progresses, you will wean your students from the complete test suites, causing them to test and design more and more on their own and undoubtedly applying the good design skills that they have learned through your modeling. This way, students will be more likely to recognize a method that needs refactoring, that is, broken down into several, more cohesive methods. Through these techniques, you will communicate your goal: that students will become good testers and good designers.

How can JUnit help me give my students more feedback?

When you give your students a complete test suite early in the year, you will essentially be giving them the ability to receive your feedback continuously but vicariously throughout the year. The feedback you give, that is, that JUnit gives, is immediate, specific, and continual. Students will run their tests over and over, that is, each time they change code. Your own feedback could not be any more specific, useful, immediate, convenient, or time-saving.

Does all of this mean that I will save time on grading labs?

In one word, YES.

In another word, Jamtester.

Jamtester, **J**Unit **A**uto-**M**atic **Tester**, is a free software tool which will enable you to grade *all* of your students' labs *simultaneously*.

You will no longer need to compile and run each student lab individually.

You will no longer need to stop and write down results.

You will no longer need to interact with a program by typing in user input to grade it.

Jamtester works very simply: you give it a JUnit test file, student directories containing the .java file(s) to be graded, and the .jar/.class files needed to make the project compile. Hit "go" and sit back as it grades the labs.

The tool allows you to save results to a .csv file, a format readily imported by Excel and most grading programs. You can also edit the student's source directly in the tool, then rerun the tests and have the new results right there in a table.

sample output
(see www.jamtester.com for an animated demo)

Field 1	Field 2	Field 3	Success	Percentage	testBestScore	testWorstScore
Litvin	Maria	3456	1 of 2	50.0%	F	
Poll	David	9876	2 of 2	100.0%		
Taylor	Reba	4321	0 of 0	0.0%	C	C
Trees	Fran	1234	1 of 2	50.0%		E
Walker	Henry	8765	1 of 2	50.0%		F

```
junit.framework.AssertionFailedError: expected:<90> but was:<85>
        at junit.framework.Assert.fail(Assert.java:47)
        at junit.framework.Assert.failNotEquals(Assert.java:282)
        at junit.framework.Assert.assertEquals(Assert.java:64)
        at junit.framework.Assert.assertEquals(Assert.java:201)
        at junit.framework.Assert.assertEquals(Assert.java:207)
```

JAM*Tester Results for StudentTest

File   Help

this is the
code
example
from above

student folders
    Litvin Maria 3456
    Poll David 9876
    Taylor Reba 4321
    Trees Fran 1234
    Walker Henry 8765

A folder of student folders (each folder in this
example contains a Student.java file)

JUnit from the Student's Perspective

Can students create, edit, and run JUnit tests easily and independently of any IDE?

Yes. In fact, Jamtester is not only a teacher tool, but it is also a student tool. Regardless of the IDE you choose for your classroom, students will be able to test their source files using JUnit files. Students can work simultaneously with their IDE and the Jamtester student tool, testing as often as they like. If they wish, they can even edit their source files directly in Jamtester without having to switch to their IDE.

(see www.jamtester.com for an animated demo)

Can use of JUnit alleviate students' stress levels?

If you have read my article to this point, you will not have a hard time understanding that JUnit will indeed reduce or likely *eliminate* student stress. If you are starting your reading at this point, this may sound far-fetched.

Stress in students arises from uncertainty. Using some of the traditional procedures, students cannot be certain that their coding is actually and accurately finished. Although they can be capable of *convincing* themselves that their code is perfect, they turn in their solutions, still anxious that they may have forgotten something—probably something fundamental, something that will cost them valuable points.

However, by using JUnit, especially from the beginning of the year, the student is virtually guaranteed that she will turn in a perfectly working Java class. *You* are giving her that chance. She knows that if she invests the time, she will reap the benefits. Most importantly, because she knows exactly where she stands academically, she can operate essentially stress-free. *She* is in control of her grades.

The famous physicist Niels Bohr once said, "An expert is a man who has made all the mistakes. . . ." However, we also know that the student is a person who becomes stressed at the prospect of *making* mistakes. JUnit gives your students the academic and emotional freedom to make mistakes and become unstressed experts.

How does the use of JUnit increase student-student cooperation?

While I do not allow my students to pass their source code around the lab for others to see, I do encourage them to share their JUnit files. This way, they share the work associated with devising test cases by brainstorming together. A natural and positive outcome of a student discussion of test cases is that the students themselves analyze their own code. In so doing, they may very well discover a deficiency in their own algorithm or design before they even write or run the test.

If your students work in pairs, they may jointly create test suites. In a recent study, Kessler and Williams determined that this technique reduces cheating. Pair-programming, along with unit testing, is part of a discipline of software development called Extreme Programming. See bibliography.

Selected Bibliography

Vaaraniemi, Sami. "The benefits of automated unit testing." *The Code Project.* 9
    November 2003. 29 April 2004
    www.codeproject.com/gen/design/onunittesting.asp.

Jeffries, Ron. "Essential XP: Emergent Design." *Xprogramming.com: An Extreme
    Programming Resource.* 21 October 2001. 29 April 2004
    www.xprogramming.com/xpmag/expEmergentDesign.htm.

Jeffries, Ron. "What is Extreme Programming?" *Xprogramming.com: An Extreme
    Programming Resource.* 8 November 2001. 29 April 2004
    www.xprogramming.com/xpmag/whatisxp.htm.

Kessler, Robert and Laurie Williams. "Experimenting with Industry's 'Pair Programming'
    Model in the Computer Science Classroom." *Pair Programming.* 29 April 2004
    www.pairprogramming.com.

Weirich, Jim. "Design by Contract and Unit Testing." Online posting. 6 July 2003. Ruby
    Buzz Forum. 29 April 2004
    www.artima.com/forums/flat.jsp?forum=123&thread=6794.

Modifying and Creating Classes: Money and Fraction

Joe Kmoch
Washington High School
Milwaukee, Wisconsin

Justification:

We are going to study an Abstract Data Type (ADT) for manipulating amounts of money. The justification for an ADT about money might be the development of classes to handle money from different countries or to maintain exact accuracy in calculations with money.

Part A:

In this part, you will study the class `Money.java`. You will need to perform several tasks with this class.

Specification:
Money will be created as a Java class. Like any class, it is a recipe (outline, blueprint) for a structure, a declaration. When thinking about manipulating U.S. money, we know that we'll need just two data items (instance, member, state variables):

```
myDollars

myCents
```

We'll begin with five actions (member functions) that we can do to a money object.

```
initialize                   // constructor-like

add two amounts of money     // modifier (mutator)

create a string to display   // accessor (observer)

access the number of dollars // accessor (observer)

access the number of cents   // accessor (observer)
```

At this point, you should study the sample program `MoneyTest.java` and its sample output to see how the class `Money` works (see Appendix within this article). If you have access to the code, try these tasks:

1. Compile `MoneyTest.java` and `Money.java`, then run `MoneyTest.`

2. Enter 6.52 (i.e., enter 6 for dollars then enter 52 for cents).

    a. What answer do you see displayed? _____

    b. What should the answer be? _____
        (The problem you observe is left as an exercise for you a bit later.)

Now, take a look at the class `Money.java` (in the Appendix) and answer these questions.

3. Look at the constructor(s).

    a. How many constructors do you see? _____

    b. Write the header for the default constructor.

        _____

    c. To what values are the instance variables set in the default constructor?

        _____

    d. Instead of using the method `initialize` in the other constructor, write code to set the instance variables `myDollars` and `myCents`.

        _____

        _____

4. Look at the code for the `add` member function.

    a. Why does the member function `add` have only one parameter when you know that two values are necessary to add?

    _____

    _____

    b. In the comments just above this method, what does the word "this" refer to?

    _____

    _____

    c. Consider this code to replace the code in the add member function.

    ```
    Money result = newMoney();

    result.myCents = myCents + amount.myCents;

    result.myDollars = myDollars + amount.myDollars

    return result;
    ```

    This code will correctly add two amounts of money, but the result will be unusual. Can you create two values of money for which result will not be in the form you would expect if you added amounts of money? Indicate what the result for your amounts would be using this code.

    _____

    _____

5. Look at the `toString` member function.

    a. As you should have noticed in exercise 2 above, the output was incorrect. What do you have to do in order to make the output appear correctly?

    _____

    _____

    b. Rewrite the code for this member function which will return the proper `String`. (Hint: You'll need to use an `if` statement.)

6. Run `MoneyTest` again and use the value 6.75.

    a. What output is displayed? _____

    b. Write a new private member function `normalize` to the `Money` class that causes the number of cents to be between 0 and 99 and the dollar amount to be adjusted accordingly. For example, if the amount of money was 4 dollars and 112 cents, the result of `normalize` would be 5 dollars and 12 cents.

    c. Where would you call the `normalize` member function?

    d. Enter your new method `normalize` into the `Money` class and also call this method where you have indicated in c. Then test it out.

There are several other methods that could be included which would make the `Money` class more useful. One of these is multiplication.

7. Think about what a multiplication method might do.

    a. Does it make sense to multiply two money values? Why?

    _____

    _____

    b. What does make sense for multiplication involving money?

    _____

    _____

    c. What is the type for the return value of this method?

    _____

    d. Devise an algorithm (set of instructions) which would multiply a money value by an integer. Write your code in the space below.

e. Does the algorithm change if you are multiplying by a double versus an integer?

_____

_____

f. Write the complete Java method to do multiplication in the space below, then enter it into your `Money` class and test it using `MoneyTest`. (Don't forget to call your `normalize` method to get reasonable answers.) Test your new routine by multiplying 2.50 by 3 and several input values of your choosing.

8. Optional for experts:

a. Work the previous exercises for a division method.

b. Work through the previous exercises for a subtraction method.

c. Are there any other operations on money which should be considered for this class?

_____

_____

Part B: Another Class – Fraction

You want a class that will allow you to deal with fractions.

Let's think about this class.

1. What instance variables might you need to include? _____

2. What values might you initialize a `Fraction` object to if you have a default constructor?

    _____

3. The initial version of the five `Money` class methods were described.

    a. List those methods which would be appropriate for our new `Fraction` class.

        _____

        _____

    b. What other arithmetic operations on a `Fraction` object might we want to also include?

        _____

    c. `normalize` is another method we'd want to have. What would this method do for a `Fraction` object?

        _____

4. After you create a `Fraction` class that at least constructs a `Fraction` object and then does a single operation (such as add), create a client test for your class. This might allow the user to input the operation to test and the fractions to use. You could have the client test program ask the user for the operation and then ask for each part of each of the two fractions. An alternative would be to allow the user to input the operator followed by four integers representing the two fractions. For example, if the user entered + 7 10 2 5, the program would add the two fractions 7/10 and 2/5.

Appendix

```
MoneyTest.java
/**
 * MoneyTest is a class used to test the class Money.
 */
public class MoneyTest
{

  public static void main(String[] args
                                  throws IOException
  {
    BufferedReader console = new BufferedReader(
      new InputStreamReader(System.in));
    String input;

    Money valueA = new Money();
    Money valueB = new Money(8, 50);
    Money sum;
    int dollars, cents;

    System.out.print("Enter a number of dollars: ");
    input = console.readLine();
    dollars = Integer.parseInt(input);

    System.out.print("Enter a number of cents: ");
    input = console.readLine();
    cents = Integer.parseInt(input);

    valueA.initialize(dollars, cents);

    System.out.println("valueA is " + valueA);
    System.out.println("valueB is " + valueB);

    sum = valueA.add(valueB);
    System.out.println("The entered value plus " +
      valueB + " is " + sum);

    System.exit(0);
  }
}
```

```
/* run #1

Enter a number of dollars: 23
Enter a number of cents: 42
valueA is $23.42
valueB is $8.50
The entered value plus $8.50 is $31.92
*/
```

```
/* run #2

Enter a number of dollars: 15
Enter a number of cents: 83
valueA is $15.83
valueB is $8.50
The entered value plus $8.50 is $24.33
*/
```

Money.java

```java
/**
 * Money will deal with money as two integer fields,
 * dollars and cents
 */
public class Money
{
  // instance variables
  private int myDollars;
  private int myCents;

  /**
   * Constructors for objects of class Money
   */
  public Money()
  {

    // initialize instance variables
    initialize(0, 0);
  }
```

```
public Money(int newDollars, int newCents)
{
  // initialize instance variables
  initialize(newDollars, newCents);
}

/**
 * Initialize (reset) an object of the Money class *
 * @param newDollars -an integer number of dollars
 * @param newCents -an integer number of cents
 * @post instance variables are set
 */
public void initialize(int newDollars, int newCents)
{
  myDollars = newDollars;
  myCents = newCents;
}
/**
 * Add two objects of the Money class
 *
 * @param amount is a Money object
 * @pre both operands (this and amount) have been
 *                     initialized
 * @return this + amount
 */
public Money add(Money amount)
{
  Money result = new Money();
  result.myCents = myCents + amount.myCents;
  result.myDollars = myDollars + amount.myDollars
              + result.myCents / 100;
  result.myCents = result.myCents % 100;
  return result;

}
```

```
/**
 * Create a string which can be used to output a
 * money object appropriately
 *
 * @pre this has been initialized
 * @return a string to display a money object
 */
public String toString()
{
      return "$" + myDollars + "." + myCents;
}
/**
 * Retrieve the value of myDollars instance
 *
 * @return myDollars
 */
public int getDollars()
{
      return myDollars;
}

/**
 * Retrieve the value of myCents
 *
 * @return myCents
 */
public int getCents()
{

return myCents;
}
}
```

Design Question Lab

Judy Hromcik
Arlington High School
Arlington, Texas

Design Question Lab (A with AB extension)

Consider the following problem. A company employs two kinds of employees: hourly wage employees and salaried employees. All employees have a name and a unique employee ID, can change their name, receive a raise, and get a paycheck every week. Hourly employees have their paycheck computed by multiplying the number of hours worked by their hourly pay rate. If an hourly employee works more than 40 hours in a weekly pay period, all hours over 40 are paid 1.5 times the hourly rate. Salaried employees receive 1/52 of their salary every week.

Obviously these two types of employees have a lot in common. Apply the "IS-A" relationship.

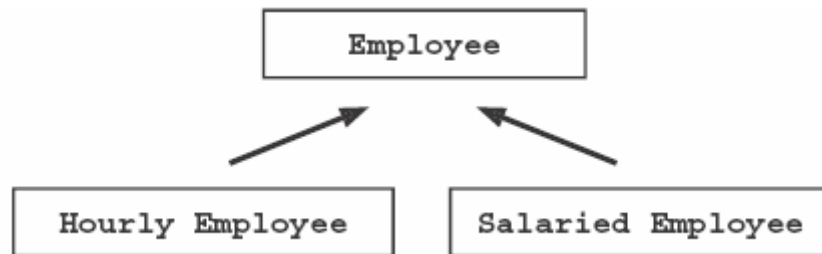An hourly employee "**IS-A**" salaried employee?

A salaried employee "**IS-A**" hourly employee?

Neither of these is true. What about:

An hourly employee "**IS-A**" employee?

A salaried employee "**IS-A**" employee?

These relationships are true. So the hierarchy should be:

But what is a plain old employee? How do you calculate an employee's pay? You can't until you know what kind of employee you have. This problem can be solved by creating `Employee` as an abstract class. In this class, the programmer will include all of the properties that the two subclasses have in common. A constructor is created to initialize those properties. The subclasses should call that constructor using a super constructor call. The abstract class will **define and implement** all of the methods that both classes have in common and that can be completed at the abstract class level. The methods that the subclasses have in common, but cannot be completed, will be defined as abstract. The subclasses **MUST** implement these methods or be defined as abstract themselves.

- Design and implement the abstract class `Employee`. All employees have a name and ID, can change their name, receive a raise, and get a paycheck every week. Add private instance variables and methods to fulfill these requirements. Provide an appropriate `toString` method for this class.
- Design and implement a non-abstract class `HourlyEmployee`. An hourly employee's pay is based on his/her hourly rate and the number of hours worked in the weekly pay period. If the hourly employee has worked overtime, work for hours over 40 is paid at 1.5 times the hourly rate. Add private instance variables and methods as necessary to implement this class. Provide an appropriate `toString` method for this class.
- Design and implement a non-abstract class `SalariedEmployee`. A salaried employee receives 1/52 of his annual salary each week. Add private instance variables and methods as necessary to implement this class. Provide an appropriate `toString` method for this class.

Now that you have designed and implemented classes for the employees of a company, consider designing a class that will manage the payroll for this company. The operations for this class must include the following:
- The payroll class must be able to add hourly employees and salaried employees to the payroll.
- The payroll class must be able to delete hourly employees and salaried employees from the payroll.
- Each week, the records for each hourly employee must be updated to reflect the number of hours that were worked during that weekly pay period. **You will need to resolve how this information (weekly hours worked) is transferred to each employee object.**
- Each week, a total payroll for the company must be computed.

AB Extension

If **H** is the number of hourly employees and **S** is the number of salaried employees, updating the hours worked for the hourly employees must run in **O(H log (H))** time. Computing each employee's paycheck and the total payroll must run in **O(H + S)** time. Choose appropriate data structures to fulfill these requirements.

Add the following requirements:

- Print employee information in the following manner:
    - Print all of the hourly employees' information alphabetically
    - Print all of the salaried employees' information alphabetically

    This operation must be done in linear time—**O(H)** for hourly employees, **O(S)** for salaried employees.

- Using an employee's ID number, access to any employee's record must be done in **O(1)** time.
- Adding and deleting employees must be done in **O(log (H + S))** time. The employee's ID will be used when deleting an employee.

Explain your chosen data structures for this class. Explain how the employee data is being stored in the `Payroll` class. Justify how your data structures meet the **Big-Oh** requirements for this problem.

Notes

1. Setting the number of hours worked for each hourly employee can be done several ways. The Payroll method could iterate through the hourly employees and read the information in from a file. It could accept a data structure with employee ID's and hours worked paired together. This is really up to the students and/or the teacher.

2. I assumed that the employee name was in the form **Last, First**. I am leaving this to the discretion of the teacher and student. You could create a Name class that encapsulates the first and last name. I chose not to do this.

3. You can find Teacher Notes for this Design Question lab and all of the code at fcbrowser.aisd.net/~jhromcik/AP/CBDesignLab.htm.

The Game of SET:
A Case Study in OO Design and Team Development

Maria Litvin
Phillips Academy
Andover, Massachusetts

Slide 1

The SET Game

- SET is a simple card game which is gaining popularity.
- http://www.setgame.com has the official rules, examples, game variations, and other resources.
- SET® is a registered trademark of SET Enterprises, Inc.

3

Slide 2

The SET Game (cont'd)

- A SET deck consists of 81 cards. A card has four attributes:
  - number of symbols (1, 2, or 3)
  - symbol shape (oval, squiggle, or diamond)
  - symbol fill (outlined, striped, or solid)
  - symbol color (red, green, or blue)
- A "set" is three cards, such that for each attribute its values for the cards are either **all** the same or **all** different.

One example of a "set"

4

Slide 3

The SET Game (cont'd)

- At the start of the game, 12 cards are open and all the players look for a "set" in them.

- The player who sees a set announces, "Set!" then promptly points to the three cards of the set.

- If the cards indeed form a set, these cards are removed and replaced with three cards from the deck, and the player gets one point; otherwise, the cards remain on the table, and the player loses one point.

- If all the players agree that the open cards don't have any sets, 3 additional cards are opened.

5

Slide 4
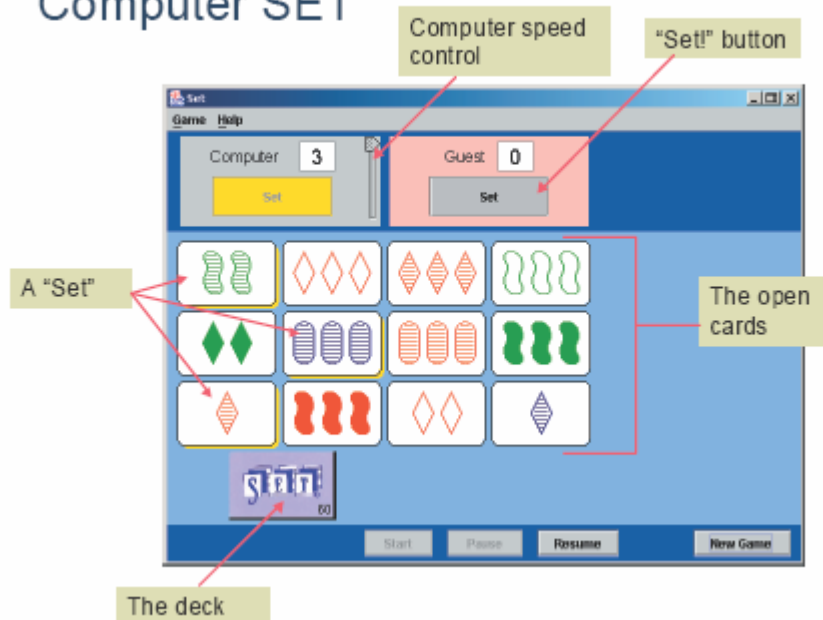
Slide 5

Identifying Classes — CRC Cards

- CRC (Class, Responsibilities, Collaborators) cards facilitate brainstorming for identifying classes in an OO project.

- A CRC card is usually an index card that lists a class, its responsibilities, and helper classes.

- CRC cards are informal; responsibilites are listed with few details (not a list of the class's methods).

7

Slide 6

CRC Cards for SET

ZetCard
- Represents a SET card
- Holds card attributes (color, fill, etc.)

ZetDeck
- Holds 81 SET cards
- Shuffles and sorts the deck
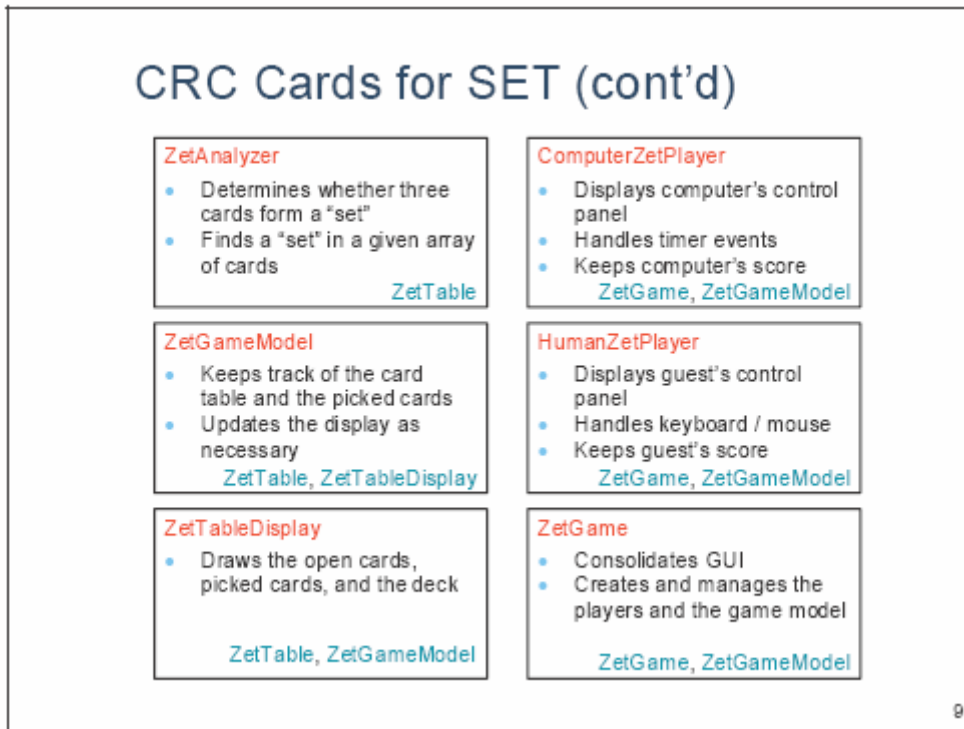- Delivers cards one at a time

ZetCard, ZetTable

ZetTable
- Holds the deck and the open cards
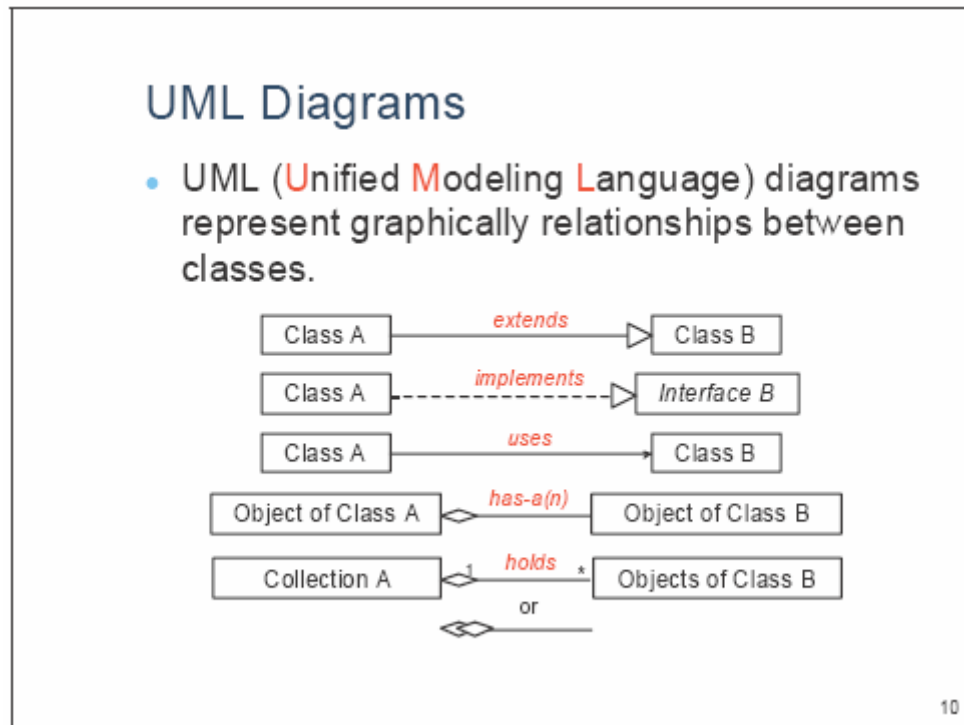- Opens and removes cards
- Looks for a "set"

ZetDeck, ZetAnalyzer

Unfortunately, the name of the game, SET, clashes with java.util.Set and with names of setter methods. To avoid confusion, in this case study we use Zet in class and method names.
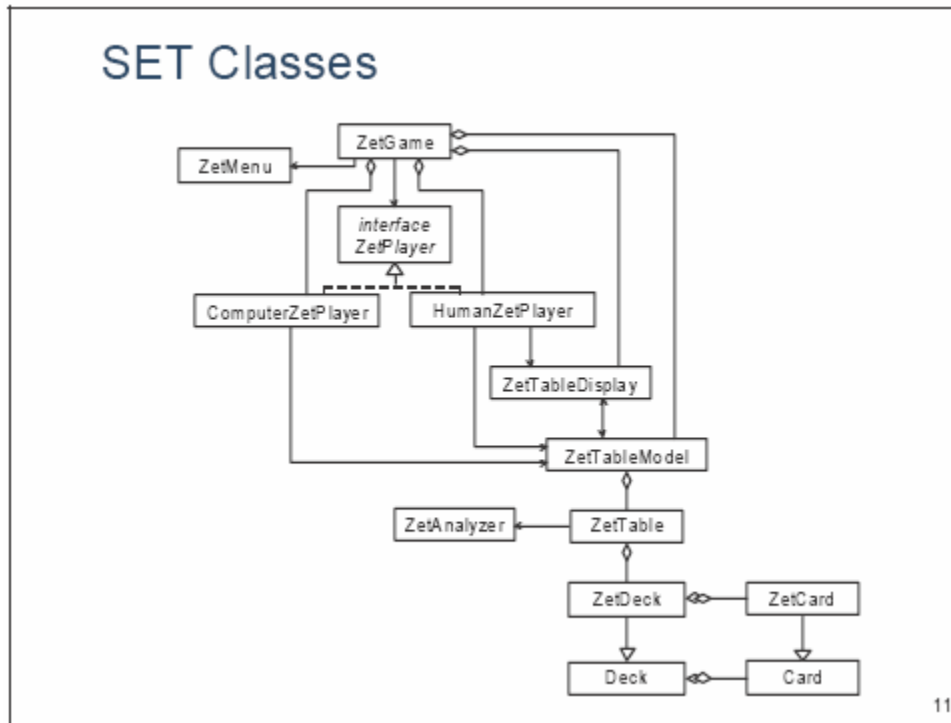
8

Slide 7

CRC Cards for SET (cont'd)

ZetAnalyzer
- Determines whether three cards form a "set"
- Finds a "set" in a given array of cards

ZetTable

ComputerZetPlayer
- Displays computer's control panel
- Handles timer events
- Keeps computer's score

ZetGame, ZetGameModel

ZetGameModel
- Keeps track of the card table and the picked cards
- Updates the display as necessary

ZetTable, ZetTableDisplay

HumanZetPlayer
- Displays guest's control panel
- Handles keyboard / mouse
- Keeps guest's score

ZetGame, ZetGameModel

ZetTableDisplay
- Draws the open cards, picked cards, and the deck

ZetTable, ZetGameModel

ZetGame
- Consolidates GUI
- Creates and manages the players and the game model

ZetGame, ZetGameModel

9

Slide 8

UML Diagrams

- UML (Unified Modeling Language) diagrams represent graphically relationships between classes.

Class A	*extends*	Class B
Class A	*implements*	Interface B
Class A	*uses*	Class B
Object of Class A	*has-a(n)*	Object of Class B
Collection A	*holds*	Objects of Class B

or

10

Slide 9

Slide 10

Slide 11

Encapsulate classes; make classes interact via well-defined public constructors and methods.
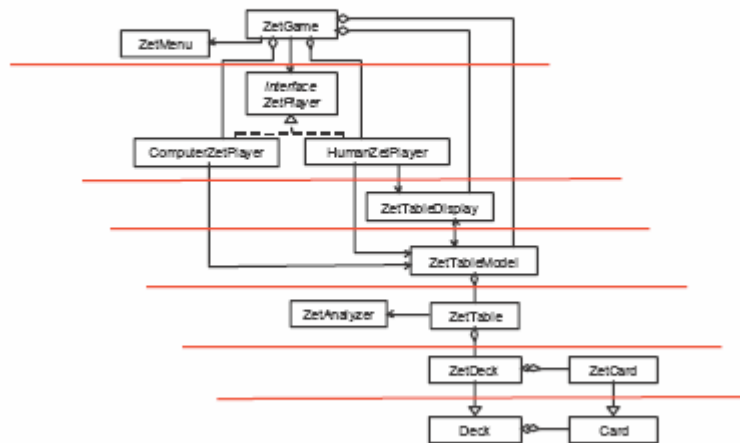
```
public class ComputerZetPlayer extends JPanel implements ZetPlayer, ...
{
  public ComputerZetPlayer (ZetGame game,
                   ZetGameModel gameModel) { ... }

  public void start ( ) { ... }
  public void stop ( ) { ... }
  public int getScore( ) { ... }
  public void setScore(int score) { ... }

  ...
  private void declareZet ( ) { ... }

  private int state;
  private int delay;
  private int score = 0;

  ...
}
```
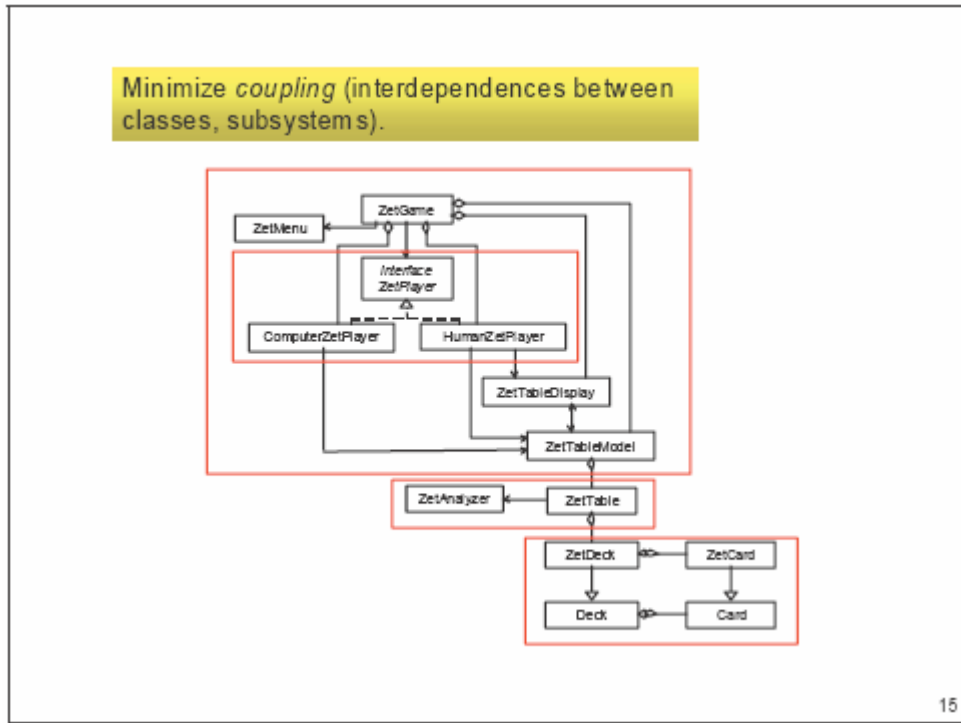
13

Slide 12

Arrange classes in layers: classes in the top layers utilize simpler, more general classes from the bottom layers.
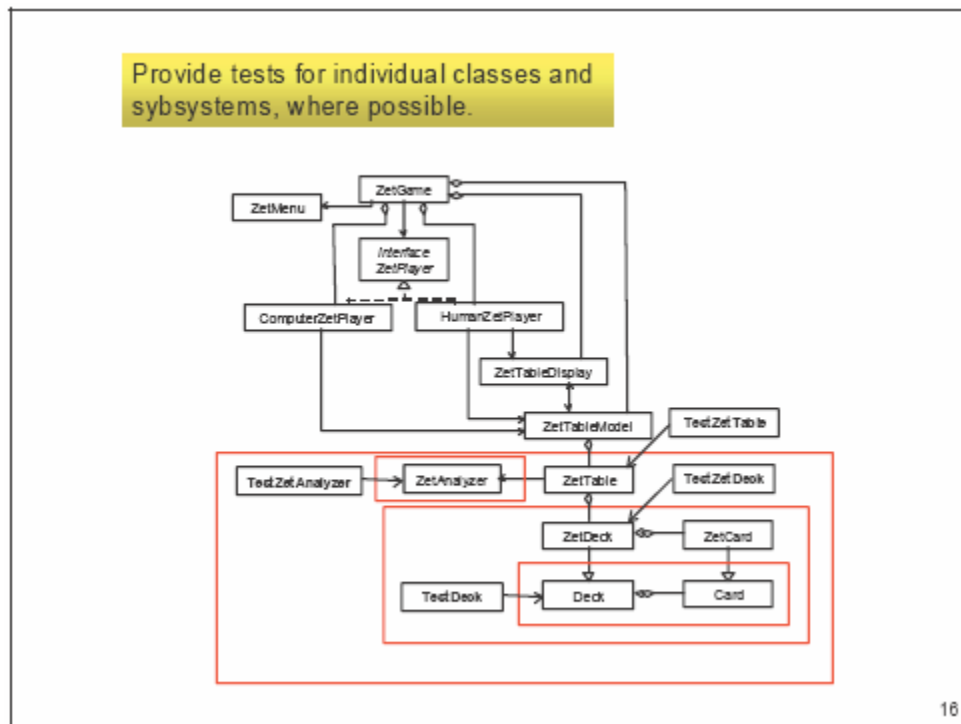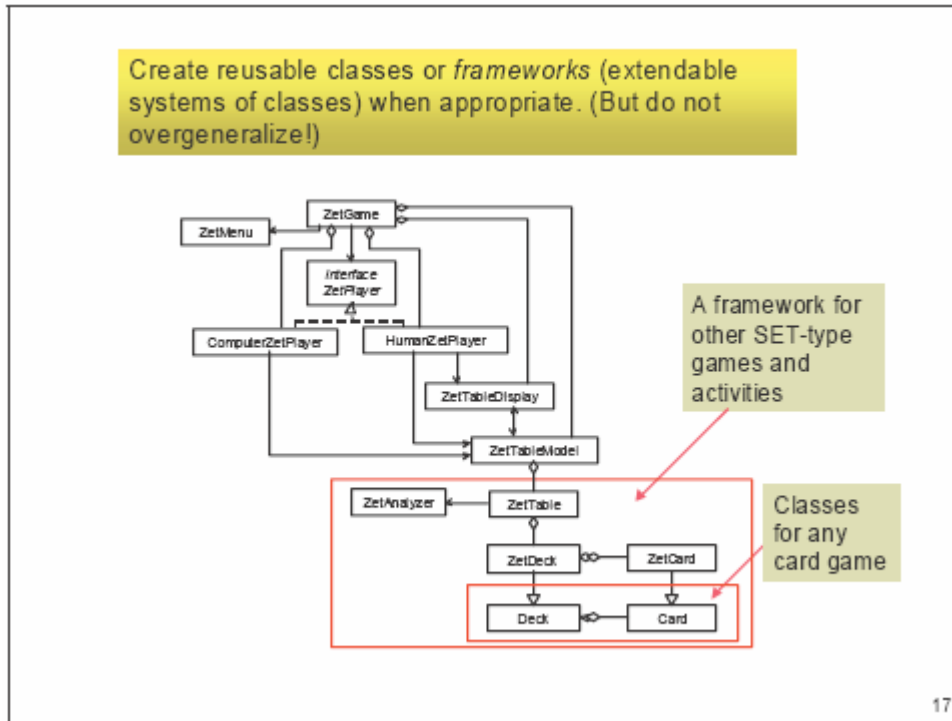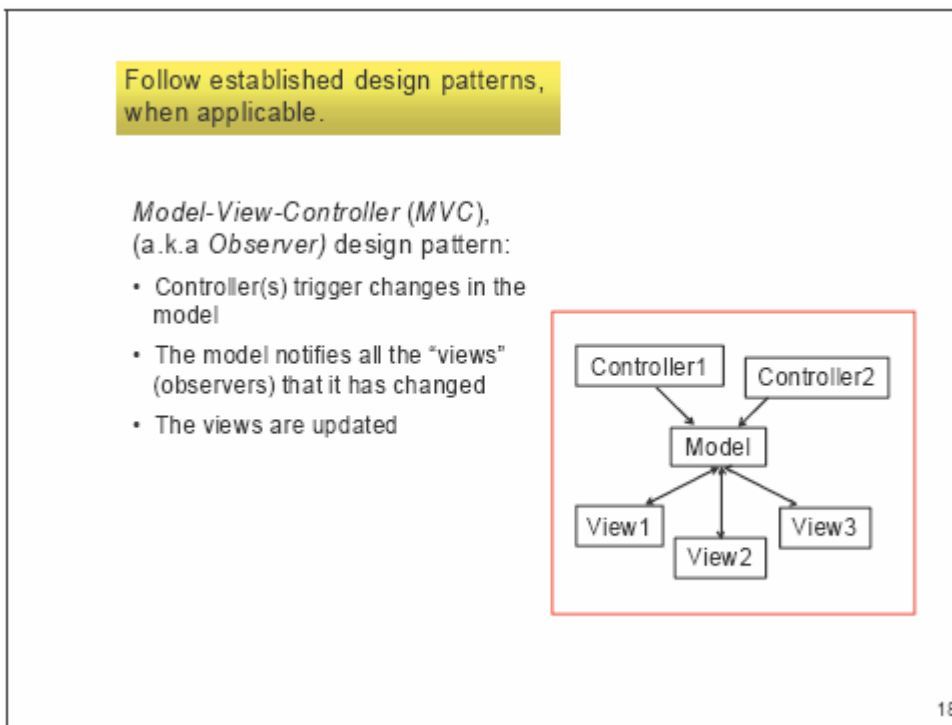
14

Slide 13

Minimize *coupling* (interdependences between classes, subsystems).

Slide 14

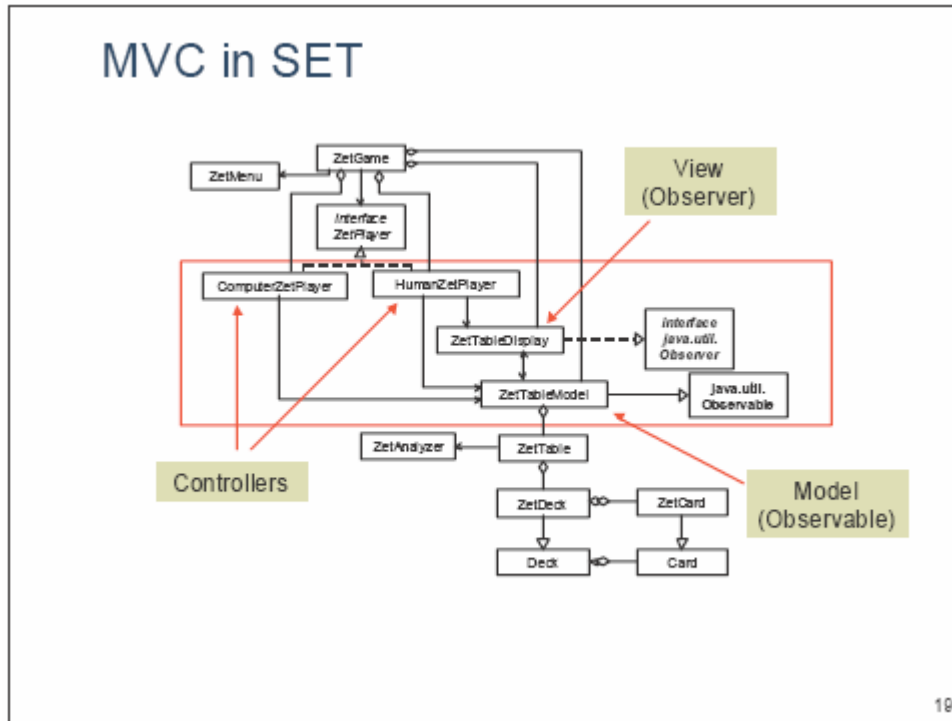Provide tests for individual classes and sybsystems, where possible.

Slide 15

Slide 16

Slide 17

Slide 18
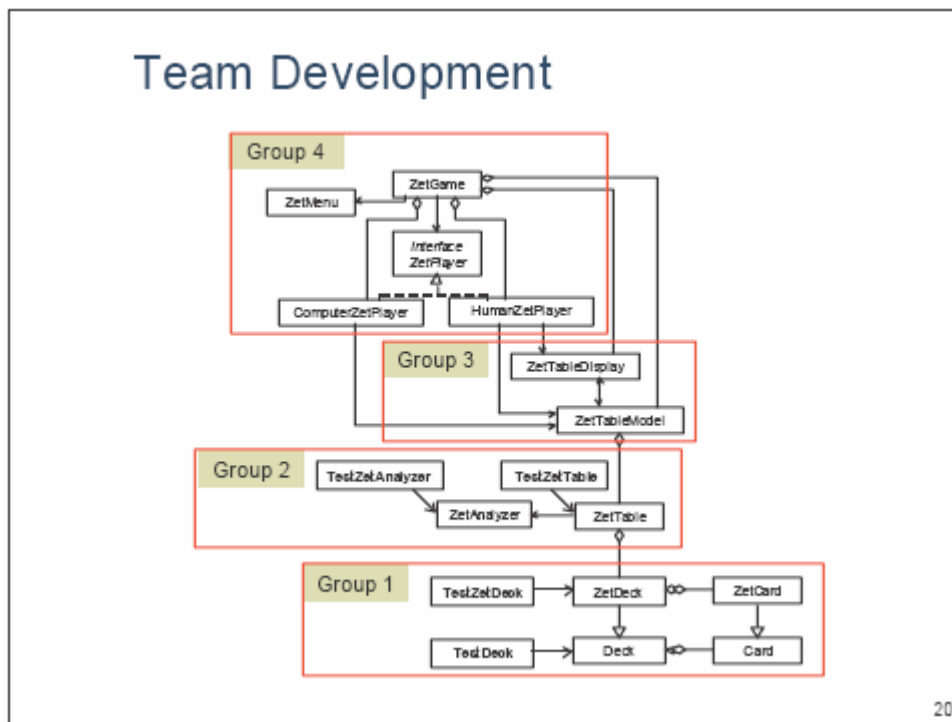
Slide 19

Slide 20

Slide 21

Slide 22

Slide 23

Slide 24

Slide 25


```
                                        Group 1  Task 1-a■

    public class Card
        implements Comparable

    {
       public Card (int id) { ... }

       public int getId ( ) { ... }
       public boolean equals (Object other) { ... }
       public int compareTo (Object other) { ... }
       public String toString ( ) { ... }

       // Fields:
       private int id;
    }

                                                      27
```

Slide 26


```
                                        Group 1  Task 1-b■■

    public class Deck

    {
       public Deck ( ) { ... }
       public Deck (int capacity) { ... }    // creates an empty deck
                                             //   of given capacity
       public int getNumCards ( ) { ... }
       public boolean isEmpty ( ) { ... }
       public int add (Card card) { ... }    // adds card to the top
       public Card takeTop ( ) { ... }       // removes card from the top
       public void shuffle ( );
       public void sort ( );
       public String toString ( ) { ... }

       // Fields:
       ...                          See implementation tips in
    }                                    Deck.java

                                                      28
```

Slide 27

Group 1 Task 1-c▪

public class TestDeck

• Create an empty deck
• Add a few cards
• Print out
• Shuffle
• Print out
• Sort
• Print out
• Remove cards one by one;
  print out after each removed card

29

Slide 28

Group 1 Task 2-a▪

public class ZetCard
        extends Card

```
{
   // Combines the four attributes to make a
   //   unique ID in the range from 0 to 80.
   public ZetCard (int number, int shape,
                              int fill, int color) { ... }

   public int getNumber ( ) { ... }
   public int getShape ( ) { ... }
   public int getFill ( ) { ... }
   public int getColor ( ) { ... }
   public String toString ( ) { ... }

   // Fields:
   ...
}
```

30

Slide 29

Group 1 Task 2-b•

public class ZetDeck

{
    public ZetDeck ( ) { ... }   // creates a full deck of
                                    //   81 SET cards
}

Group 1 Task 2-c•

public class TestZetDeck

• Create a ZetDeck
• Remove and print out three top cards

31

Slide 30

Group 2

Task 1 ▪▪ (1-2 people)

TestZetAnalyzer

ZetAnalyzer

Task 2 ▪▪▪ (1-2 people)

TestZetTable

ZetTable

32
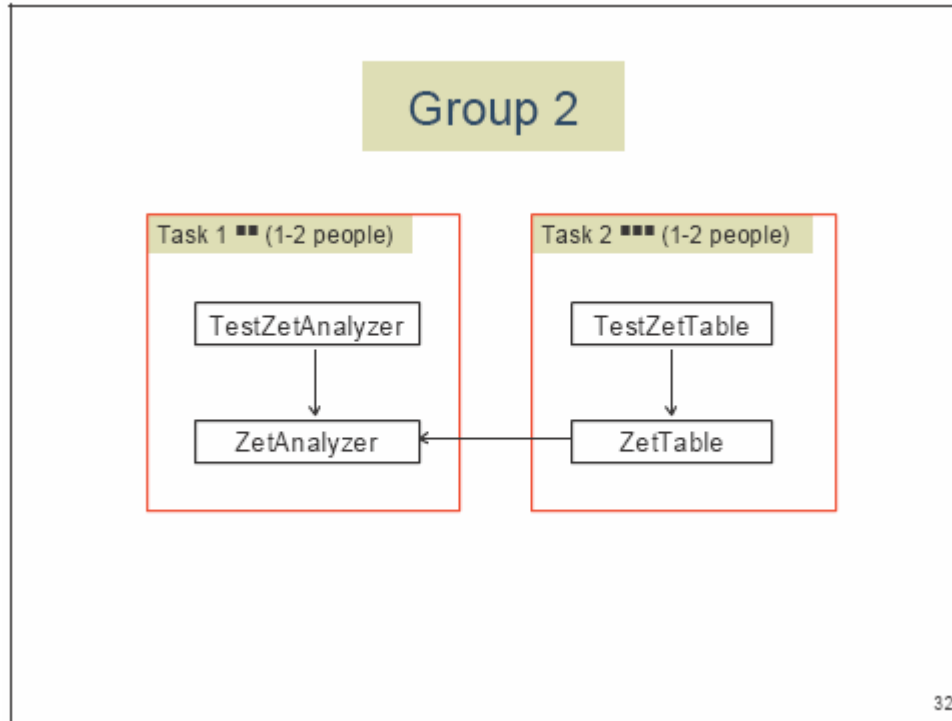
Slide 31

Group 2 Task 1-a■■

```
public class ZetAnalyzer

{
    public static boolean isZet (ZetCard card1,
                ZetCard card2, ZetCard card3) { ... }

    public static int[ ] findZet (ZetCard[ ] cards) { ... }
}
```

See implementation tips in
ZetAnalyzer.java

33

Slide 32

Group 2 Task 1-b■■

public class TestZetAnalyzer

• Create a ZetDeck
• Open and print out a few cards
• Find and print out all "sets" by calling isZet on all
  triplets of cards
• Find and print out one "set" by calling findZet

34

Slide 33

Slide 34

Slide 35

When everyone has finished...

- The project leader collects the code from the groups.
- The project leader adds the supplied classes:
  - ZetGame
  - ZetMenu
  - ZetGameModel
  - ZetTableDisplay
  - ZetPlayer
  - ZetComputerPlayer
  - ZetHumanPlayer
- The QA (Quality Assurance) team (everyone) tests the application.

37

Slide 36

Summary

- OOP helped us split the project into small classes with well-defined limited responsibilities.
- Encapsulation helped us minimize documentation and interactions between developers.

38

Slide 37

Summary (cont'd)

- Abstraction helped us make some of the classes (Card, Deck) reusable and to create a framework (ZetCard, ZetDeck, ZetAnalyzer, ZetTable) for other SET-type games and activities.

- Inheritance helped us reuse the code from Deck in ZetDeck and made it easier to add different types of players.

39

Slide 38

- These slides and the *Game of SET* case study code are posted at:

  http://www.skylit.com/oop/

- See **InstructorNotes.doc** for suggestions on how to run this project with your students.

- e-mail questions and comments about this project to:

  mlitvin@andover.edu

40

Notes for the Instructor

Objective: This exercise gives students a taste of how object-oriented design and programming might work in the "real world." The emphasis is on project design and on working as a team, not on writing Java code.

Time requirement: 2-3 hours to include the preliminary discussion of OO design principles, this project's design, coding, and lessons learned.

Team composition: 4-10 people. If necessary, several independent teams can work in parallel.

Prerequisites: The members of the team may have varying levels of technical proficiency. All participants must have some understanding of classes, objects, constructors and methods. At least one participant (Group 1 Task 1-b) should be familiar with `java.util.ArrayList` or be able to figure out how to use its `add` and `get`, and `remove` methods. The same person must be familiar with Selection Sort and a similar shuffling algorithm, or use the `Collections.sort` and `Collections.shuffle` library methods. Group 2 Task 1-a requires basic familiarity with modulo 3 arithmetic, although the programmer can get around it with more verbose code. Group 2 Task 2-a requires a fast coder who can write several methods quickly and is familiar with or can figure out a partitioning algorithm for an array (similar to one used in Quicksort).

In order to facilitate the appropriate allocation of tasks among team members, the difficulty of each task is labeled with ■, ■■, or ■■■. "Shell" Java files with method headers and development tips are provided for the more difficult tasks. The instructor can control the difficulty of the tasks by providing some code from the solution.

The project can accommodate more experienced programmers (who are familiar with graphics and MVC) by delegating the Group 3 tasks to students instead of using the provided code.

Project administration: At first, all students participate together in the initial discussion of OO design principles and project design. After that, a team is formed for implementing the code. The instructor appoints one project leader and two group leaders, and splits the rest of the students between Group 1 and Group 2. The project leader and the group leaders must be enthusiastic, have strong organizational skills, and be fluent in using an IDE and putting projects together. The group leaders can also participate as developers on some tasks. If necessary, the instructor can act as the project leader.

Teamwork: The two groups work independently of each other, writing code and testing it independently when possible. In this project, Group 2 has to use the classes developed by Group 1 for testing. Leaders of Group 1 and Group 2 test the finished code before passing it to the project leader. The members of each group work independently on their tasks. However, the group leaders must monitor their group's progress and, if necessary, mobilize those who completed their tasks to help other group members. This project is about teamwork, not competition.

Before the project: Review the main OO design principles, discuss the project design and its decomposition into independent tasks, and demonstrate the *Set Game* application.

Slide handouts: The project leader should get copies of slides 20-36. The Group 1 leader gets copies of slides 26-31 and the Group 2 leader gets copies of slides 32-36. All participants should get a copy of slide 20, or have it displayed for the whole class.

Docs handouts: All students get javadoc-generated documentation for the classes the team will be working on. It is available in the docs folder and accessible through docs\index.html.

Code handouts: The project leader gets the following files:
```
ZetGame.java
ZetMenu.java
ZetPlayer.java
ComputerZetPlayer.java
HumanZetPlayer.java
ZetTableDisplay.java
ZetGameModel.java
deck.jpg
```

The Group 1 leader gets the tips and documentation file for Task 1-b, `Group1Tips\Deck.java`.

The Group 2 leader gets the tips and documentation files for Tasks 1-a and 2-a: `Group2Tips\ZetAnalyzer.java` and `Group2Tips\ZetTable.java`, respectively.

During the project: Monitor the progress of each group and provide assistance when necessary. Developers who have finished their task can help others in their group. Insist that each task developer tests his or her code independently and that each group tests the code before having it integrated into the final project.

After the project: Test the program thoroughly. Make each group and its members present each task and the code written for it. Review how the key OOP concepts are used in this project.

Feedback: We will appreciate any comments and suggestions for improving this project. Please email mlitvin@andover.edu.

Marine Biology Simulation:
The Strategy Pattern Applied to the Fish Class

Chris Nevison
Colgate University
Hamilton, New York

Introduction

Design patterns can be a helpful guide to organizing a program. A pattern is simply a description of a way of organizing the objects and classes in the design of a program, which can be adapted to other programs. A difficulty with the Fish class is that if we want to use inheritance to create subclasses of Fish that have different ways of moving (original three-way movement, darter movement, slow three-way movement), different ways of breeding (original all-or-none, independent, one-only) and, perhaps, different ways of dying (original, based on probability, or age-based), and we want different combinations of these, then the inheritance hierarchy can become quite large and convoluted, as shown below.
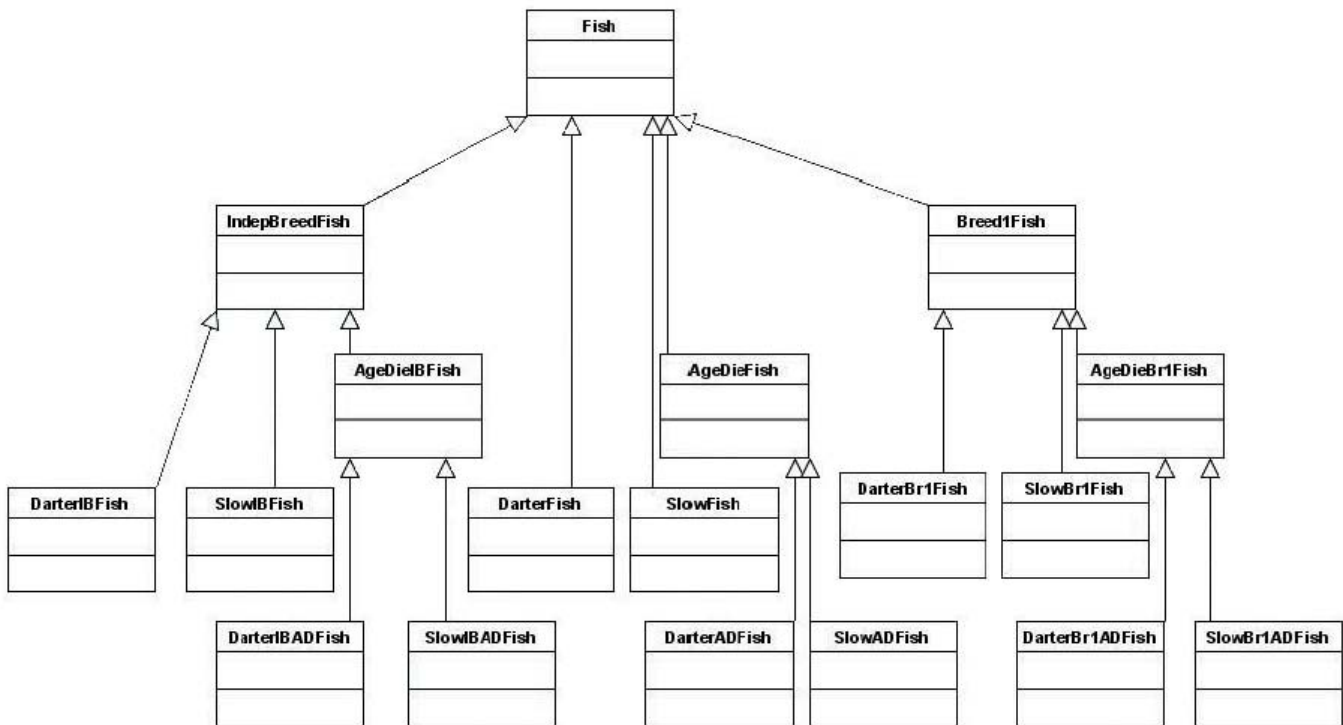
The Strategy Pattern

The strategy design pattern suggests that when a class describes objects that could implement a particular responsibility in different ways using different strategies (algorithms), then rather than create modifications through inheritance, we can regard the strategy as an object (Gamma et al., 1995). For example, the Fish class has three responsibilities: moving, breeding, and determining whether it should die. Rather than create subclasses of Fish that define different ways of moving, breeding, and/or dying, we can define the Fish class as having an instance variable, set by the constructors, for each strategy.

This is a natural place to use an interface or an abstract class. We define each strategy type using an interface (or abstract class), and then the implementing classes implement that interface. If we work with the three movements defined in the MBS and consider two ways of breeding—the original all-or-none breeding (where, based on a probability, the fish breeds in every empty neighbor or does not breed) and a different "independent" breeding (where each empty neighbor is checked independently, and, with a given probability, the fish breeds into that location or not)—then we can diagram the situation as follows (we leave the issue of dying out of this example).
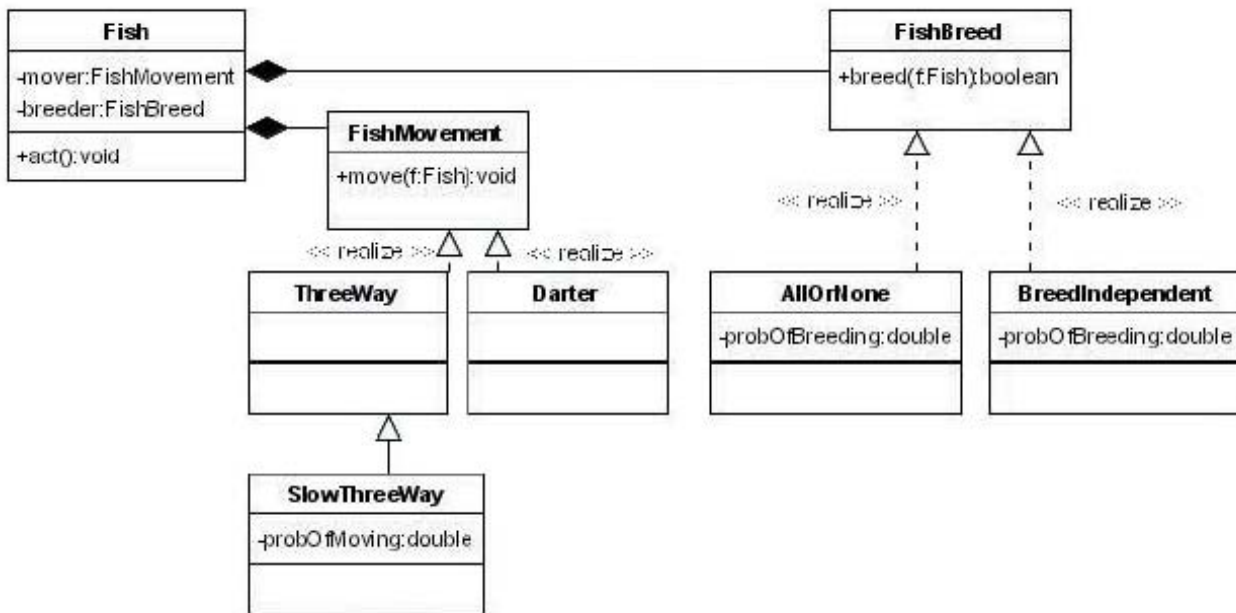
In the following lab, we ask that the student implement the Fish class in this way.

Lab Setup

You will need to do the following.

1. Create interfaces `FishMovement` and `FishBreed`. The movement interface should specify a method move that takes a `Fish` as parameter. The breed interface should specify a method breed that takes a `Fish` as parameter.

2. Modify the `Fish` class so that it has an instance variable for each of the interfaces given above and so that all the constructors set those fields from parameters. You will delete most of the methods involved in breeding and moving from the `Fish` class, but you should keep a file with a copy of this class from which to use those methods.

3. The class `ThreeWay` will implement the original fish movement strategy. It would have a move method, which is essentially the same as the original move method, except that it uses its `Fish` parameter to reference the environment, location, and other methods of the `Fish` that are used in moving. It should also copy the `nextLocation` method with the same changes (giving the method a `Fish` parameter). The `emptyNeighbors` method that is used by the `nextLocation` method presents an interesting design decision. If we place it in the `ThreeWay` class, then we will also need to make a copy in the breeding methods. What alternatives are there? Where should the `emptyNeighbors` method be placed? The methods `changeLocation` and `changeDirection` also present a design decision. They will be used by every class that implements `FishMovement`. Should they remain in the `Fish` class, should they be placed in the abstract `FishMovement` (making it an abstract class instead of an interface), or should they be copied into every class that implements `FishMovement`?

4. The class `AllOrNone` will implement the original breeding method. It will include an instance variable to store the probability of breeding and will implement the breed method breed (taking a `Fish` parameter). Again, we need access to the `emptyNeighbors` method, so the question is: Where should this method be placed or copied? For breeding, another design decision is where we should place the `generateChild` method, so that it breeds true—that is, the children fish have the same characteristics as their parents.

5. Delete the move and breed methods from this `Fish` class, as well as any auxiliary methods that are placed in the classes implementing `FishMovement` and `FishBreed` (and the probability of breeding, which will now be placed in the breeding classes). The following is a partial definition of the new `Fish` class.

```
public class Fish implements Locatable
{
  ...
  private FishMovement mover;
  private FishBreed breeder;

  public Fish(Environment env, Location loc, FishMovement mv,
FishBreed brd)
  ...

  public void act()
  {
    if(!isInEnv())
      return;

    if(!breeder.breed(this))
      move.move(this);

    ...

}
```

6. Rewrite the class `SimpleMBSDemo2` to test the new version of the `Fish` class. Each fish that is declared will need to have a `FishMovement` object and a `FishBreed` object passed as a parameter to the constructor, the latter with a probability passed to its constructor. If you use a constructor that allows you to set color, you can use that to distinguish the different types of fish for testing.

7. Now that you have the basic fish created using this new organization, we can add new ways of breeding and dying. Write two movement classes: a `DarterMovement`, implementing the `FishMovement` interface and a `SlowThreeWay`, which extends `ThreeWay`. These should implement the move method in a similar fashion to the way that `DarterFish` and `SlowFish` do, using the `Fish` parameter to access needed information about the fish. Also, write a new breeding class— `IndependentBreed`— which implements the `FishBreed` interface. `IndependentBreed` should take a probability in its constructor, and when its breed method is called, it should check each empty neighboring location and breed into the location with the given probability (so that any number of fish, from zero to the number of empty neighbors, may be created). Using your new classes, modify `SimpleMBSDemo2` to create fish with different combinations of moving and breeding.

Reference

Gamma, Erich, Richard Helm, and Ralph Jonhson. 1995. *Design Patterns*. Edited by John Vlissides. Addison-Wesley: New York.

All-or-None Breeder

```java
import java.util.ArrayList;
import java.util.Random;

public class AllOrNoneBreeder implements FishBreeder
{
  private double probBreed;

  public AllOrNoneBreeder(double pb)
  {
   probBreed = pb;
  }

  protected double probBreed()
  {
   return probBreed;
  }

public boolean breed(Fish f)
{
  // Determine whether this fish will try to breed in this
  // timestep. If not, return immediately.
  Random randNumGen = RandNumGenerator.getInstance();
  if ( randNumGen.nextDouble() >= probBreed() )
    return false;

  // Get list of neighboring empty locations.
  ArrayList emptyNbrs = f.emptyNeighbors();
  Debug.print("Fish " + f.toString() + " attempting to breed. ");
  Debug.println("Has neighboring locations: " + emptyNbrs.
toString());

  // If there is nowhere to breed, then we're done.
  if ( emptyNbrs.size() == 0 )
  {
    Debug.println(" Did not breed.");
    return false;
   }
    // Breed to all of the empty neighboring locations.
    for ( int index = 0; index < emptyNbrs.size(); index++ )
    {
      Location loc = (Location) emptyNbrs.get(index);
      Fish child = f.generateChild(loc);
      Debug.println(" New Fish created: " + child.toString());
    }

    return true;
  }
}
```

Darter Movement

```java
import java.util.ArrayList;
import java.util.Random;

public class DarterMovement implements FishMovement
{
  public void move(Fish f)
  {

    // Find a location to move to.
    Debug.print("DarterFish " + f.toString() + " attempting to
move. ");
    Location nextLoc = nextLocation(f);

    // If the next location is different, move there.
    if ( ! nextLoc.equals(f.location()) )
    {

      f.changeLocation(nextLoc);
      Debug.println(" Moves to " + f.location());
    }
    else
    {
      // Otherwise, reverse direction.
      f.changeDirection(f.direction().reverse());
      Debug.println(" Now facing " + f.direction());
    }
  }

/** Finds this fish's next location.
 * A darter fish darts forward two spaces if it can, otherwise it
 * tries to move forward one space. A darter fish can only move
 * to empty locations, and it can only move two spaces forward if
 * the intervening space is empty. If the darter fish cannot move
 * forward, <code>nextLocation</code> returns the fish's current
 * location.
 * @return the next location for this fish
 **/
protected Location nextLocation(Fish f)
{
  Environment env = f.environment();
  Location oneInFront = env.getNeighbor(f.location(), f.direction());
  Location twoInFront = env.getNeighbor(oneInFront, f.direction());
  Debug.println(" Location in front is empty? " +
          env.isEmpty(oneInFront));
  Debug.println(" Location in front of that is empty? " +
          env.isEmpty(twoInFront));
  if ( env.isEmpty(oneInFront) )
  {
    if ( env.isEmpty(twoInFront) )
      return twoInFront;
    else
      return oneInFront;
}
```

Fish

```
// AP(r) Computer Science Marine Biology Simulation:
// The Fish class is copyright(c) 2002 College Entrance
// Examination Board (www.collegeboard.com).
//
// This class is free software; you can redistribute it and/or
// modify it under the terms of the GNU General Public License as
// published by the Free Software Foundation.
//
// This class is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.

import java.awt.Color;
import java.util.ArrayList;
import java.util.Random;

/**
 * AP&reg; Computer Science Marine Biology Simulation:<br>
 * A <code>Fish</code> object represents a fish in the Marine
 * Biology Simulation. Each fish has a unique ID, which remains
 * constant throughout its life. A fish also maintains information
 * about its location and direction in the environment.
 *
 * <p>
 * Modification History:
 * -Modified to support a dynamic population in the environment:
 * fish can now breed and die.
 * -Modified to use strategy pattern for moving and breeding
 *
 * <p>
 * The <code>Fish</code> class is
 * copyright&copy; 2002 College Entrance Examination Board
 * (www.collegeboard.com).
 *
 * @author Alyce Brady
 * @author APCS Development Committee
 * @author Modification for strategy pattern, Chris Nevison
 * @version 3 April 2004
 * @see Environment
 * @see Direction
 * @see Location
 **/
```

```
public class Fish implements Locatable
{
  // Class Variable: Shared among ALL fish
  private static int nextAvailableID = 1; // next avail unique
identifier

  // Instance Variables: Encapsulated data for EACH fish
  private Environment theEnv; // environment in which the fish lives
  private int myId; // unique ID for this fish
  private Location myLoc; // fish's location
  private Direction myDir; // fish's direction
  private Color myColor; // fish's color

  private double probOfDying; // defines likelihood in each timestep

  private FishMovement mover;
  private FishBreeder breeder;

  // constructors and related helper methods

  public Fish(Environment env, FishMovement mover,
              FishBreeder breeder,
              Location loc)

  {
    initialize(env, mover, breeder,
         loc, env.randomDirection(), randomColor());
  }

  public Fish(Environment env, FishMovement mover,
              FishBreeder breeder,
              Location loc, Direction dir)

  {
    initialize(env, mover, breeder,
         loc, dir, randomColor());
  }

  public Fish(Environment env, FishMovement mover,
              FishBreeder breeder,
              Location loc,
              Direction dir, Color col)

  {
    initialize(env, mover, breeder,
         loc, dir, col);
  }
```

```
  /** Initializes the state of this fish.
   * (Precondition: parameters are non-null; <code>loc</code> is valid
   * for <code>env</code>.)
   * @param env  environment in which this fish will live
   * @param loc  location of this fish in <code>env</code>
   * @param dir  direction this fish is facing
   * @param col  color of this fish
   **/
  private void initialize(Environment env, FishMovement mover,
                          FishBreeder breeder,
               Location loc, Direction dir,
               Color col)
  {
    theEnv = env;
    myId = nextAvailableID;
    nextAvailableID++;
    myLoc = loc;
    myDir = dir;
    myColor = col;
    this.mover = mover;
    this.breeder = breeder;
    theEnv.add(this);

    // object is at location myLoc in environment

    probOfDying = 1.0/5.0; // 1 in 5 chance in each timestep
  }

  /** Generates a random color.
   * @return the new random color
   **/
  protected Color randomColor()
  {
    // There are 256 possibilities for the red, green, and blue
    attributes
    // of a color. Generate random values for each color attribute.
    Random randNumGen = RandNumGenerator.getInstance();
    return new Color(randNumGen.nextInt(256), // amount of red
            randNumGen.nextInt(256),   // amount of green
            randNumGen.nextInt(256));   // amount of blue
  }

 // accessor methods

  /** Returns this fish's ID.
   * @return the unique ID for this fish
   **/
  public int id()
  {
return myId;
  }
```

```
/** Returns this fish's environment.
 * @return the environment in which this fish lives
 **/
public Environment environment()
{
  return theEnv;
}

/** Returns this fish's color.
 * @return the color of this fish
 **/
public Color color()
{
  return myColor;
}

/** Returns this fish's location.
 * @return the location of this fish in the environment
 **/
public Location location()
{
  return myLoc;
}

/** Returns this fish's direction.
 * @return the direction in which this fish is facing
 **/
public Direction direction()
{
  return myDir;
}

/** Checks whether this fish is in an environment.
 * @return <code>true</code> if the fish is in the environment
 * (and at the correct location); <code>false</code> otherwise
 **/
public boolean isInEnv()
{
  return environment().objectAt(location()) == this;
}

/** Returns a string representing key information about this fish.
 * @return a string indicating the fish's ID, location, and direction
 **/
public String toString()
{
  return id() + location().toString() + direction().toString();
}
```

```
// modifier method

/** Acts for one step in the simulation.
 **/
public void act()
{

  // Make sure fish is alive and well in the environment -fish
  // that have been removed from the environment shouldn't act.
  if ( ! isInEnv() )
    return;

  // Try to breed.
  if ( ! breeder.breed(this) )
    // Did not breed, so try to move.
    mover.move(this);

  // Determine whether this fish will die in this timestep.
  Random randNumGen = RandNumGenerator.getInstance();
  if ( randNumGen.nextDouble() < probOfDying )
    die();
}

/** Finds empty locations adjacent to this fish.
 * @return an ArrayList containing neighboring empty locations
 **/
protected ArrayList emptyNeighbors()
{

  // Get all the neighbors of this fish, empty or not.
  ArrayList nbrs = environment().neighborsOf(location());

  // Figure out which neighbors are empty and add those to a new
  // list.
  ArrayList emptyNbrs = new ArrayList();
  for ( int index = 0; index < nbrs.size(); index++ )
  {
    Location loc = (Location) nbrs.get(index);
    if ( environment().isEmpty(loc) )
      emptyNbrs.add(loc);
  }
  return emptyNbrs;
}

/** Modifies this fish's location and notifies the environment.
 * @param newLoc new location value
 **/
public void changeLocation(Location newLoc)
{
  // Change location and notify the environment.
  Location oldLoc = location();
  myLoc = newLoc;
  environment().recordMove(this, oldLoc);

  // object is again at location myLoc in environment
}
```

```
/** Modifies this fish's direction.
 * @param newDir new direction value
 **/
public void changeDirection(Direction newDir)
{
  // Change direction.
   myDir = newDir;
}


/** Creates a new fish with the color of its parent.
 * @param loc location of the new fish
 **/
public Fish generateChild(Location loc)

{
  // Create new fish, which adds itself to the environment.
  return new Fish(environment(), mover,
                  breeder,
          loc, environment().randomDirection(), color());
}


/** Removes this fish from the environment.
 **/
protected void die()
{
  Debug.println(toString() + " about to die.");
  environment().remove(this);
}
}
```

Fish Breeder

```
public interface FishBreeder
{
  boolean breed(Fish fish);
}
```

Fish Movement

```
public interface FishMovement
{
  void move(Fish fish);
}
```

Independent Breeder

```java
import java.util.ArrayList;
import java.util.Random;

public class IndependentBreeder implements FishBreeder
{
  private double probBreed;

  public IndependentBreeder(double pb)
  {
    probBreed = pb;
  }

  protected double probBreed()
  {
    return probBreed;
  }

  public boolean breed(Fish f)
  {
    Random randNumGen = RandNumGenerator.getInstance();

    // Get list of neighboring empty locations.
    ArrayList emptyNbrs = f.emptyNeighbors();
    Debug.print("Fish " + f.toString() + " attempting to breed. ");
    Debug.println("Has neighboring locations: " + emptyNbrs.
toString());

  // If there is nowhere to breed, then we're done.
  if ( emptyNbrs.size() == 0 )
  {
    Debug.println(" Did not breed.");
    return false;
  }

  // Try to breed to all of the empty neighboring locations.
  boolean result = false;
  for ( int index = 0; index < emptyNbrs.size(); index++ )
    {
      if(randNumGen.nextDouble() < probBreed())
      {
        Location loc = (Location) emptyNbrs.get(index);
        Fish child = f.generateChild(loc);
        Debug.println(" New Fish created: " + child.toString());
        result = true;
      }
    }
    return result;
  }
}
```

Simple MBS Demo 2

```
// AP(r) Computer Science Marine Biology Simulation:
// The SimpleMBSDemo2 class is copyright(c) 2002 College Entrance
// Examination Board (www.collegeboard.com).
//
// This class is free software; you can redistribute it and/or modify
// it under the terms of the GNU General Public License as published by
// the Free Software Foundation.
//
// This class is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
// GNU General Public License for more details.

/**
 * AP&reg; Computer Science Marine Biology Simulation:<br>
 * The <code>SimpleMBSDemo2</code> class provides a main method that
creates
 * a simulation of a number of fish swimming in a bounded environment.
 * It also creates a simple window in which to view the environment
 * after each timestep in the simulation. This version of the MBS demo
uses
 * an object of the <code>Simulation</code> class.
 *
 * <p>
 * This class will NOT be tested on the Advanced Placement exam.
 * Modified to use Fish built with the strategy pattern.
 *
 * <p>
 * The <code>SimpleMBSDemo2</code> class is
 * copyright&copy; 2002 College Entrance Examination Board
 * (www.collegeboard.com).
 *
 * @author Alyce Brady
 * @author Chris Nevison
 * @version 3 April, 2004
 **/

 public class SimpleMBSDemo2
 {
    // Specify number of rows and columns in environment.
    private static final int ENV _ ROWS = 30; // rows in environment
    private static final int ENV _ COLS = 30; // columns in environment

    // Specify how many timesteps to run the simulation.
    private static final int NUM _ STEPS = 500; // number of timesteps

    // Specify the time delay for each step
    private static final int DELAY = 1000; // delay in milliseconds
```

```
/** Start the Marine Biology Simulation program.
 * The String arguments (args) are not used in this application.
 **/
public static void main(String[] args)
{
  // Construct an empty environment and several fish in the context
  // of that environment.
  BoundedEnv env = new BoundedEnv(ENV _ ROWS, ENV _ COLS);
  Fish f1 = new Fish(env, new ThreewayMovement(),
             new AllOrNoneBreeder(0.05),
           new Location(2, 2));
  Fish f2 = new Fish(env, new DarterMovement(),
             new AllOrNoneBreeder(0.05),
           new Location(12, 2));
  Fish f3 = new Fish(env, new SlowThreewayMovement(0.1),
             new AllOrNoneBreeder(0.05),
           new Location(8, 2));
  Fish f4 = new Fish(env, new ThreewayMovement(),
           new AllOrNoneBreeder(0.05),
             new Location(10, 3));
  Fish f5 = new Fish(env, new ThreewayMovement(),
             new IndependentBreeder(0.05),
           new Location(4, 6));
  Fish f6 = new Fish(env, new SlowThreewayMovement(0.1),
             new IndependentBreeder(0.05),
           new Location(6, 6));
  Fish f7 = new Fish(env, new DarterMovement(),
             new AllOrNoneBreeder(0.05),
           new Location(8, 10));
  Fish f8 = new Fish(env, new DarterMovement(),
             new AllOrNoneBreeder(0.05),
           new Location(10, 12));
  Fish f9 = new Fish(env, new DarterMovement(),
             new IndependentBreeder(0.05),
           new Location(15, 2));
  Fish f10 = new Fish(env, new DarterMovement(),
             new IndependentBreeder(0.05),
           new Location(15, 3));
  Fish f11 = new Fish(env, new TurningDarterMovement(0.1),
             new AllOrNoneBreeder(0.05),
           new Location(9, 13));
  Fish f12 = new Fish(env, new TurningDarterMovement(0.1),
             new AllOrNoneBreeder(0.05),
           new Location(11, 14));
  Fish f13 = new Fish(env, new TurningDarterMovement(0.1),
             new IndependentBreeder(0.05),
           new Location(13, 15));
  Fish f14 = new Fish(env, new TurningDarterMovement(0.1),
             new IndependentBreeder(0.05),
           new Location(15, 16));
```

```
      // Construct an object that knows how to draw the environment with
      // a delay.
      SimpleMBSDisplay display = new SimpleMBSDisplay(env, DELAY);

      // Construct the simulation object. It needs to have the environment
      // and the object that can draw the environment.
      Simulation sim = new Simulation(env, display);

      // Run the simulation for the specified number of steps.
      for ( int i = 0; i < NUM_STEPS; i++ )
      {
        sim.step();
      }
    }
  }
```

Slow Three-way Movement

```
import java.util.Random;

public class SlowThreewayMovement extends ThreewayMovement
{
  private double probOfMoving;

  public SlowThreewayMovement(double probMove)
  {
    probOfMoving = probMove;
  }

  protected Location nextLocation(Fish f)
  {
    // There's only a small chance that a slow fish will actually
    // move in any given timestep, defined by probOfMoving.
    Random randNumGen = RandNumGenerator.getInstance();
    if ( randNumGen.nextDouble() < probOfMoving )
      return super.nextLocation(f);
    else return f.location();
  }
}
```

Three-way Movement

```java
import java.util.ArrayList;
import java.util.Random;

public class ThreewayMovement implements FishMovement
{
  public ThreewayMovement()
  {}

  /** Moves this fish in its environment.
   **/
  public void move(Fish f)
  {

    // Find a location to move to.
    Debug.print("Fish " + f.toString() + " attempting to move. ");
    Location nextLoc = nextLocation(f);

    // If the next location is different, move there.
    if ( ! nextLoc.equals(f.location()) )
    {
      // Move to new location.
      Location oldLoc = f.location();
      f.changeLocation(nextLoc);

      // Update direction in case fish had to turn to move.
      Direction newDir = f.environment().getDirection(oldLoc, nextLoc);
      f.changeDirection(newDir);
      Debug.println(" Moves to " + f.location() + f.direction());
    }
    else
      Debug.println(" Does not move.");
  }

  /** Finds this fish's next location.
   * A fish may move to any empty adjacent locations except the one
   * behind it (fish do not move backwards). If this fish cannot
   * move, <code>nextLocation</code> returns its current location.
   * @return the next location for this fish
   **/
  protected Location nextLocation(Fish f)
  {
    // Get list of neighboring empty locations.
    ArrayList emptyNbrs = f.emptyNeighbors();

    // Remove the location behind, since fish do not move backwards.
    Direction oppositeDir = f.direction().reverse();
    Location locationBehind =
  f.environment().getNeighbor(f.location(),
                                    oppositeDir);
      emptyNbrs.remove(locationBehind);
      Debug.print("Possible new locations are: " + emptyNbrs.
  toString());
```

```
      // If there are no valid empty neighboring locations, then we're
done.
       if ( emptyNbrs.size() == 0 )
         return f.location();

      // Return a randomly chosen neighboring empty location.
      Random randNumGen = RandNumGenerator.getInstance();
      int randNum = randNumGen.nextInt(emptyNbrs.size());
    return (Location) emptyNbrs.get(randNum);
   }
}
```

Turning Darter Movement

```
import java.util.Random;

public class TurningDarterMovement extends DarterMovement
{
    double probTurn;
    public TurningDarterMovement(double pt)
    {
      probTurn= pt;
    }

    protected double probTurn()
    {
      return probTurn;
    }

    public void move(Fish f)
    {
      Random rand = RandNumGenerator.getInstance();

      if(rand.nextDouble() < probTurn())
      {
          f.changeDirection(f.direction().toRight());
      }
      else
      {
        super.move(f);
      }
    }

}
```

Object-Oriented Design Concepts via Playing Cards

Owen Astrachan
Duke University
Durham, North Carolina

Most students have played card games: blackjack, war, hearts, solitaire, bridge. The list of games isn't infinite, but it's practically unbounded. In this design exposition, we'll discuss the design and implementation of a playing card class. We'll talk about issues in designing classes to represent both a deck of cards and piles of cards used in different games. The Web site that accompanies this design discussion includes references, code, and exercises for many assignments and in-class discussions. In this document we'll concentrate on the playing card classes and the deck class to keep things simple.

Students and teachers often wonder when it's appropriate to use a Java interface rather than a class. Some designers and educators think all object-oriented designs should start with interfaces. It is hard to motivate this stance with only a simple appeal to experts as a justification. In the design of a playing card class, our scenario begins with a teacher providing an initial specification and code to students and then asking them to write programs that play games with cards. Our goal is for one student's game or player to interact with another's. We'd also like to ensure that student-written code for a card player does not change the cards that are dealt. Using an interface provides a simple way for students to use cards in the code they write without having access to a card's internals, without being able to create a specific card, and without knowing how cards are implemented. This process begins with the code for a card interface, an interface we call ICard.[1]

```
public interface ICard extends Comparable
{
  public static final int SPADES = 0;
  public static final int HEARTS = 1;
  public static final int DIAMONDS = 2;
  public static final int CLUBS = 3;

  public int getSuit();
  public int getRank();
}
```

The interface specifies the behavior of a card without providing information about how cards are implemented. Once they know that getSuit() returns a value like ICard. HEARTS, and that getRank() returns a value in the range of 1 (ace) to 13 (king), students can write code from this specification. For example, here's code to check whether an array of cards is sorted. We don't know how it's been sorted (e.g., do all the aces come before the twos or do all the spades come before the hearts?), but we can determine that an array is sorted.

```
public boolean isSorted(ICard[] list){
  for(int k=1; k < list.length; k++){
        if (list[k-1].compareTo(list[k]) > 0){
             return false;
        }
  }
  return true;
}
```

Starting with this simple ICard interface, we can ask students many kinds of questions to test and review concepts ranging from Java syntax to problem-solving with respect to one, two, or many cards. Some simple examples are included here, and more are available on the Web site. In answering these questions students must understand the interface since there is no implementation. Students focus on behavior rather than on instance variables and other implementation details, such as how to create a string to represent the ace of spades.

ICard Study/Code Questions

1. Write the function `isRed` that returns true if its `ICard` parameter is red (hearts or diamonds) and returns false otherwise.

   ```
   public boolean isRed(ICard card){…}
   ```

2. A pair is two cards of the same rank (e.g., two kings or two eights). Write the function `isPair` that returns true if its two `ICard` parameters represent a pair and returns false otherwise.

   ```
   public boolean isPair(ICard a, ICard b){…}
   ```

3. A flush is a hand, say in poker, in which all the cards have the same suit (e.g., five hearts, or five clubs for a five-card hand). Write the function `isFlush` that returns true if the array of cards is a flush and returns false otherwise.

   ```
   public boolean isFlush(ICard[] hand){…}
   ```

4. In blackjack or 21, the value of a hand is the total of the cards, where jacks, queens, and kings (11, 12, and 13, respectively, as returned by `getRank()`) each count as 10, and an ace counts as 1 or 10, whichever is better. A total over 21 is a bust; it's not good to bust. Write function `handTotal`, which returns the total value of a hand.

   ```
   public int handTotal(ICard[] hand){…}
   ```

From Interface to Implementation

The `ICard` interface provides enough information to write code about cards, but there's no way to create an array of cards, for example, or even a single card to test the functions written above (like `isPair` and `handTotal`). Where do cards come from? In most real-world examples, cards come from a Deck. We'll design a class that models a Deck—which is basically a factory for creating and obtaining cards.

To keep things simple, and to encourage the study of some standard Java interfaces, the class `Deck` will implement the `java.util.Iterator` interface. For example, to store all the cards from a deck into an `ArrayList` variable, we can use the following code:

```
Deck d = new Deck();
ArrayList cards = new ArrayList();
while (d.hasNext()){
     ICard card = (ICard) d.next();
     System.out.println(card);
     cards.add(card);
}
System.out.println("# of cards dealt = "+cards.size());
```

The last few lines output by this code snippet might be as shown below. They will be different each time because the Deck class developed here shuffles the cards it deals via iteration.

```
...
ace of spades
jack of clubs
six of spades
ten of hearts
ten of spades
# of cards dealt = 52
```

If we change the lines after the loop as follows, the output changes as well.

```
Collections.sort(cards);
for(int k=0; k < cards.size(); k++){
     System.out.println(cards.get(k));
}
System.out.println("# of cards dealt = "+cards.size());
```

The output shows how cards returned from the Deck class implement the Comparable interface.

```
...
nine of clubs
ten of clubs
jack of clubs
queen of clubs
king of clubs
# of cards dealt = 52
```

The complete code for the class `Deck` is shown below. The methods `hasNext()`, `next()`, and `remove()` are required for classes that implement the `Iterator` interface. The code below shows how objects of type `Card` are constructed.

```
public class Deck implements Iterator{

        private ArrayList myCardList;
        private int myIndex;

        public Deck(){
              myCardList = new ArrayList();

              for(int suit = ICard.SPADES; suit <= ICard.CLUBS; suit++){
                    for (int rank = 1; rank <= 13; rank++){
                          myCardList.add(new Card(suit,rank));
                    }
              }
              shuffle();

        }

        private void shuffle(){
              Collections.shuffle(myCardList);
              myIndex = 0;
        }

        public boolean hasNext() {
              return myIndex < myCardList.size();
        }

        public Object next() {
              ICard card = (ICard) myCardList.get(myIndex);
              myIndex++;
              return card;
        }

        public void remove() {
              throw new UnsupportedOperationException();
        }
}
```

A `Deck` object stores 52 cards—these cards can be obtained from a `Deck` object via iteration, but a `Deck` object cannot be reshuffled and re-used. Instead, a new `Deck` object must be created to deal new cards. This keeps things simple and provides an easy-to-follow example of a class that implements the `Iterator` interface. The method `remove()` is optional— for the `Deck` class calling this method throws an exception.

Deck Study/Code Questions

1. Just before the shuffle method is called in the constructor, describe the order of the objects stored in myCardList.

2. Describe how each Deck method changes if the instance variable myCardList is changed to an array of Card objects, for example,

   ```
   private ICard[] myCardList;
   ```

   Which choice for myCardList is better? Why?

3. Write client code that defines a Deck object and creates an array of 13 ICard objects that represent the spades that are dealt from the Deck. Do this by examining each object dealt and only storing the spade cards.

4. Write the body of the hypothetical Hand class constructor specified below:

   ```
   private ArrayList myCards;
   /**
     * deal numCards cards from d, store in myCards
     * (assume there are at least numCards cards left in d)
     */
   public Hand(Deck d, int numCards){

   }
   ```

From Decks to Cards

Our original concern was to use the `ICard` interface rather than worry about how cards are implemented. Nevertheless, at some point, there needs to be an implementation. It's not hard to argue that `Card` objects should be created by the `Deck` class. This is the approach we've used here. The `Card` class is a private class declared within the `Deck` class. There's actually no good reason to declare it within the `Deck` (the Deck.java file). However, by declaring it private, we make it impossible for any code class[2]; it could just as easily be declared as a non-public class within methods other than the `Deck` class to construct `Card` objects. This helps meet our original goal. Client programs can obtain cards from a `Deck`, but cannot create cards. Since the `Deck` supplies `ICard` objects, it's not possible to change a card once it's obtained from the `Deck` since the `ICard` interfaced doesn't support modification of a card. As written, the private `Card` class defined within the `Deck` class doesn't support modification either since its private state instance variables are final, but this is extra protection that's likely not needed since no client code has access the private `Card` class. The `Card` class is available on the Web site; we're not including it here since its implementation isn't directly related to our discussion about design.

A careful reader might claim that our original goal hasn't been met. Client code can cheat, for example, by creating a `Deck` object and then dealing cards from this object until an ace of spaces (or any other card) is dealt. In the current design of the `Deck` class this is true. However, we could create a singleton `Deck` object, in the same way that a single instance of the class `Random` is used in the Marine Biology Case Study. Singleton objects are typically created by declaring constructors so that they're private. In this case the `Deck` constructor would change from public to private. Client code obtains a `Deck` by calling a public `getInstance()` method, which returns a private static `Deck` object stored in the `Deck` class. The `getInstance` method creates this private object the first time `getInstance` is called. Details can be found in many texts or by studying the code from the Marine Biology Case Study.

More Code and Details

 The images used in this article, and in the card games and supporting code available from the companion website, have been released under the GPL—the Gnu Public License. They are available from the creator of the images at www.waste.org/~oxymoron/cards/ and from many other sites (including the companion site for this material). The code supporting this design document is released under a Creative Commons License, as described at www.cs.duke.edu/csed/ap/cards, where the code and more material are available.

Notes

1. Beginning interface names with an uppercase I, followed by a capitalized name, is a common naming convention in object-oriented programming in many languages, not just Java.

2. Typically classes declared within another class often make reference to the enclosing object's state. In this case the nested class `Card` is declared as a private static class, so it can't reference private non-static state within a `Deck` object. The `Card` class could reference static `Deck` state, but there is none in this code.

Object-Oriented Programming Web Resources

Debbie Carter
Lancaster Country Day School
Lancaster, Pennsylvania

Teaching Techniques and Materials

How my Dog learned Polymorphism
JavaRanch
www.javaranch.com/campfire/StoryPoly.jsp

An illustration of inheritance using Animal and Dog classes.

Role Playing In an Object-Oriented World
Prof. David B. Levine and Prof. Steve Andrianoff, St. Bonaventure University, St. Bonaventure, New York

web.sbu.edu/cs/dlevine/RolePlay/roleplay.html

Descriptions and scripts for teaching object-oriented principles through role-play activities. Examples include a first-day activity and a Marine Biology Simulation role-play.

Karel J. Robot
Karel J. Robot consists of an online text and a set of Java classes that implement robots in a two-dimensional environment. It provides a very visual, highly intuitive context in which many OOP ideas can be effectively explored and demonstrated.

- Text and classes Prof. Joseph Bergin, Pace University, New York, New York
  csis.pace.edu/~bergin/KarelJava2ed/Karel++JavaEdition.html
- Demonstrations and code examples Prof. Don Slater, Carnegie Mellon University, Pittsburgh, Pennsylvania
  - Teacher Training Institute, CMU, July 2003
    www-2.cs.cmu.edu/~djslater/presentations.html#Training
  - SIGCSE 2003: "Helping Students 'See' Polymorphism," February 2003
    www-2.cs.cmu.edu/~djslater/presentations.html#SIGCSE2003

A Simple Calculator for Novice Learning
Prof. Joseph Bergin, Pace University, New York, New York
csis.pace.edu/~bergin/polycalc/index.html

Demonstrates what can be done in an object-oriented language without using if or while. It implements the key parts of a simple four-function algebraic calculator that has no knowledge of operator precedence.

Object Oriented Analysis and Design using CRC Cards
Nils Brummond
www.csc.calpoly.edu/~dbutler/tutorials/winter96/crc_b/

CRC cards were created for teaching the OO paradigm. This site gives the background of CRC cards and guides you through the use of them in the classroom.

Team OOP Projects as a Teaching Tool (*The Quizard of OOP*)
www.skylit.com/oop/index.html
Maria Litvin

A PowerPoint presentation guides students through a design of a quiz development project that ultimately assigns group tasks for a team project.

Online Textbooks: OOP-related excerpts

Note: Free, downloadable source code is available for all texts in this list.

Introduction to Programming Using Java
David J. Eck
Version 4.0, July 2002
math.hws.edu/javanotes/index.html

- Chapter 5: Programming in the Large II—Objects and Classes
  math.hws.edu/javanotes/c5/index.html
- Section 5.3: Programming with Objects (especially the card game ideas)
  math.hws.edu/javanotes/c5/s3.html
- Section 5.4: Inheritance, Polymorphism, and Abstract Classes
  math.hws.edu/javanotes/c5/s4.html

Java: An Eventful Approach

Kim B. Bruce, Andrea Pohoreckyj Danyluk, and Thomas P. Murtagh

applecore.cs.williams.edu/~cs134/eof/

From the authors: "The key features of the approach taken by our text are 'objects first', 'events first', and 'concurrency early'. We also emphasize graphics as a pedagogical aid. Students use graphical objects and do event-driven programming (primarily using mouse events) from beginning." The objectdraw library is freely available, along with instructor resources.

Thinking in Java, 3rd ed. (online version)

Bruce Eckel

www.mindview.net/Books/TIJ/

This text introduces objects at the very beginning. (Chapter 1 is "Introduction to Objects"; Chapter 2 is "Everything Is an Object.") It also has an annotated solution guide. Other particularly O-O chapters:

- Reusing Classes (Chapter 6): Composition and Inheritance.
- Analysis and Design (Chapter 16): A five-phase process for designing and developing a system of classes.

Java Au Naturel, 3rd ed.

Dr. William C. Jones, Jr., Central Connecticut State University, March 2003.

www.cs.ccsu.edu/~jones/book.htm

This text introduces object-oriented software design at the very beginning. Its copyrighted material is available free of charge for teaching, provided you fill out and submit a five-minute questionnaire. PDF files, source code, and syllabi are provided.

Introduction to Computer Science using Java

Bradley Kjell

chortle.ccsu.ctstateu.edu/cs151/cs151java.html

A Java programming tutorial with quizzes, flash cards, reviews, and programming exercises. Part 4, Object Oriented Programming, uses car and checking account examples. It also includes good diagrams.

Java: an Object First Approach
Fintan Culwin
www.scism.sbu.ac.uk/jfl/jflcontents.html

The OOP paradigm is emphasized throughout. The end-of-chapter exercises could be modified to incorporate students' ideas.

Computer Science, Java Style
Craig Graci
www.cs.oswego.edu/~blue/java/hyperbook/org/Contents.html

Part 1, Chapter 3 is "A Conception of the Java Object Model" and develops circle, rectangle, square, triangle, and polygon classes and some applications.
www.cs.oswego.edu/~blue/java/hyperbook/part1/chapter3/Contents.html

Contributors

Information current as of original publish date of September 2004.

About the Editor

Fran Trees taught AP Computer Science from 1983 to 2001 in Westfield, New Jersey. She presently teaches CS1/CS2 at Drew University in Madison, New Jersey. Fran is a College Board consultant for AP Computer Science, an exam leader, and AP Central's content advisor for AP Computer Science.

Welcome Letter

Scot Drysdale teaches computer science at Dartmouth College and is currently serving as Chair of the Department of Computer Science. He has been on the Computer Science Development Committee for four years and recently became Chair of that committee. He has also served as a Reader for the AP Computer Science Exam.

Syllabi Contributors

Steven K. Andrianoff is a faculty member in the Computer Science Department at St. Bonaventure University, where he has taught since 1979. He holds a Ph.D. from Syracuse University. He has seven years experience as both a Reader of the AP Computer Science Exam and a College Board consultant in computer science. In addition, he has published papers in a variety of areas of computer science, most notably in the field of computer science education.

Rodney Hoffman teaches computer science at Occidental College. He also works as a software engineer at NASA's Jet Propulsion Laboratory. He is a longtime AP Computer Science Reader. He has been a local and national officer of Computer Professionals for Social Responsibility.

Kathy Larson teaches AP Computer Science and AP Statistics at Kingston High School in Kingston, New York. She has served as a Reader, Question Leader, and member of the AP Computer Science Development Committee. She also has a strong interest in developing Vertical Teams in her district.

Maria Litvin teaches mathematics and computer science at Phillips Academy in Andover, Massachusetts. She is a College Board consultant in computer science for New England and an AP Exam Reader. Maria is a recipient of the 1999 Siemens Award for AP teachers and of the 2003 RadioShack National Teacher Award. She is also a co-author of several computer science textbooks.

Don Slater is a lecturer in the School of Computer Science at Carnegie Mellon University. He has been a Reader and Question Leader for the AP Computer Science Exam since 1991.

Barbara Wells is a teacher at South Fork High School in Stuart, Florida. She has been teaching AP Computer Science since 1985. She is a Reader for the AP Exam and has received many awards during her 30-year involvement in education.

Review Contributors

Brian Ellis is a mathematics and computer science teacher at Manheim Township High School in Lancaster, Pennsylvania, where he has taught AP Computer Science for six years. He has finished his M.S. in computer science, having studied different practices in the teaching of OOP and Java.

Kathy Larson teaches AP Computer Science and AP Statistics at Kingston High School in Kingston, New York. She has served as a Reader, Question Leader, and member of the AP Computer Science Development Committee. She also has a strong interest in developing Vertical Teams in her district.

Chris Nevison teaches computer science at Colgate University in Hamilton, New York. He is currently the Chief Reader for AP Computer Science and was on the AP Computer Science Development Committee for eight years.

Theme Material Contributors

Owen Astrachan is Professor of the Practice of Computer Science at Duke University. He taught high school for seven years and served on the AP Computer Science Development Committee, including five years as Chief Reader from 1990 to 1995.

Debbie Carter is Upper School Computer Coordinator at Lancaster Country Day School in Lancaster, Pennsylvania, where she teaches computer science and assists faculty with technology integration. She is a question leader for the AP Computer Science Exam.

Robb Cutler is the Department Chair of Computer Science at The Harker School, a K-12 college preparatory school in San Jose, California, where he teaches AP Computer Science and leads post-AP seminars on a variety of computer science topics. Robb is currently a Reader for the AP Computer Science Exam.

Judy Hromcik is a teacher at Arlington High School in Arlington, Texas. Judy is a current member of the AP Computer Science Development Committee and has served as both a Reader and a question leader. Judy has a strong interest in developing curriculum for computer science and technology applications.

Joe Kmoch is a teacher at Washington High School in Milwaukee, Wisconsin. He served on the AP Computer Science Development Committee and is the Director of one of the 12 national pilot sites for the Academy of Information Technology. In addition, he is Chair of the International Society for Technology in Education's Special Interest Group for Computer Science.

Maria Litvin teaches mathematics and computer science at Phillips Academy in Andover, Massachusetts. She is a College Board consultant in computer science for New England and an AP Exam Reader. Maria is a recipient of the 1999 Siemens Award for AP teachers and of the 2003 RadioShack National Teacher Award. She is also a co-author of several computer science textbooks.

Chris Nevison teaches computer science at Colgate University in Hamilton, New York. He is currently the Chief Reader for AP Computer Science and was on the AP Computer Science Development Committee for eight years.

Dave Wittry is the Computer Science Department Chair at Troy High School, the largest magnet program for computer science in California. He has taught AP CS A and AB for eight years and has been an AP Reader for the College Board for five years. Dave is a contributor to the *Be Prepared for the AP Computer Science Exam* review book. He has been instrumental in developing and teaching the innovative computer science curriculum, which has contributed to Troy's immense success as a California Distinguished School, a National Blue Ribbon School of Excellence, and a New American High School—one of only 17 showcase schools in the nation.