Calculator Problem and the Design Recipe

Viera K. Proulx
College of Computer and Information Science
Northeastern University
Boston, MA
vkp@ccs.neu.edu

Tanya Cashorali
College of Computer and Information Science
Northeastern University
Boston, MA
cash21@ccs.neu.edu

ABSTRACT

This paper presents a superior alternative approach to designing a solution to the calculator problem due to Alphonce [1]. While Alphonce presents a fictitious one-act play between professors, this paper consists is a narrative explanation of the Java code written by this paper's coauthor, – a student who just completed the freshman year.

The purpose of this paper is to illustrate the use of the DESIGN RECIPE pedagogy and the idea of focusing on the design of the structure of data, the classes, and class hierarchies, rather than the design of program actions (the algorithmics) as the core idea for program design. By contrasting the pedagogical approach and the final outcome with the "traditional" objects-first approach, we illustrate the advantages of our curriculum that teaches the students how to think about program design in a truly object-oriented style.

1. INTRODUCTION

In May 2004 issue of the Curricular Patterns, Carl Alphoncee [1] describes the conversations of several colleagues that led to two solutions for a problem of emulating a calculator. The goal was to illustrate for students the power of polymorphism. We believe, that their approach is fundamentally flawed and leads to both a wrong model of computation and a wrong pedagogy for teaching the design of class based programs using object-oriented languages. We have the following objections to the overall approach:

- While presented as a socratic dialog, the discussion is not between the teacher and the apprentice student, but between two colleagues. As the result, it does not present a pedagogy for teaching students about the problem.
- The approach that focuses on the algorithmic solution without analyzing the structure of data produces a complex solution, that is un-inspiring and confusing example of the use of design patterns. In real life, people learn about patterns by observing a number of similar situations and abstracting over them, thus not only understanding the pattern, but also the underlying abstraction process. Our focus should be on teaching these skills.
- The approach fails to recognize that the behavior of a calculator is realy a variant of an interpreter. The design of interpreter follows closely the structure of the language of the expressions being evaluated - resulting in a cleaner, more scalable, and more systematically designed solution. Additionallly, it provides an opportunity to illustrate to students the nature of computation that underlies the program evaluation.

We believe students should discover design patterns as they design their own programs where these patterns emerge in the appropriate context. In contrast to the design patterns based approach, our curriculum, under development for the past three years, focuses on the structure of data as the key to understanding the design of class hierarchies, the model of computation, and the role of polymorphism in object-oriented programs. Combined with the pedagogy that supports the student with clearly defined steps of the design process in the form of the DESIGN RECIPE [1] we have increased the retention in our courses and produced students who are much better prepared for the subsequent course on Object Oriented Design. This claim is based both on the numerical results, and on the subjective comments of the colleagues teaching the subsequent classes. Additionally, the responses to our summer faculty development workshops over the past two years have been overwhelmingly positive, with several high school teachers using the curriculum with great success in their classrooms.

The calculator problem provides an excellent context for presenting a concrete example of the effect of our pedagogy of following the DESIGN RECIPE and focusing on the design of class hierarchies in a systematic way [6].

2. THE DESIGN RECIPE

The DESIGN RECIPE is a blueprint for the design of programs or for defining data - tailored to a particular structure of the underlying information. At each step of the design process it describes the questions to ask, and defines a tangible outcome that can be seen and checked for correctness

[1] DESIGN RECIPE, is not a design pattern!!!

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2005 ACM 1-xxxxxxxx/05/xxxx ...$5.00.

or completeness. The DESIGN RECIPE teaches students to think about problems in a systematic way. It also provides the instructor with a pedagogical intervention tool - a way to ask the student questions that show where the student is stuck, and assists in guiding the student in overcoming the problem. This pedagogical approach, known as self-regulated learning [5, 13, 15], has been shown successful in other learning situations, including in support of weaker learners.

The basic DESIGN RECIPE consists of the following steps:

Problem analysis and data definition: Analyze the problem, the available information, represent the information as data.

The purpose statement, contract and header: Write a brief statement that describes the purpose of the program, define the data it will consume and produce, write the program header (a function header or a method header, and in case of untyped languages, a contract as well).

Examples: Create working examples illustrating the use of the program, together with the expected outcomes.

Template: Write down a template of all data items available to solve the problem. For example, when the program consumes compound data, identify all components that are available.

Body: Desing the body of the program.

Tests: Use the earlier examples as test cases to test your program. You may add more tests, but they should not be necessary.

Each variant of the DESIGN RECIPE decribes in greater detail the questions student should ask when analyzing the problem, provides guidelines for developing the templates based on the structure of data, explains how to design a suitable set of examples, etc., reflecting the complexity of the problem structure and the language constraints. The series of design recipes then forms a framework for dealing with increasing levels of complexity of program design.

Notice that the DESIGN RECIPE enforces the design of the test suite prior to the development of the program body. Additionally, this design process assures that the student understands how the program she designs will be used by the client code - a step often neglected in programming courses.

Our teaching consists almost entirely of a true socratic dialog - between the teacher and the apprentice student, the questions supplied by the DESIGN RECIPE. Students quickly learn to ask the same questions and work on the design independently.

However, the DESIGN RECIPE alone would not work, if the problems our students worked on required them to address a number of confusing issues at the same time. To constrain the complexity in a systematic way, the structure of the programs students first work on follows the structure of the data that the program consumes. The series of DESIGN RECIPES then reflect the increasing complexity of the strucutre of data and programs that consume this data. This type of programs are well studied in the programming langugages community and can be proven correct. The advantage from the pedagogical point of view is that students do not need to invent algorithms or understand additional domain knowledge. In due time our examples bring in many interesting applications of computing, including animated graphics and event handlers, but within the confines of a well structured context. Students realize quickly that this approach scales up to very complex problems - as demonstrated by the programming environment they use: ProfessorJ [11] series of languages within DrScheme [10] programming environment — a programming langugages research project in its own right.

To illustrate the design process students work through, this paper is written by two authors, the teacher and the student. During the summer the student has been working for the teacher on an administrative project that does not involve programming, and she has not touched Java for three months. She agreed to come to school for a day to work on the problem. She did not know anything about the problem, other than that it would involve modeling a calculator. The teacher gave her Alphonce's paper, with the instructions to only read it on the train ride to Boston, so she would not try too hard to understand the approach presented there.

Section 3 presents the student's narrative of her experience. The teacher's comments interrupt the narrative to clarify some of the DESIGN RECIPE terminology, to fill in a few missing pieces, and to comment on the design process from the teacher' spoint of view. Section 4 contains the code, written entirely by Tanya. Section 5 contrasts our pedagogy and our way of teaching object-oriented program design with the traditional approaches. An appendix explains some of the conventions used in our teaching and shows how the code translates to a more traditional approach.

3. THE DESIGN PROCESS

Dramatis Personae

S: the student
T: the teacher

S: The first time I read over the paper on the calculator program, I was on a northbound train headed for Northeastern University in Boston. I felt a little nervous that I was going to try to design this program using the DESIGN RECIPE, which I learned in the freshman year. The program seemed too complex and the solutions offered were not easy to follow. I was also pretty rusty on my Java and programming skills in general. I read the solutions over several times and then put them away and continued reading The Da Vinci Code.

S: I arrived in Boston and met up with the teacher in her office. She immediately handed me a calculator and a pad of paper. She told me to press some buttons and solve a few equations. Then she asked, "What is happening?" It seemed too easy. I replied, "I press a number, then the plus sign, then another number, then if I press the plus sign again, it will evaluate the ongoing sum." She then told me to write down what data was available before any buttons are pushed on the calculator. The calculator starts off with "0" as the display. The initial equation inside the calculator is "0 + 0." The teacher then told me to create a chart that shows the display on the calculator, what I input, and what the calculator is storing inside. It looked like this:

Input	Information Inside Calculator	Display
start	0 + 0	0
2	0 + 2	2
+	2 + 0	2
4	2 + 4	4
+	6 + 0	6
1	6 + 1	1
+	7 + 0	7

S: This is the first step in the design recipe, which is to **analyze the problem, determine what data is given, and what type of data must be produced.** This chart helped me realize that a calculator has this type of data:

- Input - integer or operator
- Information inside calculator (int1, operator, int2)
- Display - int

T: As we can see, it is the student who discovers the structure of the data and reasons about what is going on. Teacher's intervention is mostly in terms of questions, only occassionally suggesting the preferred path.

S: I then went into an office and began working out this problem given the following information.

- The person knows how to operate a calculator and will only press numbers followed by operators. (Will not press 2, + 3, +, +, -, +)
- There is no need for equals operation, just add, multiply, subtract, divide.

S: Based on this information and the chart of data, I named the necessary functions and classes I would need to evaluate these pieces of information. The first class I created was the `Calculator` class, which contained all of the variables in my initial chart. The only operator class I started off with was the addition operator. I knew it was the only one I had to start out with because once I figured that out, the rest of the operators could be abstracted based on the way the addition class worked. Then I had to name the functions within each class and write their **purpose statements**. I made the methods `digit` and `operation`. [The method] `digit` would consume an integer and produce a new `Calculator`; it also allowed the calculator to use more than just one-digit numbers. (i.e. 2, 2, 3, = 223). This function just multiplied the first number by 10 and added the next number inputted. The last function in `Calculator` was `operation`, which consumed an operator (`+`, `*`, `-`, `/`) and produced a new `Calculator`. The `operation` method actually just invokes the abstract `evaluate` method in the appropriate `Operator` [sub]class. The `Plus` class contained a function called `evaluate` that would simply return the sum of `int1` and `int2`.

T: In our terminology a method consumes data (the instance which invoked the method, and the method's arguments) and produces a value. Methods that do not produce values (with the return type `void`) are not used at the beginning, as we are "favoring immutability" [4] and "using value objects whenever possible" [3]. The OO community is realizing that programs which avoid mutation are easier to work with, to debug, and to verify. While the advantages of immutability are well understood and supported by the research in programming languages, the educational community is mostly ignoring the lessons it could learn.

T: At this point the student is working on several methods and classes concurrently, but understands throughout the structure of data as represented in the class diagram (shown later). The student also feels comfortable with using methods for which only the purpose and the header are known. This is understandable, as the student is used to making examples of use for the methods specified by their purpose and the header. This also prepares students in general for working with libraries and other code, where the implementation of the methods is not known to the user.

S: I then proceeded to step 2 of the Design Recipe: **examples**. I then prepared examples of how the calculator program would work based on my chart.

Examples:

```
Calculator c = new Calculator(0, 0, new Plus(), 0);
    //creates an initial blank calculator.
c.digit(8)
    //inputs 8 into the Calculator c.
c.digit(8).display == 8
    //checks to see that the display is 8
c.digit(8).num2 == 8
    //checks to see that num2 is 8 (0 + 8)
Calculator c2 = c.digit(8);
    //creates a new calculator c2 and inputs 8
c2.operation(m)
    //should produce new Calculator(8, 8, m, 0)
c2.operation(p).digit(3).operation(p)
    //should produce new Calculator(11, 11,p, 0)
Calculator c3 = c2.digit(2);
//creates a new calculator c3 from c2 and inputs 2
c3.display == 82
    //checks that display is 82 on calculator
```

T: For a homework assignment the examples would have to be constructed more carefully — covering all possibilities and consisting entirely of the pairs of lines: [method invocation] ["should be" result]. In our curriculum this leads naturally to the discussion about the various ways in which two compound structures may be considered equal — a very important issue to understand.

S: Now that I knew how the program should be working, I constructed **templates** for each function. This is the third step in the Design Recipe. That means I wrote every piece of available data for each function. For example the digit function within the `Calculator` class looked like this:

Template:

```
Calculator digit (int n){
    ... this.display ...
    ... this.num1 ...
    ... this.op ...
    ... this.num2 ...
    ... n ...
}
```

S: The template made me aware of the pieces of data I had available within the digit function. The purpose for `digit` was that it

```
//consumes an integer and produces a new Calculator
```

S: I knew this because I remembered my chart every time a key was pressed, whether it was a number or operator, a new `Calculator` was always produced.

S: Following the DESIGN RECIPE, I continued to the fourth step. Using my templates as a guide, I began to narrow down the data for all of the functions and **figured out how each of the functions formulated their results based on purpose statements, the examples, and the template**. I already knew what data was available due to my templates and what each function should consume and produce, so it just came down to arithmetic thinking. I realized how the `Plus` class worked and the teacher told me to make the rest of the operator classes work just like that one. I created an abstract `Operator` class that every other operator class extended. Then it was simple, the user supplied the operator like this:

```
  p = plus, m = multiply, d = divide, s = subtract

  p.evaluate(5, 6)      // would add 5 and 6.
  m.evaluate(2, 5)      // would multiply 2 and 5.
  d.evaluate(8, 2)      // would divide 8 by 2.
  s.evaluate(6, 3)      // would subtract 3 from 6,
                        // and so on.
```

`p`, `m`, `d`, and `s` extended the abstract class, which looked like this:

```
abstract class Operator {
  // evaluate the appropriate operation
  abstract int evaluate(int n1, int n2);
}
```

S: Each `Operator` [sub]class has a method `evaluate`, which consumed two integers and the user already inputted whether to add, multiply, subtract, or divide. Therefore the only differences in the classes are the return statements, which just evaluate the two numbers using the given operator in the matching class.

T: The student does not describe the design of the `operation` method that appears in her code. She had no problem designing the method, or using it once she worked out the examples.

T: The student also designed the class diagram along the way. For more complex problems we usually separate the design of the class hierarchy from the design of the methods. Some of that thinking is reflected in the initial stages of this design process.

S: Now the calculator operations were abstracted and everything appeared to be written efficiently. It seemed too simple to be true. Sure enough, I ran my final **tests**, the final step of the DESIGN RECIPE, Professor Proulx looked them over, and the program was complete and it worked! I went through all of the proper steps of the DESIGN RECIPE and was out of the office in three hours, something I had not imagined I was capable of doing.

T: This last comment makes it clear that this was a typical student — not the 'know-it-all' wizard — who just learned to work out problems in a systematic way and could use her knowledge in a new situation.

4. THE CODE

The Operator Classes:

```
// represent an operator key on a calculator
abstract class Operator  {
  // evaluate the appropriate operation
  abstract int evaluate(int n1, int n2);  }

class Plus extends Operator  {
  Plus(){}

  int evaluate(int n1, int n2) {
    return n1 + n2; }                    }

class Subtract extends Operator {
  Subtract(){}

  int evaluate(int n1, int n2) {
    return n1 - n2; }                    }

class Multiply extends Operator {
  Multiply(){}

   int evaluate(int n1, int n2) {
     return n1 * n2; }                    }

class Divide extends Operator {
  ...
}
```

The Calculator Class:

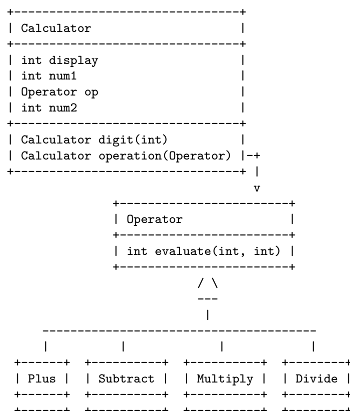
```
// represent a simple calculator
class Calculator {
  int display; //current display on calculator
  int num1;    //current number that's been hit
  Operator op; //operator
  int num2;    //next number that's been hit

  Calculator(int display, int num1,
             Operator op, int num2) {
    this.display = display;
    this.num1 = num1;
    this.op = op;
    this.num2 = num2; }

  // consumes an int
  // and produces a new Calculator
  Calculator digit(int n) {
    return new Calculator(
                 (this.num2 * 10) + n,
                 this.num1,
                 this.op,
                 (this.num2 * 10) + n );  }
```

```
  // consumes an operator
  // and produces a new Calculator
  Calculator operation(Operator op) {
    return new Calculator(
        this.op.evaluate(this.num1, this.num2),
        this.op.evaluate(this.num1, this.num2),
        op,
        0 );  }
}
```

Test Cases:

```
Operator p = new Plus();
Operator m = new Multiply();
Operator d = new Divide();
Operator s = new Subtract();

Calculator c = new Calculator(0, 0, new Plus() , 0);
c.digit(8).display == 8
c.digit(8).num2 == 8

Calculator c2 = c.digit(8);
c2.digit(2)         // should produce
new Calculator(82, 0, p, 82)

s.evaluate(3, 4) == -1
p.evaluate(56, 23) == 79
p.evaluate(1, (d.evaluate(10, 5))) == 3
m.evaluate(7, 8) == 56
p.evaluate(5, 71) == 76

c2.operation(m)     // should produce
new Calculator(8, 8, m, 0)

c2.operation(p).digit(3).operation(p)
// should produce
new Calculator(11, 11, p, 0)

c2.operation(m).digit(3).digit(4).operation(p)
// should produce
new Calculator(272, 272, p, 0)
```

The Class Diagram



5. REFLECTION

S: In conclusion, the Design Recipe is the most efficient and simplest method to use when writing programs of any level of complexity. I compared the approach used in my high school computer science AP class with the Design Recipe approach. In high school, we walked through the program line by line, adding variables as we went, making up algorithms in the middle of it all, and then testing. The Design Recipe on the other hand, provides a more structured guideline to solving any problem presented.

S: After one semester of learning the Design Recipe and coding in the DrScheme environment, we all easily adapted to programming in Java. We applied this method to our Java programs and found it rather simple to complete our homework assignments. The course jumped right into recursion and assorted data types (stacks, arrays, vectors), but we had already learned these fundamental building blocks of computer science, which made for a smooth transition.

This approach allows anyone to start out programming efficiently even if they have no prior experience. It will also improve a veteran programmers skills and make his life a lot easier, as it eliminates much of the obstacles that computer scientists run into when programming. In my experience the Design Recipe has been helping me in my other computer science courses, and it gives me confidence knowing that I can work through any programming problem I will encounter.

T: Our pedagogy and the code designed by the student is superior to Alphonce's [1]. The pedagogy enables a student to work independently on solving problems, regardless of the programming langugage used. In the context of object-oriented program design, it leads to a design where the structure and meaning of all classes and methods is clear. There are no unnecessary variables recording the current state of the calculator - the dispatch between the different actions is governed entirely by the key pressed by the user. Our implementation also models nicely the behavior of an interpreter and scales easily to deal with the equal sign, and even the precedence of operator.

It is shocking, that a first year student can design in a short time and with no preparation a program that is in many ways superior to the design that has been presented by seasoned computer science instructors as a model to emulate. The "design patterns first" approach presented in [1] is difficult even for the instructor. According to the paper, it took several days for the instructors to work out the solution. A search through textbooks [2, 14] failed to find a better solution to this problem.

6. ACKNOWLEDGMENTS

This "curriculum pattern" is a joint work of the whole team (see http://www.ccs.neu.edu/home/vkp/htdch), but the authors are responsible for the contents of this note. The authors would like to thank Matthias Felleisen who taught us both how to ask questions. Thanks to Marc Smith for valuable suggestions on how to improve this paper.

7. REFERENCES

[1] C. Alphonce, Pedagogy and Practice of Design Patterns and Objects First: A one-act play. *ACM SIGPLAN Notices*, 39(5):11–14, May 2004.

[2] D. J. Barnes and M. Koelling. *Objects First With Java: A Practical Introduction Using BlueJ* Prentice Hall, 2003.
[3] K. Beck. *Test Driven Development By Example* Addison Wesley, 2001.
[4] J. Bloch. *Effective Java*. Addison Wesley, 2001.
[5] Peggy A. Ertmer, and Tomothy J. Newby. The expert learner: Strategic, self-regulated, and reflective. *Instructional Science*, 24(1):1–24, 1996.
[6] M. Felleisen, R. B. Findler, M. Flatt, K. E. Gray, S. Krishnamurthi, and V. K. Proulx. *How to Design Class Hierarchies*. In preparation, (See http://www.ccs.neu.edu/home/vkp/htdch/).
[7] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs:An Introduction to Programming and Computing*. MIT Press, 2001.
[8] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. Structure and interpretation of the computer science curriculum. *FDPE*, 2002.
[9] M. Felleisen and D. Friedman. *A Little Java A Few Patterns*. MIT Press, 1998.
[10] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002. A preliminary version of this paper appeared in PLILP 1997, LNCS volume 1292, pp. 369–388.
[11] K. E. Gray and M. Flatt. ProfessorJ: A gradual intro to Java through language levels. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2003.
[12] Junit framework http://www.junit.org.
[13] Bonnie D. Singer, and Anthony S. Bashir", What are executive functions and self-regulation and what do they have to do with language learning disorders. *Language, Speech, and Hearing Services in Schools*, 30:265–273, 1999.
[14] C. T. Wu. *An Introduction to Object-Oreinted Programming with Java, 3rd. ed.* McGraw Hill 2004.
[15] B. J. Zimmerman, and D. H. Schnuck (eds.) Self-regulated learning and academic achievement: Theory, research and practice. Springer Verlag, 1989.

8. APPENDIX

Testing the code

To properly test all class definitions and all methods, we require that each new class definition is instantiated with appropriate examples, and that the examples of the method invocation used in the design are converted into test cases. Whenever possible, the evaluation of the test results should be automatic. We are working on making this possible in multiple environments and for different levels of equality comparison.

In a plain Java environment our sample instances would become member data of a `TestClient` class, and and the method invocations would be contained in a method that performs all tests and reports the results. So, for example,

```
m.evaluate(7,8) == 56
```

could be replaced by

```
System.out.println(m.evaluate(7,8) == 56);
```

within some test method.

Access modifiers and the use of `this`

The code does not include any access modifiers, as we delay the discussion of access modifiers (`public`, `private`, etc.) until the students are comfortable writing code where such differences are meaningful.

We also qualify each access of the member data within the method definitions with "`this.`" to highlight the use of the instance on which the method was invoked as its implicit argument. For example, the body of the method that compares the price of an instance of a book with the price of other book becomes:

```
return this.price < that.price;
```

illustrating clearly the use of the invoking instance as an implicit argument for the method.

Converting to the imperative style

The imperative style that requires mutation of one `Calculator` object is easily derived from our solution. The method digit replaces the invocation of a new constructor with the statements that comprise the body of the constructor. The new variant of the method will then be:

```
// consumes an int
// effect: the Calculator reflects the new state
void digit(int n) {
  this.display = this.num2 * 10) + n;
  this.num1    = this.num1;
  this.op      = this.op;
  this,num2    = (this.num2 * 10) + n;
}
```

Care has to be taken that values are not modified when they still may be needed to set the subsequent values, and redundant assignments, such as `this.op = this.op;` can be eliminated. The `operation` method would become:

```
// consumes an operator
// effect: the Calculator reflects the new state
void operation(Operator op) {
  this.display = this.op.evaluate(this.num1,
                                  this.num2);
  this.num1    = this.op.evaluate(this.num1,
                                  this.num2);
  this.op      = op;
  this.num2    = 0;
}
```

Error checking

One of the solutions in the original paper signals an error when the user hits two successive operator keys. Following the Design Recipe, we would realize that the calculator has a variant that responds differently to the operator key, warranting the definition of a `NoOpCalculator` subclass of the original `Calculator`, that redefines the `operation` method:

```
class NoOpCalculator extends Calculator {

  // consumes an operator,
  // produces calculator with no change
  // signal error if operator key hit twice
  Calculator operation(Operator op) {
    System.out.println("Repeated operator");

  // do not allow third operator either
  return this;
  }
}
```

The original `operation` method in the class `Calculator` then returns a new `NoOpCalculator`, indicating that the next input should be a digit.

It should be noted, that at this point, it is no longer trivial to convert the code to the imperative style, precisely because the problem illustrates the state design pattern. The discussion on how to proceed can then be framed in a clearly understood context. The change of state may even be managed by the GUI - by disabling the operator keys, once the operator has been hit, until a digit key has been pressed at least once.

Views

We have several implementations of the views for this model — one that responds to the input from the computer keyboard, one where the user interaction is via buttons in a GUI, as well as text-based evaluations in ProfessorJ's interactions window.

Late September

In late September, the student came to the teacher's office, all excited:

> "Teacher, we had to design this function in C to convert numbers from any base to any base, and Lindsay and I followed the Design Recipe and we got it and we really understand how it works. And the other students were asking us, how we got it so fast, and we told them, 'follow the design recipe!'. It works for any language!"