

## PROBLEM STEREOTYPES AND SOLUTION FRAMEWORKS: A DESIGN-FIRST APPROACH FOR THE INTRODUCTORY COMPUTER SCIENCE SEQUENCE

T.M. Rao, Sandeep Mitra  
Department of Computer Science  
SUNY Brockport,  
Brockport NY 14420  
585-395-5176  
{trao, [smitra](mailto:smitra@brockport.edu)}@brockport.edu

Roxanne Canosa, Sidney Marshall  
Department of Computer Science  
Rochester Institute of Technology  
Rochester, NY 14623  
{rlc, swm}@cs.rit.edu

Thomas Bullinger  
ArchSynergy, Ltd.  
2500 Turk Hill Road  
Victor, NY 14564  
[tom@archsynergy.com](mailto:tom@archsynergy.com)

### ABSTRACT

In this paper, we propose the use of a new design-first approach, *Problem Stereotypes* and *Solution Frameworks*, for use in the introductory computer science courses. A *problem stereotype* is simply a representative of a group of problems that can be solved using similar techniques. A *solution framework* is a typical solution to the problem stated in terms of the problem stereotype. Students are introduced to a selection of related problems, and given the problem stereotype and a solution framework for them. Homework problems come from the same stereotype, with students expected to follow the provided working examples to generate their own artifacts. We feel that this reduces the stress level for beginner students, and prevents them falling prey to the "CS is HARD" myth. We present the results of our experience with this approach in three introductory sequence classes at SUNY Brockport and RIT.

### 1. INTRODUCTION

The decline in enrolment in Computer Science programs that began following the dot-com bust has resulted in part from student perception that Computer Science is a "hard" discipline. Students get this idea right in the introductory CS courses, largely due to the techniques used to teach these courses. Many instructors wish to teach *programming*, and focus on making students develop *correctly working* programs. Consequently, these courses often emphasize the syntax and semantics of programming in high-level languages. Students are expected to code a wide range of algorithms in these languages and make them "work" – this is what determines their grade.

The "hard" perception results from the fact that many students find it very challenging to come up with correctly working code within the deadline, just from the English problem description. In today's *Objects First* [6] world, in addition to devising the algorithmic behavior of the solution, students are also faced with the need to come up with the right *structure* for it. They are taught the techniques of writing down the structure and behavior in forms suitable for the compiler, but they are taught little about the means of devising these from the natural language description of the problem. This is akin to asking engineers to design a car without blueprints of any sort. Therefore, if students could be provided with the right amount of *engineering process* in the introductory courses, their problem solving capabilities would be enhanced to the extent they could perform better in coming up with working programs. Thinking on these lines, Proulx *et al* [1] have emphasized the need to design an effective class/object structure before concerning the programmers with algorithmic detail. Caspersen *et al* [2] also ask students to first create class models in UML [5]. UML class diagrams are created from the problem statement in English, mapped

to Java code skeletons, and their methods are finally “fleshed out”, again on the basis of the problem description.

Our approach is similar, advocating the use of a process that creates a *model* before writing code. The model, however, is that of the *problem itself* rather than its eventual implementation. The instructor evaluates the models, following which students are taught *systematic techniques* of translating these models into program code. Instructors evaluating the final submissions ensure that these techniques have been followed. Students are given practice in these techniques by presenting them with numerous examples of a particular *category* of problems. Project problems are in the same category, and very similar to these examples. Students consequently learn *patterns* (see [3]) for writing code for these categories, thus gaining proficiency. This paper provides information about our experience in using this approach in a CS1 course at SUNY Brockport for one year, and for a more advanced level course (CS4) at RIT one quarter.

## 2. PROBLEM STEREOTYPES AND SOLUTION FRAMEWORKS

A *problem stereotype* is simply a representative of a group of problems that can be solved using the same technique. It consists of a set of artifacts that represent a clear understanding of the problem. A *solution framework* is a typical solution to the problem stated in terms of the problem stereotype. It consists of artifacts that describe the solution – i.e. the implementation and testing of the problem stereotype artifacts in a programming language. A parallel may be found in mathematics where we have a set of “word problems”, each of which must be first modeled as a pair of simultaneous equations, and then solved using well-known techniques. A math teacher will provide many examples of such problems, demonstrating how to model them as two equations involving two unknowns. The teacher then shows how to solve the equations. We propose to adapt this approach to teaching introductory programming.

Simple computing problems taught in the introductory sequence fall into groups of related problems. Problems in a group have similar solution techniques. For example, problems such as “Convert a temperature from Celsius to Fahrenheit,” “Compute interest on a loan,” “Find the area of a geometrical shape” all belong to the problem stereotype that may be categorized as “Expression Evaluation” problems. Solutions to such problems all involve the following algorithmic steps: “Obtain input; Compute output using a sequence of expressions; Display output.” In addition, these steps are typically described in a “computeValue()” method of a particular class. For the individual problems mentioned, the name of this class will be different, but it will certainly exist and encapsulate the method to compute the desired value using an expression. In strict object-oriented terms, therefore, we can consider a problem stereotype as expressing the *abstraction* associated with a set of problems. For pedagogical purposes, however, especially in the CS1 class, we do not describe this abstraction in terms of Java abstract classes, etc. – instead, we take the approach described below.

Our idea is to present to the student a suitable number of similar problems and show how all of them can be solved by employing the same general technique. We present a set of “artifacts” that describe the problem stereotype in programming language-independent terms. The solution framework consists of artifacts that represent the stereotype artifacts in a programming language. Problems given as assignments should belong to the same stereotype and should be able to use the same framework. Students are asked to first develop similar artifacts, finally leading to the full implementation.

Artifacts that describe the problem stereotype include the following:

- Problem Statement: this is the traditional description of the problem in natural language.
- Use cases: The principal component of a use case, which must be precisely written, is the *workflow description* shown in Figure 1. We insist that the sequence of events comprising this description be written in *structured* English, with the essential requirement that each sentence be in the following form: WHO-does-WHAT-to-WHOM.
- Class-Responsibility-Collaboration (CRC) cards [4] that specifically outline the classes and the initial set of methods for each class. We show students how these cards can be *derived* from well-written use case workflows, and how they are the starting point of the code for the solution framework. In effect, we describe how the requisite class *structure* can be systematically obtained from a correctly worded *behavior* description – i.e., the use cases.
- Test data and expected results, presented totally in the vocabulary of the problem.

Solution Framework artifacts include:

- Code skeletons for classes (in Java or your favorite OO language)
- A set of test cases representing the testing information from the stereotype, described in code.

- Sample code for some of the class methods. The pedagogical process must indicate how the code was systematically derived from the workflow descriptions in the use cases.

### 2.1 An Example from the Expression Evaluation Stereotype

The first example of an Expression Evaluation problem stereotype we used in our CS1 class was the “Area of a ring” problem. The students are provided the following artifacts:

**Problem Statement:** Assume that there are two concentric circles of different radii. The area between the outer and the inner circles is called a ring. Write a program to compute the area of the ring.

**Use Case:**

<b>Use Case Name:</b>	Find the area of a ring
<b>Description:</b>	
Find the area of the region that lies between two concentric circles	
<b>Preconditions:</b>	
/* What data should be known? */ Radius of the inner circle; Radius of the outer circle	
<b>Workflow:</b>	
/* What do you do with the data you are given? */ The Ring asks the Inner circle to compute its area using the standard formula. The Ring asks the Outer circle to compute its area using the standard formula. The Ring computes its own area by subtracting the inner circle area from the outer circle area.	
<b>Results:</b>	
/* What do you get back after doing this thing? */ The area of the ring	
<b>Alternates:</b>	
/* What can go wrong? */ 1. A radius is negative (Report error). 2. Inner Radius is > outer radius (Report error).	

**Figure 1: Use Case for computing the area of a ring**

At this point we demonstrate to students how to think in terms of objects. It is clear from the use case workflow description that we have 2 kinds of objects: *Circles* and *Rings*. Each circle object knows its radius and is capable of computing its area. A ring object encapsulates (i.e. knows about) exactly two circle objects. When the ring is asked to compute its area, it tells its components to do the same, and then subtracts the returned values. This helps us in identifying classes and methods and coming up with CRC cards. Example CRC cards are shown in Figure 2.

**CRC Cards:**

<b>Class Name:</b>	Circle
<b>Description:</b>	
Embodies what a circle knows and can do	
<b>Subclass:</b>	
-None-	
<b>Superclass:</b>	
-None-	
<b>Attribute:</b>	<b>Type:</b>
/* Attributes, types */	
radius	double
<b>Method:</b>	<b>Collaborating Classes:</b>
/* List Methods */	
getArea()	-none-

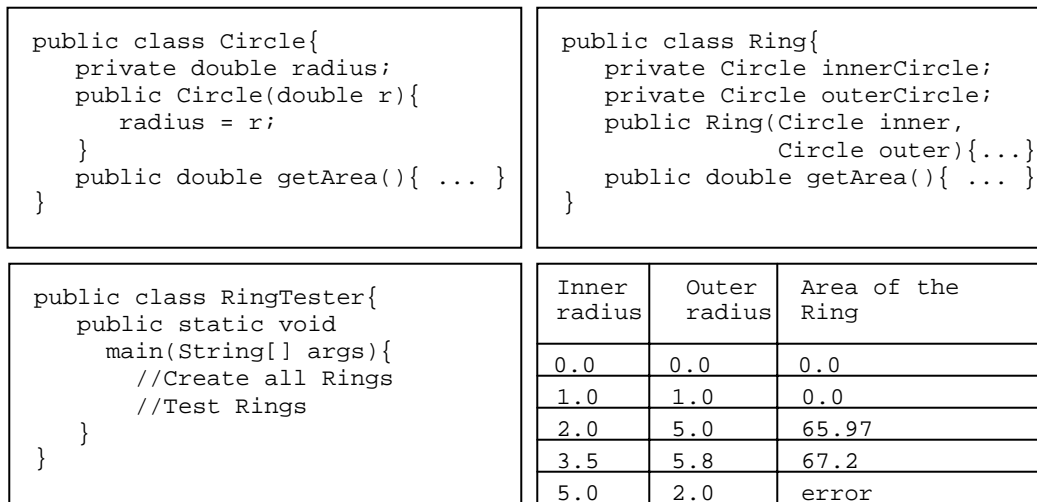
<b>Class Name:</b>	Ring
<b>Description:</b>	
Shows what a ring knows and can do	
<b>Subclass:</b>	
-None-	
<b>Superclass:</b>	
-None-	
<b>Attribute:</b>	<b>Type:</b>
innerCircle	Circle
outerCircle	Circle
<b>Method:</b>	<b>Collaborating Classes:</b>
/* List Methods */	
getArea()	-none-

**Figure 2: Typical CRC cards**

### 2.2 A Solution Framework for the “Area of Ring” Problem

The problem stereotype uses the object-oriented methodology, but is quite independent of the programming language (Java). The solution artifacts bring in the Java code. The solution framework we gave our students consisted of skeleton Java code and some test data as shown in Figure 3 (comments are

not shown). The solution framework also contains a *Tester* class that includes the main method whose only function is to implement the sequence of designed tests.



**Figure 3: Solution Framework, including test cases, for “Area of Ring” problem**

### 3. PROBLEM STEREOTYPES FOR CS1

This approach was used to teach the CS1 course at SUNY Brockport during 2005-06. All example problem stereotypes and solution frameworks were made available to the students online (see [8] for artifact templates and example programs). In an early lab exercise, the instructor went over all the artifacts of the "Area of a ring" problem, fleshed-out the skeleton code, and demonstrated the working program to students. The lab problem for that day was to write a similar program to compute the area of a frame (the area between two concentric rectangles). This was done in the third week of classes, at which time the students had a modest background in variables, constants, assignment statements, etc., but had no knowledge of methods or parameter passing. Students were asked to use a “cut and paste” approach to create their own artifacts (use cases, CRC cards, code skeleton, and test cases) and then they were instructed to flesh out the code. Almost all of them were able to create the artifacts and get the program to work very easily. Thanks to the fact that they had a modifiable example to work with, students did not have to put in huge amounts of effort. This gave them a significant sense of accomplishment.

After more practice with Expression Evaluation problems, we introduced another stereotype: problems involving selection statements. Students were then given their first *programming project*. The problem stereotypes for this project are the Expression Evaluation Stereotype and the Selection Stereotype. As this was still early in the semester (fourth week), students were provided with both the problem stereotype and solution framework and were simply asked to complete the code. These artifacts were discussed in class and made available online. This project was thought of as easy and most students were successful in completing it: 73% of the students got a ‘B’ or better on this project.

The second project was based on the same stereotypes. Students were first given just the problem statement. They were first asked to prepare the problem stereotype artifacts. After evaluating the student artifacts, instructor-prepared artifacts were made available. Students were given the choice of using their own artifacts, or the instructor-provided ones. About half the students abandoned their artifacts and used the instructor-provided ones for completing their code. On the whole, the students were quite successful in getting the code to work: 78% of the students got a ‘B’ or better.

In the course of the semester, we introduced many other problem stereotypes such as those involving processing a collection (introducing looping), and menu-driven user-interface. The third project involved writing a menu-driven program that performed search operations on a “database” of books. Students prepared all the artifacts and created a solution from the model on their own. A majority of students were successful in creating the artifacts and in writing and testing the code: 76% of the students got a ‘B’ or better.

### 3.1 Student Reaction to this Approach

At the end of the semester we conducted a survey about students' opinions on the use of our methodology. Table 1 summarizes the results of our survey from Fall 2005 (38 students) and Spring 2006 (34 students). Each question on the survey can be answered on a 0-5 scale where 0 = "Strongly Disagree" and 5 = "Strongly Agree." The opinions were mostly in favor of using this approach. We have to keep in mind that the class has a diverse group of students: CS majors/minors, Math, Physics, Earth Sciences and even Criminal Justice majors. For most of the non-CS majors this is the only required CS course.

Question (0=Strongly Disagree, 5 = Strongly Agree)	# of students	F'05	S'06
		38	34
Use Cases helped me to understand what classes I should have and what their responsibilities should be.		3.53	3.00
CRC Cards helped me to figure out the methods of each class before writing any code.		3.39	3.06
It was easy to write the skeleton code from the CRC cards.		3.92	3.74
Writing the skeleton code helped me to write the bodies of the methods more easily.		3.68	3.03
Test Data helped to figure out what methods were not working properly during debugging.		3.68	3.35
Following this methodology made it easier for me to write programs.		3.66	3.44
After taking this class I feel more confident about writing a program and making it work.		4.32	2.97
Do you recommend the use of this methodology in all programming classes?		27(y)	16(y)

**Table 1: Student Survey Results (Average Response) in CS1 (Fall 2005, Spring 2006)**

Students were also asked to write free format comments regarding the use of this methodology. Most comments were favorable, but some were not. Here is a sampling of positive comments: "Definitely Works!" "It creates more organization and allows the person to write code easier rather than doing all the work in their head," "I took a Java course at (another college) and this class is ten times clearer in helping you understand programming," "It is a lot easier to understand what the program is supposed to do using CRC cards and use case cards. Someone else that wants to modify you(r) code will understand a lot more by looking at you(r) use case and CRC cards." However, some students felt that it was too much *overhead* to prepare the design documents, and they thought they could have easily written the programs without them.

### 4. PROBLEM STEREOTYPES USED IN A HIGHER-LEVEL COURSE (CS4)

We used the problem stereotype and solution framework approach to teach an advanced CS course (CS4) at RIT during the Fall 2005 quarter. While CS1-3 are taught using Java, this one uses C++. Its goal is to provide an opportunity to learn C++, experience working as a team to solve more complex problems, and prepare for courses in software engineering. We chose a problem stereotype from AI search problems that can be solved using the *state-space search* technique. A variety of problems conforming to the abstraction associated with this stereotype, ranging from trivial to fairly complex, were assigned as projects.

**Problem Statement:** A puzzle has many possible configurations and a set of legal moves. When a legal move is applied to a configuration, it yields a new configuration (or null, if the move is not applicable). Starting and final configurations are specified. Find a solution path from a starting configuration of a puzzle to a final configuration, using only legal moves.

**Use Case and Other Artifacts:** Students were asked to think in terms of a *player* (an external agent who specifies start and goal configurations), a *solver* (who receives the start and the goal, solves the puzzle and hands back the solution) and a *puzzleAgent* (who talks to the player and the solver). The instructor made a set of problem stereotype artifacts available to the students (see [8]). The problem stereotype was complex enough to need a UML-style sequence diagram as well. Thus the artifacts provided were: a use case, a UML sequence diagram, and several CRC cards. (Note: We cannot show all the artifacts here due to space considerations, but they are available from the authors). The solution framework (prepared by the instructor) included a class hierarchy containing several abstract classes.

**Student Response:** We conducted an informal survey at the end of the semester. Most of the students who used the artifacts were able to successfully complete the implementations for all of the projects. One of these students made the following comment: "The solver took the most time. Once I got the solver to be

totally generic the second to last and last submission were trivial as you said they would be." This student's grades dramatically improved over the quarter. Several students have commented on how this approach helped them discover a good *abstraction*, and having done so, fit a different project problem into it, thus becoming more productive in generating high-quality code. We believe that this is possible due to the numerous examples oriented around a particular stereotype, presented in full, and the practice the students had with tackling very similar problems.

**Challenges for the Instructor and the Students:** Time constraints and lack of previous experience with this approach can severely restrict the number of examples presented for a problem stereotype, thus causing a significant challenge to both instructor and the student in advanced classes. We found that organizing the students in teams, and managing the team well, helps. An ideal scenario would be one in which the entire introductory sequence uses this approach, so that by the time the students are faced with a difficult project in a higher-level class, the design concepts are familiar and used with proficiency.

## 5. REFLECTIONS, CONCLUSIONS, AND FUTURE WORK

Student feedback, especially from the CS1 course, indicates that significant numbers of our students felt that this approach enabled them to learn software architecture and design skills right in the very beginning. They learned to capture these on paper and translate them systematically into code. This made the task of writing correct programs easier. Those who did not appreciate this approach mainly felt that they could write the code directly from the problem description, and these additional artifacts were mainly a "waste of time".

Software Engineering (SE) courses teach that while coding directly from fuzzy designs existing only in the mind might work for small problems, it is not practical when large complex problems are being solved. Code developed thus often has *maintainability* and *extensibility* issues. Such courses, however, typically appear in the senior year of a typical undergraduate CS curriculum, by which time students have several bad "coding habits" already ingrained in them. We believe that the essential SE message – "*Software does not live and die with the programmer who wrote it*" – must be sent early to the students. Therefore, in CS1-2, they must be taught that regardless of the simplicity of the system, it must be *designed* first, and this design must be *documented* in a manner that enables systematic code writing from it. Documentation in the form of in-line comments is not enough - it does not provide the "big picture". This message must be carried through to the later introductory courses. Students in these courses will be able to see the true advantages of this approach if they are given complex projects in which they are provided with working, well-designed and documented programs, and asked to *extend* it without doing much re-development. It is our goal to try and achieve this in one of our later introductory courses in a few semesters. To summarize, this approach introduces the essentials of SE philosophy early in the curriculum. The authors have also published work on effective techniques for teaching the traditional SE courses [7].

## REFERENCES

- [1] Proulx, V. K., Gray, K. E., Design of Class Hierarchies – An Introduction to OO Program Design, *Proceedings of the 37<sup>th</sup> SIGCSE Technical Symposium on Computer Science Education*, March 1-5<sup>th</sup>, 2006, Houston, TX, USA, pp. 288- 292.
- [2] Bennedsen, J., and Caspersen, M.E., Programming in Context – A Model-First Approach to CS1, *Proceedings of the 35<sup>th</sup> SIGCSE Technical Symposium on Computer Science Education*, March 3-7<sup>th</sup>, 2004, Norfolk, VA, USA, pp.477-481.
- [3] Bergin, J., Fourteen Pedagogical Patterns, 2000, <http://csis.pace.edu/~bergin/PedPat1.3.html>, retrieved January 17, 2007.
- [4] Wirfs-Brock, R., McKean, A., *Object Design: Roles, Responsibilities and Collaborations*, Boston, MA: Addison-Wesley Professional, 2002.
- [5] Fowler, M., *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 3<sup>rd</sup> edition, Boston, MA: Addison-Wesley, 2003.
- [6] Barnes, D. J., Koelling, M., *Objects First with Java: A Practical Introduction Using BlueJ*, Upper Saddle River, NJ: Prentice-Hall, 2003.
- [7] Mitra, S., Rao, T.M., and Bullinger, T.A., Teaching Software Engineering Using a Traceability-Based Development Methodology, *Journal of Computing in Colleges*, 20(5), June 2005, pp.249-259.
- [8] Canosa, R.L. The "Problem Stereotype and Solution Framework" Resource Center, 2006, <http://www.cs.rit.edu/~rlc/Stereotypes/>