THE PEDAGOGY OF PROGRAM DESIGN.

VIERA KRŇANOVÁ PROULX, PHD.

College of Computer and Information Science, Northeastern University,
360 Huntington Ave, Boston, MA, USA, tel. ++1-617-3732225, e-mail: vkp@ccs.neu.edu

ABSTRACT

We describe the pedagogy, the curriculum, and the software support for teaching systematic program design with emphasis on systematic testing and on understanding the connection between information and data. The curriculum begins with the BOOTSTRAP curriculum for children in grades 5-8, follows with the TeachScheme! segment for secondary schools and universities and extends to ReachJava, a full-scale object–oriented program design starting at the secondary level and extending to the software design at the university level.

The software support enables students to work with a language appropriate for their current knowledge of programming, it provides a framework for the design of interactive games that encourages creativity, and it supports the design and evaluation of tests appropriate for a novice programmer. The curriculum has been used in classrooms for a number of years and nearly all materials (software, the text, lab materials, worksheets, assignments) are available free on the web.

Keywords: *informatics, program design, object-oriented design, testing, games*

INTRODUCTION

Over the past fifteen years our team has been designing curriculum for teaching introductory programming and computing for students of all ages. The original curriculum labelled TeachScheme! has been used in secondary schools and universities throughout the world. Since I joined the team seven years ago I have worked on extending the curriculum to include program design in the object-oriented style, building a path to full scale software design curriculum. At the same time, other tem members designed a curriculum for children in grades 6–8 that combines the learning of computing and the basic algebra in the context of designing of a simple interactive game.

Our curriculum provides a path for learning that starts at the most introductory level, but leads to understanding deep concepts of program design at a very advanced level. The backbone of the curriculum at all levels is the pedagogy for describing the design process as a collection of **design recipes**.

This paper provides an overview of how the pedagogy of design recipes helps both the student and the teacher in all stages of learning.

1. DESIGN RECIPES

The *design recipe* is a series of steps a programmer should follow when designing a program, regardless of the expected complexity of the program. The recipe consists of several steps. At each step the programmer explores the next problem step by asking and answering a series of questions, producing an answer, and verifying the correctness of the answer. The pedagogical benefit to the learner is in the self-regulatory learning style the design recipe supports: the student is empowered to explore and solve the problem on her own, knowing how to proceed at each step. The benefit to the instructor is in the support for pedagogical intervention. When a student seeks help with a problem, the instructor can determine what step of the design recipe caused the difficulty, and lead the student through working out that part of the design recipe.

1.1. Design Recipe for Designing Functions

We start the programming instruction by designing functions. These are functions as seen in mathematics: each takes one or more arguments and produces a new value. Rather than changing the state of variables, our programs produce new values at each step. This type of programming (functional programming) has the advantage that the outcome of each function depends only on the values of the arguments, not its position within the program. It is easy to define the expected behavior of such functions. The *design recipe for functions* takes advantage of this functional behavior:

1. Analyze the problem, represent the relevant information as data.
2. Write down the purpose statement for the function, identifying clearly what data it consumes and what value it should produce. Write this information formally as a contract, then design the function header.
3. Make examples of the data the function will consume, write down the expected results. Do as many of these as needed to understand the problem.
4. Take an inventory of all available data and functions already defined that could be used in defining this function. This includes fields of complex data and functions that consume data of the type of data available to us as the function arguments.

5. Now design the body of the function. If the task looks to be too complex, divide it into smaller parts and make a *wish list* of functions you will need to solve the problem. (These functions will be designed later, following the same steps. For now we assume they have been defined already.)
6. Convert the examples from part 3 into formal tests and run them. Fix any errors and test again.

1.2. Design Recipe for Data Definitions

Without a solid understanding how the relevant information is represented as formal data students cannot design programs. A great deal of our curriculum focuses on understanding how information can be represented as data and how a piece of data can be interpreted to describe the information encoded within.

The *design recipe for defining data* guides the student through the process as follows:

1. Check if the information can be represented by one simple piece of data, such as a number, a symbol, or a String.
2. If several related pieces of information are needed to describe one object, combine them into a structure (later these become classes). Define a composite structure with one field per piece of information. (For example, a book with a title represented as a String, and a year of publication given as a number.)
3. If the information refers to another complex object, use containment. (This will be a structure or a class with a field that is an instance of another class: A book with a title, author (with name and year of birth), and a year of publication.)
4. If the information consists of several related object with some common features but with several variants that distinguish between them define a union (An shape is one of: a circle, a rectangle, a triangle). The define each member of the union separately. (In Java these turn into classes that implement a common interface or extend an abstract class.)
5. Connect fields with their data definitions as appropriate drawing containment and inheritance arrows.

1.3. Design Recipe for Designing Abstractions

Once the programs become sufficiently complex students begin to see that certain sections of code appear again and again, perhaps with a slight variation.

This motivates the design of abstractions. A list of books, or a list of persons is just a list of objects. When we sort data, we need to compare the two items and need a function that performs the comparison. Given the comparison function, the same sorting program works for any data.

The design recipe for abstractions provides the following guideline:

1. Compare two pieces of code that are similar, highlight their differences.
2. Replace each pair of differences with a parameter and rewrite the general solution.
3. Implement the solution to each of the original programs and run the tests.

2. COMPUTATION AND ALGEBRA

Our middle school curriculum is taught in the after-school programs by volunteer teachers. The class meets for 90 minutes for 10 weeks. Most of the students are weak in math. This is a very short time, yet the pupils design and implement an interactive game and on the final day not only show off their game, but talk about variables, conditionals, and function evaluation.

The goal is to design a game with one object moving as the time ticks and another object controlled through the arrow keys pressed by the game player. We restrict the movement of every object to one dimension (left-right or up-down). Collision of the two objects produces a special effect: the game may end, an explosion is shown, or the game score may be updated.

Children program the movement of the game components by defining functions that compute the position after one time tick, or the new position after a key has been pressed. They detect collision of two objects by computing the distance between them (we give them the formula for this). They learn what is a variable, how to evaluate expressions, learn a bit of geometry by defining the positions of their game components, learn about boolean values when detecting collisions or making sure the game pieces stay within the game canvas.

2.4. The Curriculum

The curriculum has been designed with two goals in mind. It has to speak to the child in his language, making sure the children are not distracted or overwhelmed by the programming language complexity. It also has to be divided into daily lessons that can be taught by volunteer teachers with a minimal training.

We use a special variant of the Scheme programming language. All values are color-coded, so that the children can easily distinguish between Strings, booleans, numbers, and symbols.

To explain the evaluation of expressions and functions we enclose each subexpression in a *CIRCLE OF EVALUATION.* The operation (plus sign, or a name of a function) is on the top and the values that are needed for the computation are shown below in the appropriate order. This translates directly into Scheme functions. that are evaluated immediately in the *Interactions* window of the *DrScheme* programming environment:

```
> (- 9 5)
4
```

More complex expressions are drawn as circles within circles and children learn to evaluate the innermost circles first.

Children quickly learn to convert an expression such as (+ (* 2 3) (- 8 6)) to the collection of circles of evaluation and compute the value of the expression. The variables are then shown with the name above and the substituted value below a horizontal line.

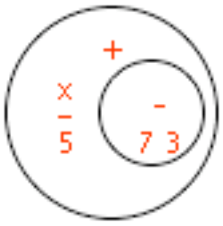

Fig. 1 Circles of Evaluation

Figure 1 shows the circle of evaluation for the expression *(x + (7 + 3))*, with *5* substituted for *x*. In Scheme this would be written as *(+ (x (- 7 3))*.

Students are motivated, as the goal is to design their own game. Children brainstorm about the game they want to design during the first lesson and the teacher finds appropriate images for every student's game by searching the clipart on the web. The supporting software library allows students to draw any image at the selected location. Children define the function onKeyEvent that produces the game scene after the player has hit the left or right arrow key, the function onTick that produces the game scene after one clock tick, and the function that detects the collision of the two objects.

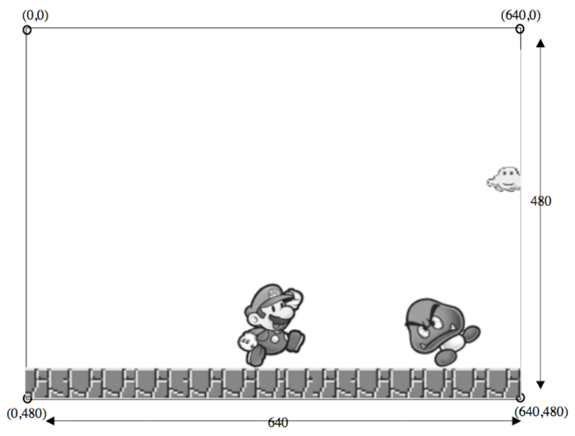

Fig. 2 A worksheet for Mario game

Children learn about conditionals, use the distance formula that we provide, and test their functions one by one as they design them. They follow the design recipe with worksheets where each step is carefully worked out and tested as a running program. The last day of the session they show their games to their parents, teachers, and friends, and answer questions about conditionals, functions, function evaluation, and their game design. We have seen some of these children succeed in math in their later years, but the number of students who have completed our program is too small to provide a statistically significant evidence of success.

3. UNDERSTANDING DATA

In the program design courses in the secondary schools and at the college level the first one or two weeks are similar to those of the younger children. However, to design programs of any complexity one must deal with sufficiently complex data. While the typical language-based programming instruction focuses on *algorithmics*, our curriculum is based on the premise that most of the production programs are driven by the structure of the data that they process. The art of defining data that represents the given information is the driving force behind program design.

3.5. Designing Data

The *design recipe* for data definitions allows us to define an ancestor tree as follows:

```
An Ancestor Tree (AT) is one of
- 'unknown
- Node : (make-node String AT AT)
(define-struct node (name mother father))
```

The *struct* definition allows us to build an instance of a complex data, provides a constructor (*make-node "Jan" mom dad)*, a predicate (*node? x)* to verify that a data item *x* is a *node* and a selector function for every field: *node-name, node-mother, node-father*.

For each new data definition we make sure students make examples of data and can interpret the data representation as information, as well as define new data items that represent the given information. For this example, students would convert their ancestor tree into data and read their friend's data and answer questions about the maternal grandfather, great grandmother, etc.

3.6. Data Driven Design of Functions

The first step in designing function is to identify the information that is available as well as the desired outcome, and understand how it will be represented as data.

In the second step students define the function header, a purpose statement that explains what the function will accomplish, and define the types of data the function consumes as arguments as well as the type of the result it will produce. We do not use assignment in the early weeks of the curriculum and every function produces a new value. This restriction greatly simplifies the program students write and allows us to reason about each function independently of its place in the program.

Before attempting to design the actual function, **the third step** of the *design recipe* asks that students make examples of the data that the function consumes, and explain the desired behavior by showing sample function invocations together with the expected outcomes.

To help define a function for more complex types of data **the fourth step** of the *design recipe* asks the students to decompose the problem into parts by designing an inventory. An inventory for the function that consumes complex arguments lists the fields of the function arguments as well as the functions that can be used with them (from earlier definitions). The inventory for a function that consumes an *AT* would be:

```
(define (fun at)
  (cond [(symbol? at) …]
        [(node? at)
              … (node-name at) …
              … (node-mother at) …
              … (node-father at) …
              … (fun (node-mother at))…
              … (fun(node-father at) )…  ]
```

If this function was counting the nodes in the tree (other than unknown) it would be clear that the two function applications to the mother and father subtrees will produce the count of known relatives in each ancestor subtree, and the rest of the computation trivially follows.

Only now, in **the fifth step** of the *design recipe*, do the students proceed with the design of the function body. If the task looks to be too complex, it is divided into smaller parts, with subtasks delegated to other functions, making up a *wish list.* Each function on the *wish list* must have the function header, contract, and purpose statement, so that it can be used in completing the original function design. Of course, later on we need to take care of the functions on the *wish list.*

Once the function has been designed, **the sixth step** of the *design recipe* directs the students to turn the examples defined earlier into test cases that are run and evaluated. Because every function produces a new value and has no side-effects, the tests only need to compare the expected value with the value produced by the function. The test design is simple. It is supported by a test harness that allows us to define the tests simply as:

```
(check-expect (count-nodes my-at)  5)
```

The test harness then reports which tests failed and shows the actual and expected values side-by-side.

In the early curriculum the structure of the functions students design follows exactly the structure of the data the function consumes. A large number of everyday programs are of this type: parsers, interpreters, programs that manipulate databases, GUI interactions programs, and more. We introduce algorithms that require some additional insight (e.g. quicksort) only after the students understand the program design driven by the structure of data.

3.7. Designing Abstractions

In order to build reusable programs we need to generalize: we need to abstract over the data type, over the functional behavior, the data structure manipulation, or the traversals over the data in a collection of data.

In both the early curriculum that uses *DrScheme* functional languages, and the early part of the class-based instruction we follow by introducing abstractions that simplify the program design and allow us to make the programs more general and reusable as libraries. In the Scheme-like languages we introduce the use of functions as arguments and the generalized Scheme loops such as *orMap, filter,* and *fold*.

We motivate abstractions by showing side-by-side two programs that are very similar, highlighting the places where they differ, and replacing the differences with parameters. So, for example a program that computes the sum of all numbers in a list and a program that computes the product look structurally the same, but differ in the base value and in the operator used to accumulate the resulting value. That means that the base value and the operator (a function of two arguments) become parameters for the generalized solution. The resulting general program can be then used to concatenate a list of Strings into one String, with the base value given as an empty String and the operator being the concatenation function.

To motivate the use of a function as an argument to another function we show how a sort function depends on the comparison function that determines the ordering of two items. Therefore, the comparison function becomes an argument to the sort function.

The design of abstractions depends in a substantial way on the language support for the abstractions we wish to discuss. On the other hand, the desire to design more general programs allows us to motivate specific language features that have been included in the language precisely to support such abstractions.

3.8. Curriculum Support

The *TeachScheme!* curriculum, supported by the *DrScheme* teaching languages and the text *How to Design Programs,* is used in over 700 high schools and universities both in the USA and in many countries throughout the world. The supporting software includes libraries for the design of interactive graphics-based games as well as the library for designing multi-player client games managed by a server and played over the internet.

Our students design several games during their first year. However, the functions that comprise the game are all designed according to the *design recipe* and are fully tested.

4. OBJECT-ORIENTED PROGRAM DESIGN

4.1. Understanding Data

The curriculum for our second course is known as *ReachJava*. A draft of the textbook *How to Design Classes* is nearly complete.

This course focuses on program design in a class-based language with object-oriented programming. Students already have a good understanding of basic functional program design. The first lectures of the course focus on designing classes. The *struct* definitions from the Scheme turn into class definitions. The unions of data defined in Scheme become classes that implement a common *interface*. The *struct* definitions that contain fields that are also *struct*s become classes with fields that are objects of another class.

We represent the class hierarchy as class diagrams using a simplified UML notation. Every class includes a full constructor. Every field is referenced as *this.xxx*. Students make examples of data and get to see that the data representation is in many ways similar to what they already know.

```
               +-------------------------------------------+
               | +---------------------------------------+ |
               | |                                       | |
               v v                                       | |
           +--------+                                    | |
           | IShape |                                    | |
           +--------+                                    | |
               / \                                       | |
               ---                                       | |
                |                                        | |
    ---------------------------------------              | |
    |                   |                 |              | |
+--------------+     +----------+   +------------------+ | |
| Circle       |     | Square   |   | Combo            | | |
+--------------+     +----------+   +------------------+ | |
| Point center |-+ +-| Point nw |   | IShape top       |-+ |
| int radius   | | | | int size |   | IShape bottom    |---+
+--------------+ | | +----------+   +------------------+
                 v v
              +-------+
              | Point |
              +-------+
              | int x |
              | int y |
              +-------+
```

Fig. 3 Sample class diagram

Students define an *Examples* class (with the default constructor) and write all examples of data there. Before going any further, they must be able to translate the given information into its representation as data, and be able to interpret the data as the information the data represents. The *Examples* class acts as a client to student's code.

The main new idea in designing methods for the object-oriented languages is that each method belongs to a specific class and the object that invokes the method is somewhat *invisible* in the method definition. We require that students formulate the method purpose statement by referring to *this* object and the *given* arguments, so that the role of all data the method consumes becomes clear. Within the method definition we reference all fields of the current class with the *this.* prefix:

```
// compute the price of this book
// with the given discount
double salePrice(double discount){
  return this.price * discount;
}
```

Again, we follow the same *design recipe*. The examples of method invocation make it quite clear how the method invocation works and what is the role of the object that invokes the method. For the first couple of weeks we still follow the mutation-free style of programming where the outcome of the method is always a new object or a primitive value.

The inventory part of the design recipe now lists all the fields in *this* class as well as all arguments to the method, any methods already defined for this class, and any methods that can be invoked by any of the fields already listed:

So, in the class *Node* that implements the *AT* interface we would have:

```
// does this AT contain a person
// with the given name?
boolean contains(String name){

/* Inventory:
 Fields:
… this.name …     -- String
… this.mother …  -- AT
… this.father …    -- AT

Methods:
… this.count() …   -- int

Methods for fields:
… this.mother.count() …  -- int
… this.father.count() …    -- int

… this.name.equals(String) … -- boolean
*/
}
```

We can see that there is a great emphasis on making sure students understand very clearly the meaning of data and its structure. Combined with the examples of data they defined earlier the design of the actual method body becomes straightforward.

4.2. Test-First Design

The third step in the *design recipe* asks students to make examples of the method invocation with the expected results. In the absence of mutation this is quite easy. Students just have to provide as expected result the instance of an object with the expected values.

When the examples turn into tests in the last step of the *design recipe*, we just need to make sure that the resulting object matches the expected one, field by field. We would like the test to follow the earlier style:

```
Cat cat = new Cat("Boots", "Sam");
check-expect (
   cat.newOwner("Jan")
   new Cat("Boots", "Jan"));
```

The problem is that the comparison of objects by their value is not supported by the popular object-oriented languages. All equality comparisons besides the reference (identity) equality have to be defined by the programmer. This may not be too difficult for simple classes with just a few fields. However, defining equality of two *AncestorTree* objects is not a simple task for a novice programmer. The problem is much worse if the classes allow for circularly referential data (a *Book* with the *Author* field, while the *Author* class has a *Book* field).

To make the test-first design possible we must provide software support for the design and evaluation of tests that matches student's knowledge of the language and his program design maturity.

Our students start during the first couple of weeks using *ProfessorJ Beginner* and *ProfessorJ Intermediate* languages within the *DrScheme* IDE. The *Beginner* language is a Java-like language that allows for a class to implement a single interface, it has no loop constructs and no assignment statement. It also does not support any field or method modifiers. However, it provides support for the test design similar to that available for the Scheme programs. The test case for the *newOwner* method mentioned earlier would be written as:

```
boolean testNewOwner =
   check cat.newOwner("Jan")
   expect new Cat("Boots", "Jan");
```

The program definitions are written in the *Definitions* window of the *DrScheme* IDE. The results of running the program are shown in the *Interactions* window. For Java-like programs the Interactions window shows pretty-printed display of all fields defined in the *Examples* class. Any failed test cases are shown in a separate *TestCase* window and show not only the failed test but also the values that have been compared together with a link to the failed test. There is almost no new syntax needed to define the tests, there is no work involved in evaluating the test cases or understanding how to report the results.

After about three weeks we move on to standard Java language using one of the common IDEs. (We have chosen Eclipse, but other schools use NetBeans, or even BlueJ). Without a similar support for test design, evaluation, and reporting, it would be impossible to continue with the systematic use of the *design recipe*. To support this test-first design throughout the curriculum we have designed and implemented a *tester* library for standard Java that provides support for the design of tests appropriate for a novice programmer. The *tester* library examines the user's program through Java reflection support and evaluates every test method defined in the *Examples* class. The test cases shown above would be written as:

```
boolean testNewOwner(Tester t){
 return
 t.checkExpect(cat.newOwner("Jan"),
     new Cat("Boots", "Jan")) &&
 t.checkExpect(cat.newOwner("Martin"),
     new Cat("Boots", "Martin"));}
```

The *tester* library compares the objects by their values, reports any failed test cases, pretty-prints both the actual and the expected value of the failed test and provides a link to the failed test. Optionally, it pretty-prints every field in the *Examples* class, or prints all test results (successes as well as failures).

4.3. Understanding Test Design

The industry has embraced Test Driven Design (TDD) as leading to a more granular, comprehensible, and testable design. The pedagogy of the *design recipe* is not TDD (we do not build stubs that are further refined until the method is designed) but designing tests first is the common basis for both, and the benefits are similar.

Teaching students to design tests before they design the body of a method helps students think about the problem and understand what data is involved in the computation. The examples help them understand what is the method expected to accomplish. The examples also point out the situations when the desired method is too complex and is trying to do too many tasks at once. This suggest to students to design helper methods that provide solution to some of the needed sub-tasks. Running the tests and observing the test reports helps students diagnose errors and design programs that behave as expected. We teach students good programming practice from the beginning.

Practicing test-first design in introductory courses is impossible without proper software support. There are three parts that have to work together when designing tests. First, the programmer must define what actual and expected values should be compared. This part is a natural part of program design and with the proper support should be an inseparable component of program design for all students.

The second part concerns the test evaluation. This requires that the actual and expected values are compared in the manner that is consistent with the user's expectations. This part is very difficult and would be an undue burden for a beginner student. Novice programmers cannot define correctly the equality comparison methods until they understand very well the underlying object model, know how to compare different types of data, how to detect circularity, etc. In Java, overriding the *equals* method also requires that the programmer overrides the *hashCode* method, adding another layer of complexity. While learning about the different levels of equality should be a part of learning how to design programs, such instruction should proceed in a gradual manner as appropriate for the students' level of competence. Meanwhile, the test evaluation should be managed by a library that matches students' understanding of equality and their mastery of the test case design.

The third step in practicing test-first design is the reporting of the test results. Typical test libraries may report that certain tests failed, and possibly provide links to the failed tests. However, they do not show the actual and expected values that have caused the failed test. The programmer can provide methods that display the data in a readable format (*toString* method), but again, this adds another burden, another routine task that student must do just to get the program running. Without the library support, student would have to do this before writing the first method, when the syntax overhead of just defining one class with a constructor is huge. Additionally, when the data may be circularly referential, the student must worry about breaking the circularity.

Our *tester* library has been designed to provide robust support for test design for students at all stages of their programming instructions. As student learn how to override the *equals* method, and how to define their own *toString* method, the *tester* library allows them to use their methods when needed, while using the library support for classes without their customized methods. Over the past year we have added new types of test scenarios to the library, responding to the needs of students using the first versions of the *tester* library. There are test methods that check whether the actual value matches one of the several possible random outcomes, whether the actual value falls within the given range of values, whether the invocation of a method by a given object throws the expected exception, and more. The latest version of the library provides a test coverage tool. We are working on designing special support for tests that change the state of an object and tests that verify changes in the structure of data.

The library has been used by students in five different schools (secondary schools and universities) and by over 200 students in our classes. At this time we use the *tester* library for our own software development on a daily basis.

5. EXPANDING HORIZONS

Most of the programs our students write rely on the *tester* library to display the results of the test evaluation and the values of the objects defined in the program. We do not process user input, as that is a task for a seasoned programmer who understands the intricacies of parsing the input String and converting it to meaningful data. Additionally, programs with the user input are very difficult to test, and are typically first tested with the simulated inputs. We add some exercises that handle user inputs later in the course, but still require that the program be tested on sample data apart from the actual input.

5.1. Creative Design

To make sure students see programs with user interactions and practice their independent design skills on interesting problems, we again provide a library that supports the design of interactive games. Our students focus on the game engine and a simple display of the game components on a canvas. We provide three libraries for Java programs. The first one supports the functional style of programming where students design the classes that represent the game state and implement methods that produce the new game state after a tick of the clock, *onTick*, and in response the key press by the user, *onKeyEvent*. They define the *draw* method that shows the game components as simple graphics objects on the given *Canvas*. Their game world extends our abstract *World* class that provides a *Canvas* for drawing, abstract methods *onKeyEvent*, *onTick*, and *draw*, and a *bigBang* method that starts the game on a *Canvas* of the given size running the clock at the given speed. Students do not worry about display panels, layouts, listeners, actions, etc., but instead, they concentrate on the design of the game logistics.

The second, imperative library is very similar, but the state of the World changes *onTick* and *onKeyEvent*. The third library provides the same user interface as the imperative one, but allows the students to convert their games into Java Applets.

The game assignments give the students motivation and an opportunity to decide for themselves how to organize the game components into classes, decide which class should be responsible for each task, where the collisions should be detected, etc. We conduct individual project reviews with every student after they have completed their first game and comment on their design flaws, the program style, and the test coverage of their programs.

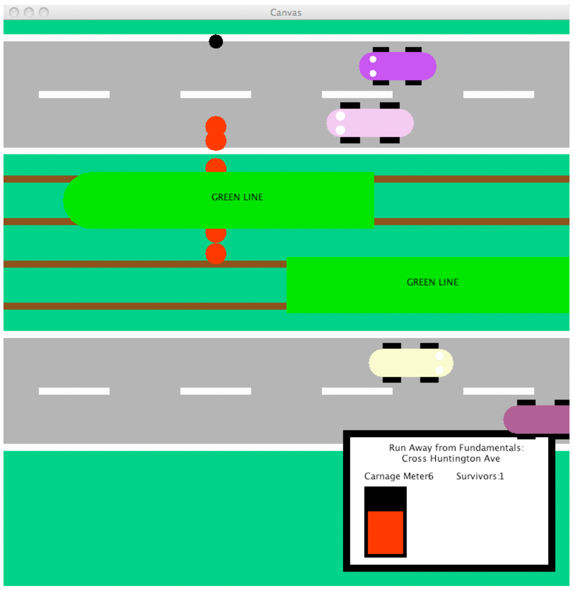

Fig. 4 *Crossing Huntington Avenue* game

5.2. Abstractions and Libraries: Reusable Code

Once the students understand the fundamentals of designing classes to represent data and designing methods for their class hierarchies, we use the repetition in their code as a motivation for introducing abstractions. Rather than starting with Java library classes, we first learn how to define abstract classes, interfaces that represent functional behaviour, interfaces that represent a traversal over a collection of data, classes that implement an interface that represents an abstract data type, and how to design generic algorithms that leverage the traversal and function objects.

Students who understand how to design programs with generalized behaviours learn quickly how to use libraries that have been designed on the same principles.

At this point we look at the cost of computation, learn about different data structures that support more elegant and efficient program design, learn about the complexity of algorithms, and run stress tests on large amounts of data to experience firsthand the differences between various algorithms and the underlying data structures they use.

5.3. User Interactions and GUIs

We believe that user interactions should be designed in a systematic way, just like the rest of model part of the program. At the end of the semester we introduce students to the JPT library that supports user interactions and GUI design through a cleanly designed abstraction layer. Students build an interactive GUI program with text fields, action buttons, sliders, colour choosers, radio buttons, graphics, key and mouse interactions, and a parsed console input after just one two hour lab session.

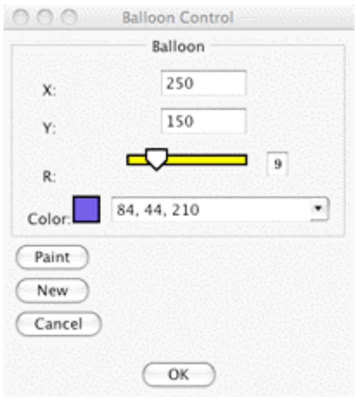

Fig. 5 A sample GUI panel

The *JPT* library includes *Java Power Framework (JPF).* Any class that extends JPF automatically generates a *GUI* with a *Canvas* for drawing and a panel that contains one button for every method with no arguments that returns *void*. A button click runs the method in the context of the class where it has been defined. This provides some for some interaction and supports experimentation.

The simplicity of the level of abstraction presented by the JPT library and the JPF tools makes it possible for students to master quickly the basics of GUI design and to incorporate user interactions into their final project of the semester.

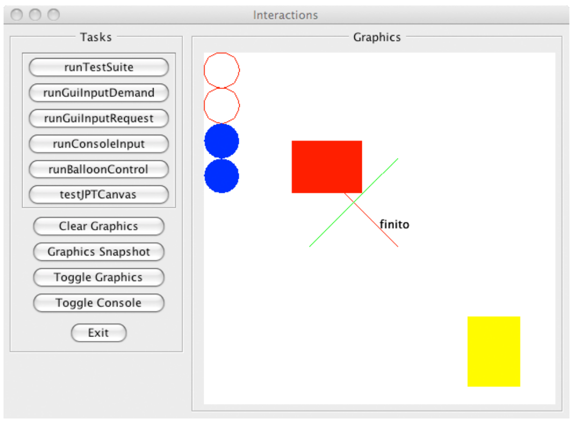

Fig. 6 *Java Power Framework* (*JPF*) window

6. CONCLUSION

The pedagogy of program design that drives our curriculum empowers the student and the instructor. In the early stages of learning to program the design recipes guide the students though the design process in clearly defined steps. Students learn to analyze the problem, define data, decompose complex problems into simpler ones, and they understand well the expected behaviour of every program they write. When students encounter a problem the instructor knows what questions to ask and how to help the student discover the answer and the solution.

The same pedagogy, the same *design recipe* that works for young children and novice programmers, helps seasoned programmers to find a well-structured solution. By writing tests for every method all students acquire a lifelong design discipline that will make their programs better and safer.

7. ACKNOWLEDGEMENTS

The *TeachScheme!* curriculum has been designed by Matthias Felleisen, with the support of the coauthors of the text *How to Design Programs*: Robby Findler, Matthew Flatt, and Shriram Krishnamurthi. They are also the founders of the *PLT* team that continues to develop the *DrScheme* programming environment and languages.

The *ProfessorJ* languages and their support for testing have been implemented by Kathryn Gray and Matthew Flatt.

Matthias Felleisen designed the curriculum for the *ReachJava* component that Viera Proulx implemented in the classroom for the past seven years. Felleisen is also the lead author of the draft How to Design Classes. Proulx has developed a large collection of laboratory materials, programming assignments and projects and a collection of lecture notes and sample programs.

Viera Proulx designed and implemented the *tester* library. Weston Jossey has joined the *tester* library team last year and has since designed and implemented numerous enhancements to the *tester* library.

The *JPT (Java Power Tools)* has been developed jointly by Richard Rasala and Viera Proulx. Richard Rasala has been working on the library enhancement and maintenace for the past seven years.

Useful links:
How to Design Programs: http://htdp.org
TeachScheme/ReachJava: http://teach-scheme.org
Java Libraries: (tester, draw, idraw, adraw, geometry, colors): http://www.ccs.neu.edu/javalib
How to Design Classes materials: http://www.ccs.neu.edu/home/vkp/HtDC/
JPT and JPF: http://www.ccs.neu.edu/jpt

REFERENCES

[1] Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: TeachScheme! project: Computing and programming for every student. Computer Science Education, vol. 14, no. 1, 2004, pp. 55-77.

[2] Felleisen, M., Findler, R.B., Flatt, M., Gray, K.E., Krishnamurthi, S., Proulx, V. K.: How to Design Classes. in preparation.

[3] Felleisen, M., Findler, R.B., Flatt, M., Krishnamurthi, S.: How to Design Programs. MIT Press, Cambridge, MA, 2001.

[4] Findler, R.B., Flanagan, C., Flatt, M., Krishnamurthi, S., Felleisen, M.: DrScheme: A pedagogic programming environment for Scheme. In H. Glaser, P. Hartel, and H. Kuche, editors, Programming Languages: Implementations, Logics, and Programs, volume 1292 of LNCS, Southhampton, UK, September 1977, Springer, pp. 36-388.

[5] Gray, K.E., Flatt, M.: ProfessorJ: a gradual introduction to Java through language levels. In Companion of the 18th annual ACM-SIGPLAN Conference on Object Oriented Programming Systems, Languages, and Applications, Anaheim, CA, Oct. 2003, pp. 170-177.

[6] Gray, K.E., Felleisen, M.: Linguistic support for unit testing: Report uucs-07-013. http://www.cs.utah.edu/research/techreports.shtml

[7] Proulx, V.K.: Test-Driven Design for introductory OO programming. In SIGCSE Bulletin, vol. 41, no. 1, 2009.

[8] Proulx, V.K., Gray, K.E.: Design of class hierarchies: An introduction to OO program design. In SIGCSE Bulletin, vol. 38, no. 1, 2006, pp. 288-292.

[9] Proulx, V.K., Rasala, R.: Java IO and testing made simple. In SIGCSE Bulletin, vol. 36, no. 1, 2004, pp. 161-165.

[10] Rasala, R., Raab, J., Proulx, V.K.: Java Power Tools: Model software for teaching object-oriented design. In SIGCSE Bulletin, vol. 33, no. 1, 2001, pp. 297-301

APPENDIX: SAMPLE CODE WITH TESTS

```
import tester.*;

// to represent a traffic light color
class Light{
  String color;
  int time = 0;

  Light(String color){
    this.color = color; }

  Light(String color, int time){
    this.color = color;
    this.time = time; }

  void setTime(int time){
    this.time = time; }

  // reduce the time available by one
  void tick(){
    if (this.time > 0)
     this.time = this.time - 1; }

  boolean off(){
    return this.time == 0; }
}

// to represent a traffic light
class Traffic{
  Light current;
  int currIndex;
  int[] times;
  Light[] lights =
    new Light[]{new Light("Red"),
                new Light("Yellow"),
                new Light("Green")};

  // the constructor
  Traffic(int red, int yellow,
          int green){
    this.times =
      new int[]{red, yellow, green};
    this.currIndex = 0;
    this.current =
      this.lights[this.currIndex];
    this.current.setTime(
      this.times[this.currIndex]); }

  // change the light:
  // make the next light current
  void newCurrent(int index){
    this.currIndex = index;
    this.current =
       this.lights[this.currIndex];
    this.current.setTime(
      this.times[this.currIndex]); }

  // update the clock by one
  // change light if necessary
  void tick(){
    this.current.tick();
    if (this.current.off()){
      this.newCurrent(
        (this.currIndex + 1) % 3); }
  }
}


// a class for sample data and tests
class Examples{
  Examples(){}

  // test the method setTime
  // in the class Light
  void testSetTime(Tester t){

    Light r = new Light("Red");
    r.setTime(5);
    t.checkExpect(r,
        new Light("Red", 5)); }

  // test the method tick
  // in the class Light
  void testTick(Tester t){
    Light r5 = new Light("Red", 5);
    Light r0 = new Light("Red", 0);
    r5.tick();
    t.checkExpect(r5,
        new Light("Red", 4));

    r0.tick();
    t.checkExpect(r0,
        new Light("Red", -1)); }

  // test the method off
  // in the class Light
  void testOff(Tester t){

    Light r5 = new Light("Red", 5);
    Light r0 = new Light("Red", 0);
    t.checkExpect(r5.off(), false);
    t.checkExpect(r0.off(), true); }

  // test the method newCurrent
  // in the class Traffic
  void testNewCurrent(Tester t){

    Traffic tr = new Traffic(5, 2, 7);
    tr.newCurrent(2);
    t.checkExpect(tr.currIndex, 2);
    t.checkExpect(tr.current,
        new Light("Green", 7)); }

  // test the method tick
  // in the class Traffic
  void testTickTraffic(Tester t){

    Traffic tr = new Traffic(5, 2, 7);
    tr.tick();
    t.checkExpect(tr.currIndex, 0);
    t.checkExpect(tr.current,
        new Light("Red", 4));

    tr.tick();
    tr.tick();
    tr.tick();
    tr.tick();
    t.checkExpect(tr.currIndex, 1);
    t.checkExpect(tr.current,
        new Light("Yellow", 2));

    tr.newCurrent(2);
    tr.tick();
    tr.tick();
    tr.tick();
    tr.tick();
    tr.tick();
    tr.tick();
    tr.tick();
    t.checkExpect(tr.currIndex, 0);
    t.checkExpect(tr.current,
        new Light("Red", 5)); }

  // start to run program here
  public static void
      main(String[] argv){

    // make an instance of Examples
    Examples e = new Examples();

    // report all test results
    // show all data in Examples class
    Tester.runReport(e, true, true); }
}
```