Java IO and Testing Made Simple

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

Richard Rasala
College of Computer and Information Science
Northeastern University
Boston, MA
rasala@ccs.neu.edu

ABSTRACT

We present software tools that support robust input processing and comprehensive testing in Java. The software includes the JPT library that supports error-checked typed input via console or GUI for all primitive types. This provides a robust encapsulation of typical interactive input requests encountered in introductory programming courses.

The Java Power Framework and its extension allow the user to develop a comprehensive test suite independent of the target classes. The type-safe input framework allows us to implement an external iterator interface for several types of input data sources, including the console, a GUI, a file, and an internal data structure. Student's programs that implement various algorithms can then process data independently of its source: running tests on existing data structures; creating inputs interactively; or running stress tests and timing tests on large input files.

Categories and Subject Descriptors

D.1.5 [**Software**]: Programming Techniques; D.2.10 [**Design**]: Methodologies; D.2.3 [**Coding tools and techniques**]: Object oriented programming

General Terms

CS 1/2, Object-oriented Issues, Curriculum Issues, Courseware

Keywords

Languages, Design, Algorithms

1. INTRODUCTION

Java programmers, especially novices, face numerous obstacles when trying to write programs that interact with the user. The patchwork of partial solutions is cumbersome to use, provides model of bad practice that students learn to accept and emulate, and does not scale to real applications.

We focus on three fundamental problems: the processing of interactive user input from the keyboard; the support for building test suites for complex class based programs; and the handling of the input from a file and output to a file at the proper level of abstraction.

In introductory courses the code that handles the user interactions is an order of magnitude more complicated than the code that focuses on the key concepts. Even the input of numeric values requires parsing of the input string and establishing a protocol for dealing with errors. To process the input of a collection of values that comprise the data needed to construct an object becomes a nightmare if appropriate abstractions are not in place. Some environments [7, 3, 16, 17] provide a solution in the form of GUIs, but these cannot be used outside of these environments.

Another problem related to user interactions is the support for developing test suites for class based programs. Some IDEs tried to address this problems by providing object viewers and a framework for interactive method invocation [17, 18]. It encourages students to interact with objects, but does not allow for saving the complete test suite for later use. Alternately, the test suite is built with JUnit [9] - a complex environment that students have to learn to use in addition to the overwhelming complexity of the Java language itself.

The third problem overlooked in most introductory texts and courses is a systematic use of files as sources and depositories of test data. Students write programs that either deal with an existing internal data structure or use only file input as the source of data. Producing output is similarly restricted. Ideally, programs that implement typical algorithms in the introductory courses should work with a data structure that can be initialized in an arbitrary manner: through input from a console or a GUI, from a file, or from an existing internal structure. This allows for testing for errors in the context of small data sets, and for stress testing of the same program on large data sets, without making any modifications in the program. It is also desirable that the resulting data set be deposited in a file, saved as an internal data structure, or displayed in an arbitrary manner, without the need for modifying the code for the underlying algorithms.

We believe that these three problems are intimately connected and can be solved with the same set of tools and techniques. We present such tools and techniques in this paper. At the core of these tools and techniques is the Java Power Tools (JPT) package [14]. JPT includes support for typed input for all primitive types through either a console or a GUI text field, with a comprehensive error strategy in place. This provides the appropriate infrastructure for building higher-level abstractions for input processing. While the implementation presented in this paper is Java based, the concepts are applicable to other object-oriented languages, such as Squeak or C#.

The first section discusses the design of the JPT input processing and the error strategy and shows how it can be used to implement higher level abstractions for interactive input of all data needed to define a new object.

The second section presents the Java Power Framework (JPF), a tool for rapid prototyping and systematic development of test suites for class based programs.

The next section presents the algorithmic abstraction that provides the programmer with a unified view of the input, regardless of whether the source of data is a file, interactive user input from a console or a GUI, or the contents of an existing internal data structure without requiring any change in the program. Output is handled at a similar level of abstraction.

The last sections compare our results with existing work and reflect on our experiences with using these tools in our courses.

2. JPT SUPPORT FOR TYPE-SAFE INPUT

Most higher-level programming languages provide only a limited support for processing the input of numerical data from the keyboard. While is possible and desirable to introduce programming without the use of explicit input[4, 5], the problem cannot be avoided forever. When writing programs that interact with the user through keyboard-based input, Java programmers, especially novices, face numerous obstacles. The input string needs to be processed, parsed, and cast to the appropriate data type. Dealing with errors compounds the complexity of the task. If the programmer wants to initialize an object with several data items specified by the user input, the task becomes even more complex. We first describe how JPT supports type safe input of Java primitive data, then show how the JPT tools can be used to implement type safe input of all data items needed to instantiate a new object.

2.1 Input of Primitive Types.

Keyboard input from the user is entered either into the console, or into a text field in a GUI. JPT supports its own `console` for user interactions, separate from the `System` console and has a `TextFieldView` class that implements a GUI text field with additional functionality.

To process input entered as a `String` and representing data of any of the primitive types, the JPT library includes a parser that determines whether an input string represents a valid value of the specified type. The user may type in numbers in any of the valid formats, or as an arithmetic expression that may contain a number of standard functions available in the Java math library, such as `sin`, `sqrt`, `abs`, etc. Similarly, a `boolean` value may be entered as any valid expression that evaluates to a `boolean` value.

The classes `console.in` and the `TextFieldView` both contain methods of the form `demand`*`Type`*, one for each primitive type. These methods parse the input and if the data cannot be interpreted as a valid value for the specified type, the user is prompted to supply correct input. In the console, the error prompt is textual; for the `TextFieldView` the program generates a modal dialog that persists until a valid string is typed in. The class `TextFieldView` also contains similar methods of the form `request`*`Type`*, that give the user the option to cancel the input. In that case the method throws an exception. The user can abort input from the console, if the programmer uses the `reading` method. This method returns the value of the desired type as the method argument (cast as a mutable object that represents the value of the primitive type) and returns a `boolean` value `true` if the user supplied valid input. If the input is invalid, the user is prompted to resubmit the input. If the user hits return without any input, the method returns a `boolean` value `false`, indicating the end of the user's input.

Student code is then simple and robust at the same time:

```
    int x = console.in.demandInt("Next:");
```

or

```
    int x;
    try
        {x = xTextFieldView.requestInt();}
    catch(CancelledException e)
        {System.out.println("No More Input");}
```

The optional `String` argument to the `demand`*`Type`* methods for the console is displayed as a prompt to the user.

2.2 Input of Compound Data

Typical Java programs use objects that require several different data fields for their constructors. If this data is provided by the user either through the console or through a text field in a GUI, the code for processing the input becomes even more cumbersome. Furthermore, without proper encapsulation it is difficult to test the program on existing data and run the same code without change with input from the keyboard.

From the programmer perspective, the interaction with the user input should look the same as if the input was extracted from existing internal data model. At the time when a program needs the data for the next object, it should just request that a new instance be created. The fact that the user types on a keyboard or performs mouse manipulations is irrelevant.

Using the JPT input processing tools, it is possible to write a helper method for each class that collects inputs for several fields of an object (e.g. name, age, eye color) and invokes a constructor that delivers an instance of the desired class. The source of the input can be either the console or a collection of GUI fields, or a combination of these. GUI fields may include options selection, check boxes, color views, sliders, or menu choices as well. A helper method for input of a `Person` object may have the header:

```
    Person demandPerson(String prompt);
```

or

```
    Person requestPerson(String prompt)
        throws CancelledException;
```

The method `demandPerson` can then be implemented as follows:

```
Person demandPerson(String prompt){
    console.out.println(prompt);
    return new Person(
        console.in.demandString("Name:"),
        console.in.demandInt("Age:"),
        console.in.demandColor("Eye Color:"));
    }
```

While this approach does not support formatted input of several data fields on one line, it works very well in the learning environments and eliminates the complications of input processing. Furthermore, students have a model of well designed input processing and are free to "look under the hood" and learn how such a robust system is implemented.

3. JAVA POWER FRAMEWORK (JPF) FOR BUILDING TEST SUITES

A typical student program may contain a number of classes and interfaces that interact with each other either through containment or a union. For example, a program that sorts lists of books and authors by different attributes may include the following classes: `Book`, `Author`, `BookList`, `EmptyBkList`, `ConsBkList`, `EmptyAuthList`, `ConsAuthList`, comparators: `ByTitle`, `ByAuthor`, `ByPrice`. It is clear that the code that creates the lists of books and authors and invokes the sorting does not belong to any of these classes. The user of these classes is some external class. Therefore, the code that tests this class hierarchy should also be logically separated from the target code.

For production purposes one should build a test suite that runs a comprehensive set of tests for all constructors and methods in the class without any interaction with the user. However, at the time when students are developing the class hierarchy, it may be preferred that they can see and run the test for each method or a constructor independently of the rest.

JPF provides the environment for either of these options. The Java Power Framework and its extension JPFalt creates an environment for running and testing of Java programs. The user's program consists of all classes in the program to be tested, together with one class that holds the test suite.

In the test class the programmer instantiates sample objects for the target program and encapsulates in separate methods the code for each different set of tests. Running of the JPF program then opens a JPT console, and creates a simple GUI with a list of action buttons - one for each method in the test class that requires no arguments and returns `void`. A programmer can perform the desired tests in an arbitrary sequence by selecting the appropriate action button. Additionally, the JPF creates a graphics panel that can be used for display of graphics and images, and for the input of mouse events.

The JPF is built as follows. The `TestSuite` class contains its `public static void main` method that constructs an application that has this `TestSuite` instance as its member data. The application uses the Java reflection classes to build a GUI with a button for each method in the `TestSuite` class that is not `private`, has no arguments and returns `void`. In this manner, each method in the `TestSuite` class can contain the test code for arbitrary use of classes in the target class hierarchy. Additionally, instances of the classes in the target hierarchy that are used in several tests can be defined as member data in the `TestSuite` class.

Of course, the name of this class does not have to be `TestSuite` - a change in the class name only needs to be repeated in the comment line and in the argument to the constructor for the application class, called in the `public static void main`.

The application also builds a graphics panel that can be used for graphics and for mouse interactions, and the `TestSuite` class also has access to the JPT `console`. The user can save the contents any part of the console interactions, as a text file.

The code in the `TestSuite` class provides a comprehensive view of all tests students ran, with proper documentation. The transcript from the console interactions provides a record of the test outcomes.

Students insert the test code into the following skeleton of the `TestSuite` class:

```
/* TestSuite.java */
public class TestSuite extends JPFAlt{
  public static void main(String[] args){
    new TestSuite();
  }

/* Place to instantiate objects
   in the target class */

   ...

/* Place for methods that test
   the target class constructors and methods */
   ...
}
```

To emphasize proper test development for our students, in JPFalt we augmented the original JPF with convenience methods that allow for specifying the expected and actual values as follows:

```
    void testIsWithin(){
        testHeader("isWithin");
        Circle c = new Circle(0, 0, 10);
        expected(true);
        actual(c.isWithin(3, 4));
        expected(false);
        actual(c.isWithin(5, 12));
    }
```

and producing the following output in the console:

```
    Testing the method isWithin:
    Expected: true
    Actual:   true
    Expected: false
    Actual:   false
```

Each of the methods `expected` and `actual` is overloaded to consume as argument a value of any of the primitive types, or any Java `Object`. For Java `Objects` it uses the `toString` method to generate a `String` representation of the object. This feature is exploited in all programs our students write, by requiring that each class contains a boilerplate implementation of the `toString` method modelled after the Felleisen and Friedman [6]. In this way students can see the relevant values of the objects they work with, either after the construction, or after some mutation has been performed, or, before some method is invoked.

4. LEVERAGING THE EXTERNAL ITERATOR INTERFACE

The lack of comprehensive input-processing strategy also means that students rarely use file input in a systematic way. Textbooks for introductory courses in Algorithms and Data Structures typically say very little about the use of file input, or omit the issue altogether [2, 12, 15, 13, 19]. However, the only way students can understand the meaning of algorithmic complexity and the need for stress tests for their programs is to run their programs with realistic large data sets that illustrate the concepts presented in a theoretical framework.

We developed a clean way of encapsulating the processing of input data into a traversal of the data through an iterator. This iterator is then implemented for arbitrary source of data: the console, a GUI, an internal data structure, or an external file. As a result, students write programs that process data regardless of its origin. When developing the program, students run tests on predefined sample data sets, or interact with their programs through a GUI or the console input. Once they believe the program works correctly, they perform stress tests using large files of data, without changing their programs.

We replaced the Java `Iterator` interface, by our `IRange` interface whose functionality matches the control structure of a Java `for` loop:

```
interface IRange{
    Object current();
    void next();
    boolean hasMore();
}
```

The main difference is that `current()` may deliver the same object several times, while `next()` only advances the iterator to reference the next item in the data set.

The method that uses the data set will have the following structure:

```
    Datatype useDataSet(IRange it){
        // ... code to initialize the loop ...
        for (it;
            it.hasMore();
            it.next()){

            ... it.current() ...
        }
        // ... code to clean up ---
        // and return the result ...
    }
```

It can then be invoked as follows, using either data from an internal structure, from the console, or from some given file:

```
Dataset ds1 = useDataSet(new MyDataRange(adataset));
Dataset ds2 = useDataSet(new MyConsoleRange());
Dataset ds3 = useDataSet(new MyFileRange(fileinfo));
```

where the three classes `MyDataRange`, `MyConsoleRange`, and `MyFileRange` implement the `IRange` interface for the three different sources of data.

4.1 Iterator Implementation for File Input

Most of the work needed to read the data from a file is done in two methods: the constructor and the method that parses the input line and produces the data items needed to instantiate the desired object.

For parsing a line of input that represents all data needed to instantiate a Person object, we use a method with signature

```
Person fromStringData(String line);
```

that parses the input line, extracts the name, age, and eye color data, and invokes the constructor to produce a new `Person` object. This method may reside within the class `Person`, or it may appear within the class that implements `IRange` iterator for files in which each line of data represents one `Person` object.

The constructor for the `PersonFileRange` is responsible for opening the desired file, initializing a `BufferedReader` and invoking the `next()` method to make sure a `Person` object is available when the `current()` method is invoked. If the constructor fails to construct a `Person` object for whatever reason, it sets internal state variables that in turn cause the `hasMore()` method to return `false`.

The rest of the implementation is straightforward. The method `current()` just delivers the available `Person` object. The method `next()` invokes the `fromStringData()` method on the next line of input, if it is available. Otherwise, it signals the end of input though internal state variables.

4.2 Iterator Implementation for Console

The constructor for the console input must read the first set of data to instantiate the desired object, even if it seems to be 'out of place'. The reason is, that the loop cannot proceed, if the `hasMore()` method returns `false`, and this needs to be known at the beginning of the loop.

The reading of the data leverages the `reading(Object o)` method for the `console.in` class to signal the end of input whenever the user fails to supply additional data, and to throw an `IOException` with a message `"End of Input"` to prevent any further request for data items for this object. The wrapper method that in turn asks for each data item catches the exception to print the `"End of Input"` message.

The rest of the implementation is straightforward.

Input from a GUI, or even a combination of console and GUI together is then done in a similar manner. The code that implements the `IRange` iterator for various kinds of input sources is itself a nice example for students to study and adapt in new settings.

4.3 Pedagogical Considerations

The `IRange` iterator interface has been designed to match the control structure of a Java `for` loop statement. In our courses students first experience list processing in a purely object oriented manner - through recursion that leverages dynamic dispatch of method calls over the `EmptyList` and `ConsList` subclasses of an abstract `AList` class. The transition to an external view of the elements of the list is an easy one. They also understand that an iterator object "self-destructs" during the traversal. This sets the stage for several lessons. First, it becomes clear, that the iterator object is best instantiated at the beginning of the loop. The fact that we may be processing the first request for data a bit prematurely is of little consequence. A pedantic pedagogue may explain the gory details, but for a novice programmer this is of little consequence, and can be explained as a simple compromise that was made so that we can focus on more important issues.

At this point students understand how the iterator is implemented. We now introduce Java `Array` class, and use the same iterators for the first traversal. Study of the implementation of the `IRange` iterator for `Array` data set then leads to the simplification of the traversal in the manner of typical counted loop.

5. COMPARISON WITH OTHER WORK

Processing of the input has been a nemesis of the introductory courses for years. A recent discussion on the SIGCSE mailing list produced no satisfactory solutions. A number of attempts have been made to simplify the input processing, whether from the keyboard or from a GUI, simpleIO [10] and BreezyGUI [11] being two examples.

We mentioned earlier the environment designed to simplify student's interaction with Java programs. BlueJ [17] is the most popular, others include MiniJava [16], a graphics based environment from Williams College [3], and DrJava [1]. They all provide only a partial solution to the problems addressed in this paper.

There is no environment we know that integrates the support for systematic user interaction that includes permanent record of the test suite, support for type safe input, and support for seamless processing of file input and output. We believe that the combination of the well designed tools with well designed abstractions presented here provide a solid foundation for introducing input processing to novice programmers. A promising work in this direction is ProfessorJ [8] — a new set of Java-like languages implemented within the DrScheme [7] environment.

6. EXPERIENCES IN THE CLASSROOM

We have used these tools and techniques in courses taught both by the authors and by other instructors. Students readily understood the environment of the `TestSuite` and learned from the beginning to design and document tests for all constructors and methods. The ability to add any test, or some sample code to the existing test suite encouraged student's exploration of available alternatives. For the instructor it provided a clean record of student's work.

The iterator interface for reading data sets from multiple sources was used in timing trials on sorting algorithms applied to a set of cities with zip codes. Students first tested their work on small internal data sets, then ran stress tests and timing trials on files of up to 30000 entries. One implementation that was based on a recursively defined list failed to complete the task when the file had about 30000 items.

Students learn from seeing and using abstractions that provide tangible and significant benefits. Replacing seamlessly one data set with another without modifying their program illustrates vividly the need for the separation of the algorithm implementation from the data source. The encapsulation of the input processing promotes proper design discipline of separating the model from the view.

7. ACKNOWLEDGMENTS

The first author would like to thank Matthias Felleisen for helping her discover a new way to see and understand programming and teaching. We also acknowledge the support of the NSF CCLI-EMD DUE-9950829 Grant: *Computer Science Laboratory Projects: Breadth Through Depth.*

8. REFERENCES

[1] E. Allen, R. Cartwright, and B. Stoller. Dr.Java: A Lightweight pedagogic environment for Java. *SIGCSE Bulletin*, 34(1):137–141, 2002.

[2] D. A. Bailey. *Java Structures.* McGraw Hill, 2 ed., 2003.

[3] K. Bruce, A. Danyluk, and T. P. Murtagh. A library to support a graphics-based object-first approach to CS1. *SIGCSE Bulletin*, 33(1):6–10, 2001.

[4] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs:An Introduction to Programming and Computing.* MIT Press, 2001.

[5] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. Structure and interpretation of the computer science curriculum. *FDPE*, 2002.

[6] M. Felleisen and D. Friedman. *A Little Java A Few Patterns.* MIT Press, 1998.

[7] 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.

[8] 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.

[9] Junit framework http://www.junit.org.

[10] E. Koffman and U. Wolz. A simple package for gui-like interactivity. *SIGCSE Bulletin*, 33(1):11–15, 2001.

[11] K. A. Lambert and M. Osborne. *JAVA Complete Course in Programming and Problem Solving.* South-Western Educational Publishing, 2000.

[12] Y. Langsam, M. Augenstein, and A. M. Tennenbaum. *Data Structures Using Java.* Pearson Education/Prentice Hall, 2003.

[13] D. S. Malik and P. S. Nair. *Data Structures Using Java.* Thomson Course Technology, 2003.

[14] R. Rasala, J. Raab, and V. K. Proulx. Java Power Tools: Model software for teaching object-oriented design. *SIGCSE Bulletin*, 33(1):297–301, 2001.

[15] D. D. Reily. *The Object of Data Abstraction and Structures Using Java.* Pearson Education/Addison Wesley, 2003.

[16] E. Roberts. An overview of MiniJava. *SIGCSE Bulletin*, 33(1):1–5, 2001.

[17] J. Rosenberg and M. Koelling. http://www.bluej.org.

[18] J. Rosenberg and M. Koelling. Objects first with Java and BlueJ. *SIGCSE Bulletin*, 33(1):429, 2001.

[19] P. T. Tymann and G. M. Schneider. *Modern Software Development Using Java.* Brooks/Cole, 2004.