Little Smalltalk in Java

Phani Srikanth Pandrangi
Department of Computer Science
Oregon State University
Corvallis, OR 97331
pandraph@research.cs.orst.edu

A research paper
submitted in partial fulfillment of
the requirements for the degree of
Master of Science

Major Professor : Timothy A. Budd
Minor Professor: Toshimi Minoura
Other Committee Member : Bella Bose

Little Smalltalk in Java

Phani Srikanth Pandrangi

1.0 Abstract

Little Smalltalk is a small, reasonably fast, easy-to-understand, easy-to-modify Smalltalk system. The system was originally developed in 1984 as a part of an implementation project to develop a minimal Smalltalk system, closely resembling Smalltalk-80. The system was developed in C. The current project is an experiment to port the Little Smalltalk system to Java with the dual-aim of testing the usefulness of Java for developing interpretive systems and also trying to make Little Smalltalk system reach a larger audience thru the world-wide-web.

2.0 Little Smalltalk - An Introduction

2.1 History

Little Smalltalk was created by Dr.Timothy Budd and a group of graduate students in 1984 as part of a graduate level seminar on programming language implementation. The goals of the implementation project were :

- Support a language that is as close as possible to Smalltalk-80.
- Make the system run under Unix using conventional terminals.
- Write the system in C so that it is as portable as possible.
- Develop a small system.

The system was succesfully developed and the goals were met. The system was extremely portable and has been transported to many varieties of Unix running on different machine architectures.

2.2 Philosophy

The guiding principle behind the development of Smalltalk was the idea of recursive design - "making the part have the same power as the whole". The basic idea of object-oriented programming is computation viewed as a process carried out thru interactions of a collection of autonomous interacting computing units. This idea was the basic principle behind the development of Smalltalk. The contribution of Smalltalk was not so much the language per se, but the philosophical approach embodied in the idea of object-oriented programming.

Little Smalltalk, although a scaled-down version of Smalltalk, follows the basic philosophy of object-oriented programming :

- Everything is an object.

  There is no way to create within the language an entity that is not an object. This uniformity created both the simplicity and power of the language.

- The Smalltalk philosophy (Byte 81)

  "Instead of a bit-grinding processor raping and plundering data structures, we have a universe of well-behaved objects that courteously ask each other to carry out their various desires."

  There is a universe of objects and the process of computation is initiated with a message and the computation is performed by sending messages to and fro until the work is done.

- Smalltalk is typeless.

  Any identifier can be used to refer to objects of any type and can be changed at anytime to refer to objects of different type.

- Objects have unscoped lifetimes.

  Objects can exist outside of procedure invocation (if we take message passing in Smalltalk equivalent to procedure invocation) and may persist for indefinite periods of time.

- Smalltalk is interactive.

The user is free to create or modify identifiers at run-time. Basic features like class descriptions may also change during execution.

- Smalltalk is a multiprocessing language.

  It is possible for a user to specify a number of different processes and have them execute concurrently.

2.3 Differences between Little Smalltalk and Smalltalk-80

The main differences between Little Smalltalk and Smalltalk-80 are listed below :

- Browser

  Little Smalltalk does not include the programming environment and the browser that is a characteristic of Smalltalk-80.

- Standard Library

  The Little Smalltalk system contains fewer classes when compared to the Smalltalk-80 system. For example, lack of browser obviated the need for some of the classes.

- Internal Representation

  The internal representation of objects, bytecodes, processes in Little Smalltalk is completely different from that of Smalltalk-80.

- Primitives

  Both syntax and the use of primitives is different in Little Smalltalk compared to Smalltalk-80. Primitives cannot fail in Little Smalltalk.

- New Syntax in Little Smalltalk

  1. Byte Arrays. The syntax is pound sign followed by a left square brace, followed by a sequence of unsigned integers in the range 0 to 255, followed by a right square brace.

  2. Primitives. The new syntax permits specification of primitives with arbitrary number of arguments. Primitives are of the form <primitive_number argument_list>

- New Semantics in Little Smalltalk

1. Cascades. The result of a cascaded expression is always the result of the expression to the left of the first semicolon.

2. Primitives do not fail in Little Smalltalk. They must return a value.

- Some features missing in Little Smalltalk

1. Class Protocols. Only instance protocol can be redefined.

2. Pool variables, global variables and class variables are not supported.

3.0 Motivation for Version 5 [Little Smalltalk in Java]

The motivation for the Java version of Little Smalltalk is two-fold :

1. Test the usefulness of Java in developing interpretive systems.

   Java, being a new language, is looked at skeptically by many people. Several people have been trying to apply Java to various types of applications to try to analyze its usefulness and its future.

   How useful is Java in developing interpretive systems? What are the features in Java that help in developing systems like Little Smalltalk? What are the features (or lack of features) that impede development of interpretive systems?

   These are some of the questions we will try to look at in this project.

2. Make Little Smalltalk available to a larger audience.

   The unique "applet" feature of Java allows us to make the Little Smalltalk system available on various platforms, without the need for porting to each platform. The system will now run on all systems on which Java runs.

   JDK (Java Development Kit) is currently available for SPARC/Solaris 2.3,2.4,2.5, X86 Solaris 2.5, Microsoft Windows NT, Microsoft Windows 95 and MacOS.

   Browsers supporting Java on all these platforms are available. This makes the Little Smalltalk system available to more audience than before.

3.1 Advantages of Java version

- Applet

  "Applets" are Java applications that run inside a Java-enabled web browser. By making the Little Smalltalk interpreter run in an applet, we take advantage of the platform independence provided by Java. With the growth of internet and world-wide web, the use of web browsers has become very common. With the advent of Java technology, there is an increasing number of web browsers supporting Java. Therefore by making the interpreter available on the web, we can reach a larger audience.

- Automatic garbage collection

  The previous versions of Little Smalltalk had to take care of garbage collection by themselves because of the lack of automatic garbage collection in C. Most of the code for the system involved garbage collection routines. Java provides automatic garbage collection. The runtime system of Java performs the memory management tasks and therefore the programmer is relieved of the memory management taks.

- Object-Oriented Design

  The current version takes advantage of the object-oriented features supported by Java. This helps in making any changes or upgrades to this version to be done with relative ease. Some of the advantages of the object-oriented design :

  1. Representation of objects : The internal representation of objects can be changed without affecting the working of the interpreter. For example, instead of using a vector to hold instance variables and temporaries, we can choose to use a linked list. Only the methods *addData*, *insertElement* and *getData* of the class representing objects (**LSTgenObject/LSTbyteObject**) need to be changed. The interpreter doesn't need any changes.

  2. Changes to the GUI : The graphical user interface of the system can be changed without effecting the working of the interpreter. Any changes to the GUI are made only in the **Smalltalk** class.

3.2 How useful is Java for developing interpretive systems?

This project seemed to be an ideal candidate to test Java's usefulness, it strong and weak points as far as the development of systems like this is concerned.

- Interpretive systems are usually slow. Is Java going to make them "slower"?
- How useful is Java's automatic garbage collection in design? How useful is it in performance of the system?
- What are the features that make Java "slow" and why?

These are some of the issues we examine in this project.

4.0 Little Smalltalk in Java

4.1 Introduction

The Little Smalltalk system requires, to start, a snapshot representation of the memory. This snapshot representation is called an **object image**. Several modules are combined to form the initial object image. The modules include

- module containing basic classes and methods.
- module containing methods for those objects having magnitude, which are the basic subclasses of class *Magnitude*.
- module containing methods for the collection subclasses.
- module containing methods and classes used for file operations.
- modules containing code which is used to initialize the initial object image.
- modules containing code for the multiprocessing scheculer.

The object image for this version is the same as the previous version of Little Smalltalk. So, this versions just makes use of the old object image.

The Little Smalltalk system can be broadly divided into two parts - the interpreter and the user interface. The interpreter is the bytecode interpreter which does the actual work. Given a message, the interpreter collects the necessary components for executing the method associated with the message, namely the bytecodes for the method, receiver of the message, literals and the context needed by the method, the stack to be used by the virtual machine executing the method etc., and produces the result. The user interface is basically text-oriented and acts as a simple read/eval/print loop. The user interface on the web page essentially consists of a text area for displaying results and test field to accept input.

The Little Smalltalk (Version 5) applet is on the web page at the URL

http://www.engr.orst.edu/~pandraph/Smalltalk.html

4.2 User Interface

The user interface part of the Little Smalltalk system uses the AWT classes of Java. AWT stands for Abstract Windowing Toolkit and it provides many useful classes for writing Graphical User Interfaces.

The user interface of system consists of two main components-the text area and the text field. The text area is basically used to display the output of the read/eval/print loop and the text field is used to read the input.

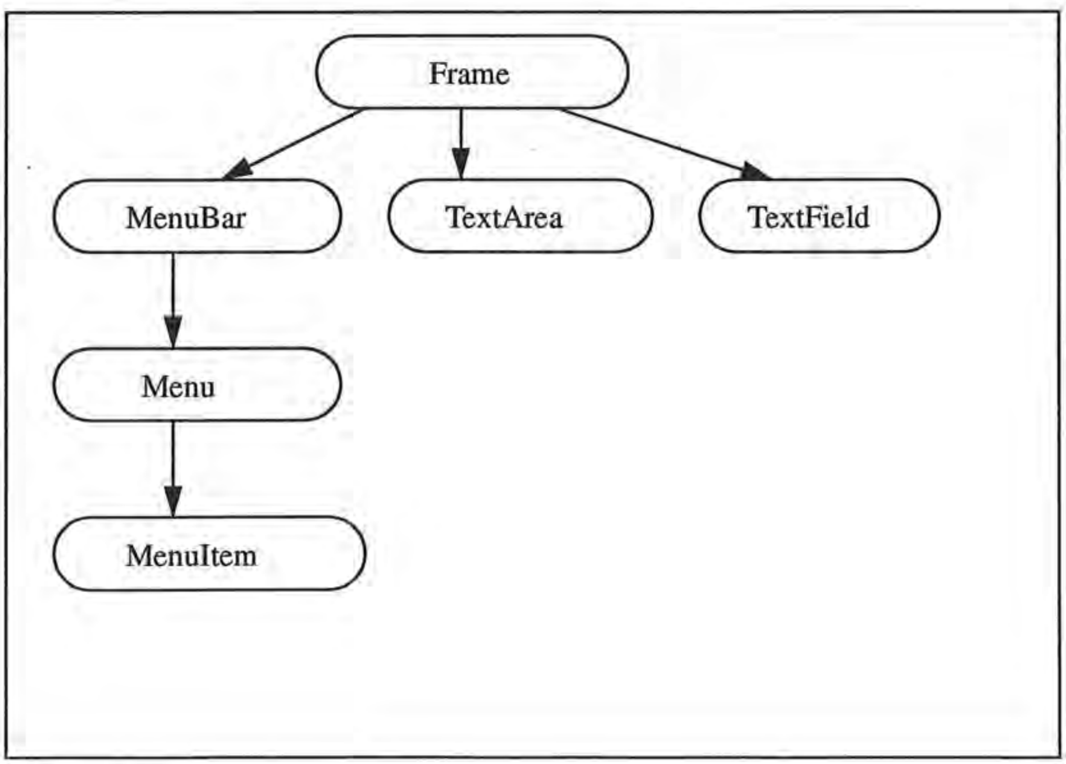
As noted earlier, the interface to Little Smalltalk is basically a read/eval/print loop.To form this, we need capabilities to read text from and write text to a window. Since the system should be written as an applet, all I/O must be done thru the AWT components. Among the AWT components, the only components that are useful for text-based interaction are **TextArea** and **TextField**. Theoretically speaking, we could use just a TextArea to get our work done. But taking care of both reading and writing to the same TextArea would be difficult to implement. So, I chose to take all the input from a TextField, and use TextArea for displaying all output. The user basically types-in text in the TextField and when he presses "return", the text entered in the TextField becomes available for the interpreter. The interpreter reads in the text character-by-character, does the required work and prints the output in the TextArea.

The user-interface also consists of a **MenuBar**. The MenuBar consists of two **Menus** - File and Help. The only MenuItem that is inside the File Menu is Exit. The user selects Exit to exit from the Little Smalltalk system. The Help menu consists of two MenuItems - The "How to" help and the manual.

All the AWT components described above are contained in a **Frame**. The frame pops up when the user enters the web page and presses the "Start Little Smalltalk" button on the page. The frame gets destroyed either when the user selects "Exit" from the File menu or when the user leaves the page.

The hierarchy of various GUI components used is given in fig 1.

FIGURE 1.



A snap shot of the Little Smalltalk system user interface is shown in fig. 2.

FIGURE 2.

As seen in fig. 2., the user-interface contains a TextArea (on the top) and TextField (on the bottom). The menubar contains a File menu and a Help menu. As mentioned previously, the File menu has only one MenuItem (*Exit*) associated with it.

4.2.1 A drawback of version 5

In Little Smalltalk, some messages like *editMethod* require editing capabilities. In the previous versions, the editing was easy - a standard editor like **vi** was invoked and the reading and writing of the file to be edited was taken care of by the editor.

In the current version, editing is complicated by the fact that all I/O in this version is done by reading and writing from a **URLConnection**. Reading from URLConnection is easy. A *getInputStream*() on URLConnection opens the stream for reading. So, to read a file tmp.txt from the server, we need the following sequence of code...

```
URL u = new URL("http://www.engr.orst.edu/~pandraph/tmp.txt");

URLConnection uc = u.openConnection();

BufferedInputStream bi = new BufferedInputStream(uc.getInputStream());
```

Writing to a URLConnection is complicated by the fact that we need some CGI program that takes the output from the output stream of the URLConnection and then actually writes the data onto the file on the server.

This drawback arises from the fact that we cannot invoke local applications from a Java applet without giving-up the multiplatform use of the applet. Since one of the main aims of this project was to make Little Smalltalk available on the web, it is not possible to give-up the multiplatform use of the applet.

The implication of the lack of editing is that the user cannot modify the methods in the class library once the system image is read from the server. The user cannot modify the methods that he has wrtten once.

The problem could be averted with the use of more complicated mechanisms like using sockets to do the I/O. This method would need a server to be running on the server-side which takes the output from the editor and then actually rewrites the edited file with the new contents.

4.3 Interpreter

Little Smalltalk interpreter represents programs internally in an intermediate representation known as *bytecode* format. There are several reasons for using an intermediate representation -

- Compactness - Intermediate representation is much smaller than character representation.
- Efficiency - Once we have the class representation in the intermediate format, we can directly use the internal format without having to reparse the class description each time a method is invoked.

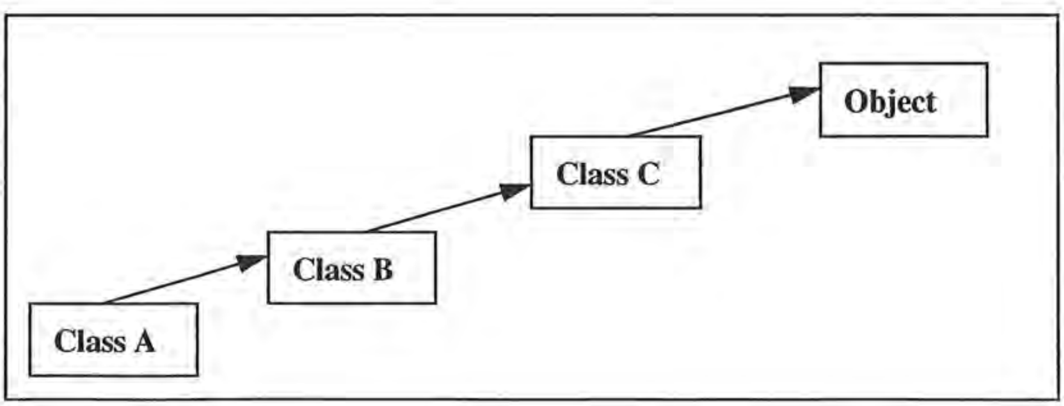
Bytecodes will be described in detail in section 4.3.2.

Now that we know what bytecodes are, we can walk-thru the internal working and representations of the Little Smalltalk (version 5) system.

4.3.1 Objects

A fundamental problem in the representation of objects in Little Smalltalk is developing a scheme that permits representation of a method of a class independent of superclasses. The problem is illustrated by considering the class hierarchy in fig.3.

FIGURE 3.



In Little Smalltalk, all classes are subclasses of class **Object**. Now, in fig.3, class **A** is a subclass of class **B** and class **B** is a subclass of class **C**, which in turn is a subclass of **Object**. Suppose we have an object **objA** of class **A**. If we pass a method ***calculate*** to the object **objA** and if the **calculate** method is inherited by class **A** from class **B**, when the method is actually executed, the only variables available to calculate in class **B** will be the instance variables of class **B** and not of class **A** or class **C**. So, how do we make sure that the representation of methods in a class are independent of the class description of superclasses? The Little Smalltalk solution to the problem is to have a reference to an unnamed object of the superclass as a part of the object representation. So, when an object is created, we not only have the instance variables corresponding to the class of the object available but at the same time, have access to the methods inherited from the superclasses.

As mentioned above, an object may contain instance variables but only variables appropriate to the class of the object. But how many instance variables? This question leads us to another important member of our object representation-the number of instance variables. Another important consideration should be the way in which we are going to store the instance variables. In the previous versions of Little Smalltalk, the object representation just had a pointer which points to the first instance variable and basically the instance variables are represented as a linked list. But in the current version of Little Smalltalk, the instance variables are stored in a Vector. The dynamic growth property of vectors in Java makes the choice particularly attractive.

So, each of the objects contain the following fields :

- Size - The number of instance variables in the object
- Class - This represents the class to which the object belongs to.
- Instance variables - The values of the instance variables.

There are three types of objects in Little Smalltalk.

- LSTgenObject

  This represents the general object type consisting of size, class and the values of instance variables.

• LSTbyteObject

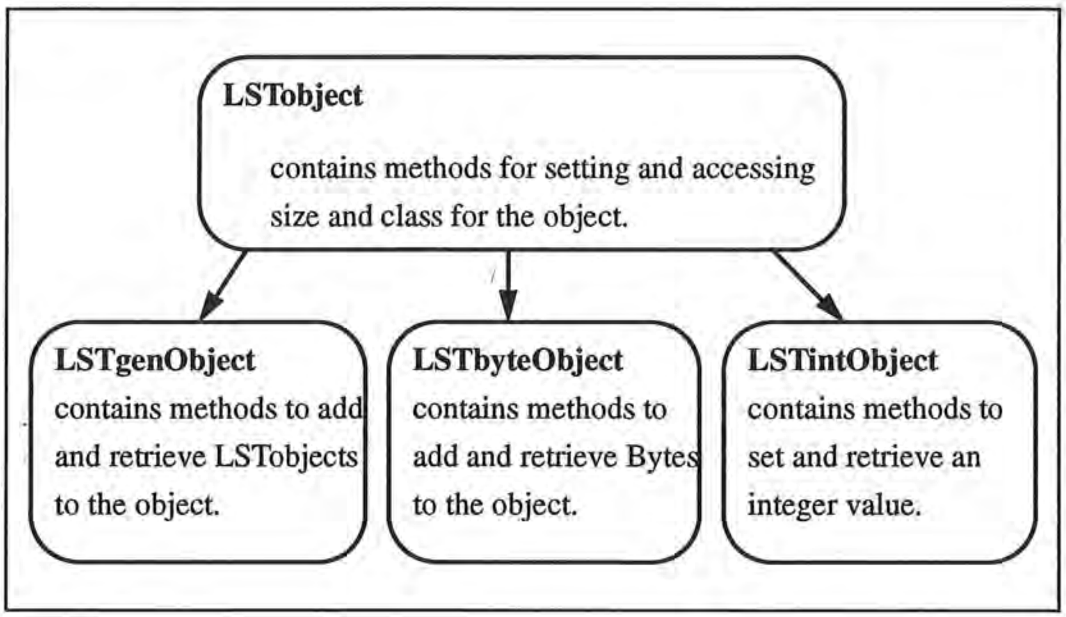
These objects are used to represent strings and symbols.

• LSTintObject

These are integer objects that are used to represent small integers.

The Little Smalltalk interpreter has the following class hierarchy for the representation of objects.

FIGURE 4.



The **LSTbyteObjects** can hold values of type **Byte**. The **Byte** class is basically a type wrapper for **short** values.

The **LSTgenObjects** can hold values of any LSTobject - LSTgenObject, LSTbyteObject or LSTintObject.

The **LSTintObjects**, as mentioned previously hold integer values.

4.3.2 Bytecodes

The Little Smalltalk system represents the class descriptions internally in an intermediate representation called *bytecodes*. This internal representation is advantageous over the character representation because of its compactness. The internal representation , once built, can be used any number of times without having to reparse the class description.

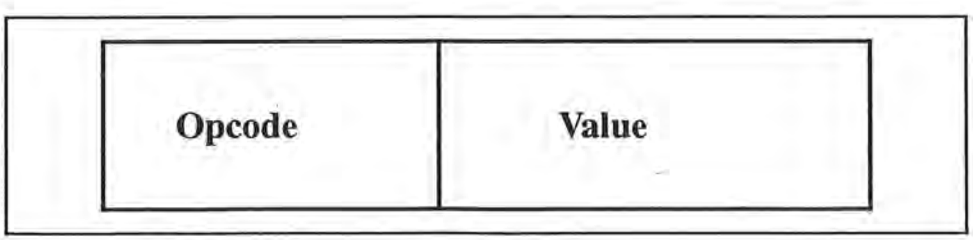
The Little Smalltalk system has a stack-based virtual machine. The virtual machine basically performs actions like :

- Accessing the instance variables.
- Modifying the instance variables.
- Accessing arguments to a method call.
- Accessing temporary variables.
- Modifying temporary variables.
- Accessing literals.
- Sending messages etc.

All these operations should be represented in such a way that the virtual machine has access to the *opcode* (the operation) and the values used by the opertation.

The current version of Little Smalltalk system follows the same opcode format as the previous versions of Little Smalltalk. The opcode-value pairs are represented as shown in fig.5.

FIGURE 5.



The opcode values and the description of what they represent is given in table.1.

TABLE 1.

Opcode	Description
0	Extended : Implies that the value field contains an opcode and the entire next byte is a value.
1	Push Instance : As the name implies, push the instance variable onto the stack of current context.
2	Push Argument : Push an argument onto the stack of the current context.
3	Push Temporary : Push the temporary onto the stack of the current context.
4	Push Literal : Push the given literal onto the stack of the current context.
5	Push Constant : Push the given constant onto the stack of the current context.
6	Assign Instance : The specified instance variable gets a value.
7	Assign Temporary : The given temporary variable is assigned a value
8	Mark Arguments : Load the argument array
9	Send Message : Send a message to a receiver, which is already pushed onto the stack. The necessary argeuments are also on the stack.
10	Send Uncary : Send a special unary message based on the lower order byte. Basically handles isNil and notNil. Used for optimizing unary messages.
11	Send Binary : Used for optimizing certain binary messages. Handles <. <= and +.
12	Push Block : Create a block object and pushes onto stack.
13	Do Primitive : Performs a primitive operation like block invocation, new object allocation, reading a char from input, multiplication, division etc.
14	Not Used.
15	Do Special : Special operations like stack pop, block return etc are handled here.

4.3.3 Structure of the interpreter

The Little Smalltalk interpreter is basically a read/eval/print loop. The structure of the interpreter is given below as pseudo-code :

```
while(system_not_exited) {
        high = nextByte();
        low = high & 0x0F;        // The lower nibble
        high >>= 4;               // Shift the higher nibble
        if (high==0) {            // Extended operation
              high = low;
              low = nextByte(); // actual operand
        }
        switch(high) {
              case PushInstance : ...
              ...
              case PushArgument: ...
              ...
        }
}
```

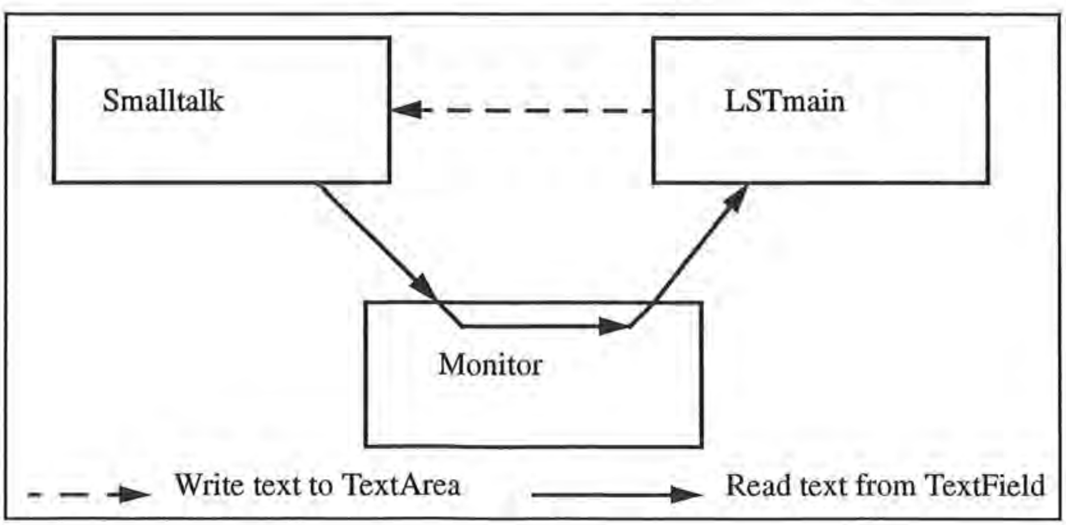
As seen above, the heart of the interpreter is the `switch` statement which executes the code sequence corresponding to the bytecode that is read. The structure of the interpreter closely resembles the structure of the previous versions of Little Smalltalk interpreter.

5.0 Implementation Overview

The following are the main classes that were used in the implementation of current version of Little Smalltalk :

- **Smalltalk** : The Applet. This class contains methods to initialize the user interface and start the interpreter thread. Also contains methods to add text to the TextArea and get text from TextField.
- **Monitor** : Monitor is the class responsible for setting-up the link between the user-interface and the interpreter. Reading text from TextField is not as easy as it might seem because the input is non-blocking for a TextField and the interpreter requires a blocking input. So a monitor, consisting of two methods - get() and put() was used to set-up the link between the user-interface and the interpreter. The user-interface does a put() on the monitor and the interpreter does a get() to read data. Fig 6. shows how the interpreter reads data from the TextField.
- **LSTmain** : The Interpreter. This contains the code for the interpreter. As shown in fig.6., writing data to TextArea does not require a monitor.
- **LSTobject, LSTgenObject, LSTbyteObject** and **LSTintObject** are already discussed previously. The are used to represent the various types of objects handled by the Little Smalltalk system.

FIGURE 6.



The Little Smalltalk interpreter was initially implemented as a stand-alone application. The stand-alone application version of Little Smalltalk reads input with System.in.read() which is blocking. So it does not require the monitor class. The user interface for the stand-alone version is text-based. It does not use any java.awt classes.

6.0 Porting to Java : Problems and Solutions

6.1 User Interface

6.1.1 TextField

The older versions of Little Smalltalk take input from the terminal and the interpreter expects that the input is blocking input. But in the current version, the input is accepeted from a `TextField` (java.util.TextField) and the `getText()` method in the TextField is non-blocking. The problem therefore was to somehow simulate blocking input while still using the TextField for input.

The problem was solved by setting-up a monitor and sending the input to the interpreter by a `get()` method on the `monitor` object and accepting input from the user from the TextField using a `put()` method on the monitor object. The simple task of accepting input has now become a producer-consumer problem.

6.2 Interpreter

6.2.1 The method clone()

Netscape Navigator, a popular web browser, does not run any applet that calls `Object.clone()`. Clone() is a method provided in java.lang.Object which is used to create a new object of the same class as the object with which it is invoked. So, we are now forced not to use clone() in the applet. This means that we have to get around the problem by copying each element of the object on which we intended to use clone() into a new object.

6.2.2 Lack of pointers in Java

The previous versions of Little Smalltalk made extensive use of pointers and the relationship between pointers and arrays. For example, code resembling the following snippet is used extensively in the previous versions.

```
object *data;

....

someVal = data[the_index];

....
```

Because of lack of pointers in Java, the current version is faced with the problem of deciding on a data structure which not allows indexed access but also allows dynamic resizing. Vectors provide both these features. The current version therefore makes extensive use of vectors wherever these features are required.

6.2.3 Slowness of the interpreter

The Java version of Little Smalltalk is very slow. The slowness of the interpreter can be attributed to several reasons:

- Reading-in the image file.

  The snap-shot image file is read byte-by-byte. This takes up a lot of time. One way to get around the problem is to read object-by-object instead of byte-by-byte. This is possible because of the availability of **Object Serialization** mechanism in Java. Object Serialization extends the Java input/output classes with support for objects. But the current version of Little Smalltalk does not use Object Serialization. [See sec 7.2]

  The current version reads-in the image file in the following way :

  1. Reads-in the type of the object (LSTbyteObject, LSTgenObject or LSTintObject)

  2. Depending on the type of the object, keeps reading in byte-by-byte until the object data is filled.

With Object Serialization, the above steps could be replaced by:

1. Read the type of the object

2. Read in the object in its entirety. [One step instead of byte-by-byte .]

- Automatic garbage collection.

  Java provides automatic garbage collection. This means that the Java runtime system does the memory management for the programmer by freeing him from the tasks of manually tracking the allocation and deallocation of memory. There is an asynchronous thread that is always running in the Java runtime which takes care of the garbage collection.

  However, the problem with automatic garbage collection is that it takes up some time in doing its work. Little Smalltalk system creates lots of objects in the execution and obviously the time taken by the garbage collector is also proportionate to the number of objects created.

  The following table shows the total time taken by the garbage collector in comparision with the total time taken to execute the command "File class".

TABLE 2.

Run #	**GC**	**Total**	**Percent**
1.	1770	7525	23.52
2.	1773	7934	22.34
3.	1385	7085	19.54
4.	1750	7916	22.10
5.	1678	7483	22.42

From the above table, we see that about 20 to 25% of the time is spent by the interpreter in garbage collection alone.

- API Classes

  API classes like **Vector** do more than what is needed in a typical application with a corresponding increase in the execution time. The Little Smalltalk system uses Vectors a lot and the performance of Vector plays a major role in the performance of the interpreter on the whole.

When the usage of class java.util.Vector is replaced by a class called **myVector**, which has just the methods needed by the application viz. elementAt(), insertElementAt() and addElement(), it was observed that the execution time has considerably improved.

When this was further investigated, it was found that the difference in the performance of myVector and java.util.Vector was mainly due to the lack of the synchronized access to member functions in myVector.

The APIs written in native code are fast. But API classes like java.util.Vector which are written for very high level reuse, in Java, make no attempt to be fast.

The following table shows the results of an experiment to compare the performance of java.util.Vector and myVector. Thousand elements are added and accessed using addElement() and elementAt() to obtain the results. addElement() and elementAt() are frequently used in the interpreter.

TABLE 3.

Run #	java.util.Vector	myVector	Improvement %
1.	44	23	48
2.	44	20	55
3.	43	20	53
4.	46	21	54
5.	57	32	44

From the above table we see that our hypothesis about java.util.Vector is indeed correct. We see that there is a speed-up of about 50% by using myVector in the place of java.util.Vector.

Another java.util class that is used extensively is **Stack**. Since myVector is more efficent than Vector for this application, it makes sense to add the stack operation push(), pop(), peek() and peekAt() to myVector and use it as a stack also. Note that peekAt() is not supported in java.util.Stack. Since we are using our own implmentation of stack, it is easy to write the methods that suit our needs. With java.util.Stack, peekAt() was simulated by

someReturnedValue = ((Vector) stackObject).elementAt(someIndex);

because java.util.Stack subclasses from java.util.Vector. As seen above, a run-time cast is needed here.

Since push() and pop() are the most frequently used Stack operations, an experiment involving push()ing and pop()ing 1000 elements was conducted to compare the efficiency of java.util.Stack and myVector used as stack. The following table presents the results of the experiment.

TABLE 4.

Run #	**java.util.Stack**	**myVector**	**Improvement %**
1.	66	27	59.09
2.	65	27	58.46
3.	68	25	63.23
4.	74	24	67.56
5.	67	25	62.68

- "Register" Variables

  Java does not provide C style "register" variables.

  In the Little Smalltalk system, the **context** associated with a process is accessed frequently. In the older C versions of the interpreter, the context was declared as a register variable. But in the current version, we do not have that facility.

- All references of objects are through memory pointers. These pointers must be traversed for each access.

- Object-Orientedness

  Object-Oriented methodology can inhibit the performance of Java code. Polymorphic variables/methods require run-time type resolutions. Reusable code (say, the Java API) provide generic interfaces most of which are often not used by the application.

- More Instructions

  Java executes more instructions. Why? There are several reasons. Some of them are listed below :

  1. Array bounds checking

Java does extensive array bounds checking to ensure that a reference to an element out of the array bounds is always caught. While this type of checking is definitely advantageous in avoiding errors, it does consume a lot of time because of the checking done for each reference. To prove this point, an experiment was conducted to compare the execution times of using the container class myVector (described previously) with and without array bounds checking. This experiment is significant because myVector is used extensively in the implementation and the performance of myVector plays a vital role in the overall performance of the interpreter. Any improvement in the performance of myVector can increase the speed of the interpreter significantly. The following table shows the results of the experiment.

TABLE 5.

Run #	**myVector**	**myVector2**
1.	22	20
2.	23	21
3.	26	20
4.	25	22
5.	23	22

While the values in the above table do not carry any significance the difference in the times taken for execution of the test program using myVector (*with* array bounds check) and myVector2 (*without* array bounds check) shows that array-bounds checking does indeed consume a significant time.

2. Security considerations

A user running a Java applet in his browser should feel secure running it. To achieve this, Java designers have imposed certain restrictions on the capabilities of applets. Each applet viewer (browser) has a SecurityManager object associated with it that checks for applet security violations. This checking is done by some "extra" code. As seen above, "extra" code means "extra" time.

While time consumed due to security considerations is an issue for the performance considerations of any applet, it is even more important for applets which demand high performance like Little Smalltalk intepreter.

3. Type checking

Dynamic binding demands extensive type checking to be done by the Java run-time. Again, this task requires extra instructions to be executed (for determining the type) and as said previously executing these extra instructions consumes extra time.

While dynamic binding is a desirable feature for this project and for writing many object-oriented systems, it does involve a performance overhead.

Let us briefly go thru why dynamic binding is a performance issue for this project.

As said previously, there are three types of objects in the Little Smalltalk interpreter. LSTgenObject, LSTbyteObject and LSTintObject - all subclasses of LSTobject.

When a method is invoked on an object of class LSTobject, the runtime should know the runtime class of the value contained in that object and invoke the method in the corresponding class. This type of type-checking is always used when virtual functions are used.

- File Operations

  Java does not allow applets to connect to any host other than the host it came from. So, all the files that are needed for the Little Smalltalk system should be kept on the server. All file operations need to fetch/write data from the server. This is extremely slow for two reasons :

  1. Network delays : Connecting to the server by opening a `URLConnection` to it and fetching/writing data will experience the network delays inevitably.

  2. Data transfer : All data transfer is done byte-by-byte. This makes the file I/O extremely slow in Java. In Little Smalltalk, when a user wants to fileIn or fileOut a file, the file is going to be read/written byte-by-byte.

  This problem can be avoided to a certain extent by using `BufferedDataInputStream` and `BufferedDataOutputStream` for reading and writing respectively. Reading and writing to buffered streams does not necessarily cause a call to the underlying system for each byte read/written. The data is read block by block and kept in a buffer.

- Run-time Casts

Why are run-time casts required at all? Java's static type checking combined with the generic classes provided by the Java API will require the use of run-time casts in some situations. The problem is illustrated below :

In the Little Smalltalk interpreter, we are using a vector to store instance variables and temporaries. Internally, a vector has an array of class Object to hold the values that will be stored in the vector. So, an elementAt() on a vector will return an object of class Object. Now, when we try to compile code with the following sequence of declaration and assignment, we get a syntax error because of Java's static type checking.

```
LSTgenObject someObject = new LSTgenObject();
myVector vec = new myVector();
....                    // somewhere here we put an LSTgenObject at location 10
someObject = vec.elementAt(10);
```

What is needed here is a cast from the type of object returned by elementAt() to LSTgenObject. So, we need to replace the last statement with

```
someObject = (LSTgenObject) vec.elementAt(10);
```

Now the question arises - how to avoid run-time casts?

Since we are using our own version of vector (called myVector) for this project, why not make the array in myVector to be of type LSTgenObject instead of Object? Well, this solves the problem only for LSTgenObjects. But a vector can contain LSTobject, LSTgenObject, LSTbyteObject or LSTintObject in it. This means that making the array contained within myVector to LSTgenObject/LSTbyteObject/LSTintObject/LSTobject does not solve the problem.

- Compiler

One of the major reasons for the slowness of the current version is the lack of an optimizing compiler for this version. The javac compiler provides an optimization option (javac -O) which basically optimizes by inlining static, final and private methods. This is the only optimization done by the javac compiler. The previous versions of the Little

Smalltalk system had the advantage of having optimizing compilers because of the availability of different optimizations (code motion, strength reduction, common sub-expression elimination etc.) in C compilers.

7.0 Comparing the C version with Java version

In this section, we will compare the previous versions of Little Smalltalk with the current version.

- Implementation

  Implementation of Little Smalltalk in C was cumbersome because of the following reasons :

  1. Memory management - Done by the programmer.

  2. Pointers - Managing pointers is a cumbersome and risky issue. So, a lot of care is needed when dealing with pointers.

  Implementation in Java was relatively easy because of the following reasons :

  1. No memory management - Automatic garbage collection.

  2. Using container classes eliminated most of the need for pointers and since pointers were anyway unavailable in Java, the risk involved in dealing with pointers is not there.

- Performance

  The previous versions are much faster than the current implementation. The following table shows a comparision of the execution speeds for some Little Smalltalk statements

  .

TABLE 6.

Statement	Version 4 (C)	Version 5 (Java)	Slow-down
File class	237	7466	31
2+3	195	5930	30
Object subclass:#foo	389	10042	26
Object listMethods	295	8693	29
File viewMethod:#fileIn	1182	34687	29

From table 6., we see that the Java version of Little Smalltalk has about 25 to 30% slow-down compared to the C version.

We have to keep the following points in mind when we compare the two execution speeds :

1. We are comparing compiled code (C) with interpreted code (Java).

2. To have a unbiased comparision of the execution speeds, we need a compiled version of the Little Smalltalk interpreter in Java. This is possible with what are known as "Just-In-Time" compilers, which convert the java byte code to machine code before running the code. There are several JITs available now mainly for Windows 95/NT platforms. JITs promise a 10-30 times increase in the speed of Java programs. This means that the current version is not actually as slow as it appears to be - because with a JIT, it could reach the speed of the C version.

The performance of the Java version therefore not as big a disadvantage as it appears to be because with more and more JITs for various platforms and better Java compilers in future, the Java version will be as good as the C version of Little Smalltalk.

8.0 Using the Little Smalltalk (ver 5) System

Using Little Smalltalk system (ver 5) is very much like using the previous versions of Little Smalltalk. The input is typed in the text field of the user interface. Another difference is that the previous versions used a standard editor available on the system for editing purposes. We have seen previously (sec. 4.2.1) that the editing option is not available in this version. The text area portion of the user interface is only for display and no editing is allowed in the text area.

The user starts-off by opening the URL for the Little Smalltalk page. The interpreter window pops up and the system is ready.

The user will get best performance from the interpreter if the user's browser supports a JIT. Netscape Navigator has JITs included in the Windows 95 and Windows NT versions. JITs for Solaris versions of Netscape Navigator are expected.

9.0 Future Work

9.1 What can be done?

- Object Serialization

  Object serialization can be used to read object-by-object instead of reading the data byte-by-byte. The object serialization package is a relatively new feature provided by Java and doesn't come with JDK. It can be downloaded separately and installed.

- Just-In-Time Compilers

  Just-in-time compilers translate the Java byte codes into machine code which can make many Java programs execute considerably faster. Just-in-time compilers are currently available from some of the vendors of JDK.

- Support the class browser

  Coming versions of Little Smalltalk can make attempts to support a class browser which is currently not available.

- Bytecode format

  Coming versions of Little Smalltalk can make attempts to change and experiment with new bytecode formats to increase the performance of the system.

9.2 Why not now?

- Object Serialization

  Object serialization is not currently available on all platforms. It is not available on Mac OS versions of JDK.

- Just In Time Compilers

  Just-in-time compilers are also not available for all platforms. The user's browser, if it contains a JIT compiler, will improve the speed of the interpreter. Not many vendors have come-up with JITs for the Unix versions of JDK.

- Class Browser and New bytecode format

The current project was basically aimed at porting Little Smalltalk to Java as-it-is and observe the difficulties in the process. Modifying the Little Smalltalk system wasn't one of the objectives.

10.0 Conclusions

The development of interpretive systems like Little Smalltalk in Java have the following advantages :

- The programmer is relieved of the memory management tasks. Explicit garbage collection is not required.
- Java provides several features that are easy to use and make system development easy. For example, there are many classes available that are specific to graphical user interface design. Several classes are available for networking needs.
- Object-Oriented features of Java are helpful in design process.

Although Java is very helpful as far as designing systems like this is concerned, it has several disadvantages that are specific to interpretive systems :

- The slowness of Java is a major drawback in developing interpretive systems. JITs are the main hope right now.
- Garbage collection, although helpful as far as development of the system is concerned, makes the system slow because of the time it takes in freeing objects and managing memory.
- Lack of features like register variables make development of interpretive systems difficult.
- Lack of an optimizing compiler is another major drawback. As said previously, the `javac` compiler does not do many useful optimizations.

All in all, it can be concluded from this project that although Java is extremely helpful in the design and development of interpretive systems, it loses much of its charm when it comes to performace of the system. Since Java is a new language, we can expect that some of these drawbacks will be eliminated as the language and implementation evolve.

11.0 References

1. A Little Smalltalk. Timothy Budd. Addison -Wesley Publishing Company.
2. The Java Tutorial. Object-Oriented Programming for the Internet.

   http://www.javasoft.com:80/books/Series/Tutorial/index.html
3. Java API Documentation. James Gosling, Frank Yellin, The Java Team

   http://www.javasoft.com:80/products/JDK/1.0.2/api/
4. Object Serialization

   http://chatsubo.javasoft.com/current/serial/index.html
5. Introduction to Object-Oriented Programming. Timothy Budd. Addison -Wesley Publishing Company.
6. Java as an Intermediate Language. Jonathan C. Hardwick and Jay Sipelstein.

   http://www.cs.cmu.edu/~scandal/html-papers/javanesl/
7. Advanced Java- Idioms, Styles, Programming Tips and Pitfalls. Chris Laffra. Tutorial Notes, OOPSLA 1996.
8. How Do I...

   http://www.digitalfocus.com/digitalfocus/faq/howdoi.html
9. Not Using Garbage Collection, Chuck McManis, Java World September 1996.

   http://www.javaworld.com/javaworld/jw-09-1996/jw-09-indepth.html
10. Java Unleashed. Sams Net.
11. Performance Optimization

    http://g.oswego.edu/dl/oosdw3/ch25/ch25.html
12. The comp.lang.java FAQ List

    http://sunsite.unc.edu/javafaq/javafaq.html

13. Performance Java, Paul Tyma, preEmptive Solutions Inc.

http://www.preemptive.com/lectures/Optimization.html

14. Performance testing C++ code, Neil Hunt, Pure Software

http://www.pure.com/quality/performance.html

15. Java Newsletter Back Issues, Glen McCluskey & Associates

http://rmi.net/~glenm/backjava.html