The Omegahat Environment: New Possibilities for Statistical Computing

Duncan Temple Lang
Bell Labs
Murray Hill, New Jersey

July 28, 2000

Abstract

Statistical environments such as *S*, *R*, *Xlisp-Stat* and others have had a dramatic effect on the way *we*, statistics practitioners, think about data and statistical methodology. However, the possibilities and challenges introduced by recent technological developments and the general ways we use computing conflict with the computational model of these systems. We explore some of these challenges and identify the need to support easy integration of functionality from other domains, and to export statistical methodology to other audiences and applications, both statically and dynamically. Existing systems can be improved in these domains with some already implemented extensions (see Section 5). However, the development of a new environment and computational model that exploits modern tools designed to handle many general aspects of these challenges appears more promising as a long term approach. We present the architecture for such a new model named Omegahat. It lends itself to entirely new statistical computing paradigms. It is highly extensible at both the user and programmer level, and also encourages the development of new environments for different user groups. The Omegahat interactive language offers a continuity between the different programming tasks and levels via *optional type checking* and seamless access between the interpreted user language and the implementation language, Java[1]. Parallel and distributed computing, network and database access, interactive graphics, and many other aspects of statistical computing are directly accessible to the user as a consequence of this seamless access. We describe the benefits of using Java as the implementation language for the environment and several innovative features of the user-level language which promise to assist development of software that can be used in many contexts. We also outline how this architecture can be integrated with existing environments such as *R* and *S*.

The ideas are drawn from work within the Omega Project for Statistical Computing[14]. The project provides open-source software for researching and developing next generation statistical computing tools.

Keywords: Java, Reflectance, Optional type specification, Object oriented programming, Omega Project.

1 Introduction

High-level interactive statistical computing environments have had a profound effect on how statistical analysis and research is performed. This is evidenced by the Association of Computing Machinery (ACM) recently awarding John Chambers its Software System award for *S*, citing that system as having "forever altered how people analyze, visualize, and manipulate data." Systems like *S* exploited the best innovations

[1] Java is trademark of Sun.

from computer science, providing environments customized for data analysts and researchers. The recent revolution in hardware performance and prices, Web technology and the spectacular emergence of high quality Open Source software present us with new opportunities and challenges. These include not only the ability to incorporate these new technologies for our own benefits, but also the challenges to make our technology available to other disciplines and broaden the use of statistical methodology. We must change from the monolithic, stand-alone systems we currently use to ready ourselves for the new open style of computing in which each application is just an agent dynamically exchanging data and services with other agents. It is an important time to create an infrastructure for statistical computing in which the exchange of new developments in different domains is both feasible and easy. This is a highly non-trivial task and requires significant research and effort by the statistical community to prepare what is arguably one of its most important tools, statistical computing, for current and future developments in the general computing community.

Section 2 outlines the computational and developmental model on which most of the existing systems are based and lists several of the challenges for that model posed by some applications that are becoming more common. We propose two approaches in response to these challenges. The first is to develop a new environment, and the second is to extend the existing systems with what we term inter-system interfaces to allow dynamic communication amongst themselves and with other applications. In Section 3 we discuss the development of a new system, Omegahat. We outline the benefits of choosing an appropriate implementation language, in our case Java, and illustrate the extensibility provided by an object oriented framework and how Java encourages such a design. We describe some of the more novel aspects of the Omegahat interactive language in Section 4. The direct access it provides to Java classes allows new technology to be immediately integrated into a user's session. The optional type specification of the Omegahat interactive language allows users to smoothly move between the roles of interactive user and application developer and facilitates compilation and exporting of Omegahat functions to other environments. Finally, in Section 5, we outline how the facilities provided by the Omegahat software can be accessed from existing, familiar environments such as *R* and *S*.

This paper is not intended to serve as a tutorial for the Omegahat environment. (See [3] for examples of using Omegahat.) Instead, it discusses the design of a new statistical computing environment and fundamental software engineering practices. While it may initially appear to have practical relevance only to those developing such systems, a broader audience will be able to utilize the information. This is a consequence of the open and extensible nature of the Omegahat environment which allows users direct access to the internals of the system itself. Thus an understanding of the system will aid programming in the system, and possibly provide a better understanding of existing systems.

The Omegahat environment is just one element of the software developed as part of the Omega Project for Statistical Computing. This is an effort to research statistical software ideas for the next generation and broadening the use of statistics in other software. We hope to promote experimentation and innovation not only in the implementation and use of statistical methodology, but also statistical environments and technology transfer. The membership of the project includes a significant number of the developers of existing systems around the world, including the development "teams" of *R* [11], the current *S* [4], Voyager [17] and *Xlisp-Stat* [21].

Many of the general ideas discussed in this paper have been distilled from the initial work on the Omegahat software components which now serve as working examples of this approach. Work on a high-level graphics framework, algorithms for numerical computations that integrate several linear algebra packages, and distributed computing approaches (within Omegahat, *R* and *S*) to different statistical techniques has given us some promising evidence that these approaches represent the best strategy for future work. We should note that the Omegahat tools are not at the moment intended to be used as a statistical system that can be used to replace any of the existing ones. However, we believe that the current Omegahat tools will

be valuable to others who are building new software for statistics. All of the software is available as open-source and we hope this will encourage others to make use of our experiences and to also contribute to the research. We strongly believe that the open-source model will help to improve the quality and generality of the software and the approaches. Given the complexity of the challenges, encouraging others to become involved is vital, especially given the limited research attention statistical computing receives. We strongly encourage people to make use of the existing software and participate in the development process. More information about the project and its activities is available at `www.omegahat.org`.

2 Challenges for Statistical Computing

Most of the existing interactive computing environments are built with similar computation and development models. It is assumed that there is a single user, or specifically that there is a single source of commands. The user's commands are performed sequentially. The data on which the computations are performed can be imported into the system from files or other applications. However, the data is then under the management of this system and the dynamic connection with the original source is removed.

The user's commands are specified via a high-level language developed for that environment and the domain of problems it supports. This language insulates the user from the internal low-level details, usually implemented in another language such as *C*. The user's commands manipulate internal data objects by referencing variables and calling user-level functions that eventually invoke internal routines. While the user language provides a more convenient expression of common tasks, the architecture separates advanced users from the system internals and prohibits direct use of that code. The user interface in this model works very well and we capture this in the Omegahat environment while attempting to minimize some of the other constraints.

Most of these systems are extensible in the sense that one can add functions in the user-level language to provide additional, or replace, existing functionality. Additionally, one can integrate "foreign" routines developed in languages such as *C*/*C++* or Fortran in order to provide new facilities. This allows tools previously developed with no knowledge of the statistical environment to be utilized by writing interfaces between it and the user-level programming language. The extensibility is, however, limited to certain levels of the environment, specifically the user level functions.

From the perspective of the developers of the system, much of the effort is applied to dealing with low-level, non-statistical details. These include facilities such as memory management, persistent storage of data across sessions, dynamic loading of shared libraries, configuration for different systems, and so on.

Nevertheless, this model has proven extremely effective. It allows inexperienced and casual users to work interactively, reading data from a variety of sources and formats, explore the system in their own time frame. More competent users can develop functions to simplify reusing common idioms. When efficiency becomes more pressing, the user can decide whether to use the lower-level foreign language interfaces. And advanced users can create software for others to use.

So have we reached the limit of developments in these types of environments? Clearly not. We can, of course, make the algorithms used by these systems more efficient and allow them to work with larger datasets. We can introduce new state-of-the-art algorithms. However, some of the comments above suggest that these systems may not be convenient for dealing with the demands of the ever-changing computer world. In the following paragraphs, we briefly describe different tasks and paradigms that are not readily supported by the computational and software engineering models of these existing systems. These fall into two basic categories. The first suggests issues that affect the use of statistical computing by statisticians and data analysts. The second relates to how we can make our work more accessible to others. These scenarios are of course selected to highlight the potential difficulties and are not intended to malign these systems that have proved so useful over the past 20 years.

- **Programming Spectrum.** The gaps between interactive commands, programming and software development must be reduced. Productivity in computer work comes most commonly from code reuse, and the languages we use should encourage the development of software that can be reused. However, most programmers do not identify the possibility for reuse in spite of repeated evidence. Somehow, these high-level languages must encourage users to think in these terms by providing idioms that make it easy and natural. This not only covers creating reusable functions, but libraries for others to use, and encouraging documentation as part of the language and environment. The progress made in *S* version 4 [4] with respect to object oriented programming represents a significant contribution towards this end. Ideally, the language would also allow extensibility and reuse of the tools within the system itself.

- **Extensibility.** The existing systems support a high-level of customization of computations done within the *user-level language*. It is difficult however to customize and adapt the internal algorithms due to the dependencies on low-level data structures and memory management issues particular to the system. For example, one cannot easily substitute optimization procedures with routines from other libraries, or use sparse matrices with the linear algebra system employed by most of these applications. Future systems should allow users to *easily* override aspects of some of the computations to take advantage of contextual knowledge. An object-oriented system provides a natural framework for such "specialization". Even with this, the divide between the internal implementation and user-level languages prohibits easy integration and needs to be reduced to make such customizability possible.

- **Massive Datasets.** Large datasets are becoming increasingly common. The demands these place on the computational environment mean we must look for a different paradigm for computing which supports both large and small datasets seamlessly. Parallel (multiple processors) and distributed computing (multiple machines) are a natural way to deal with larger volumes of data, and more rapidly. The user level language should ease the development of these traditionally hard-to-program algorithms. Additionally, we must be able to handle algorithms that deal with streams of data, dynamically made available to the system.

  A separate issue, though related, is the need to deal with data from many sources. Database management systems (DBMS) such as Oracle, MySQL, etc. are used pervasively. However, statistical computing environments typically have limited support for accessing these systems. Many allow simple queries to read data from the DBMS. Few allow one to write values back to the database, and even fewer permit more advanced interaction and connectivity such as meta-data programming, event/trigger notification, and so on at the user level, if at all. As with many aspects of statistical computing, future systems must support a model in which they are just one component in a larger network of computing agents.

- **Methodology Availability.** A common barrier to more widespread use of modern statistical methodology is the need for non-statisticians to learn one or more new languages and environments so that they can access that functionality. Better ways of communicating the methodology to allow the user to work in a more familiar environment are important. Some of the possible solutions to this problem are reasonably common, but require additional facilities from the existing systems we use.

  – We should be able to convert user-level functions into stand-alone libraries that can be used in other, more common programming environments and languages such as *C*, Java, etc.

  – Alternatively, we can use distributed or inter-process computing to dynamically provide the functionality from the statistical language to different client applications. For example, we may want to be able to access functionality within other applications such as visualization software

(e.g. XGobi) from our environment, and also provide statistical techniques (e.g. smoothing) to it for use in its displays. This is important so that we can benefit from existing tools developed in different languages, running on different machines and also reciprocate these gains. Support for CORBA in *R* and *S*, and COM in *Xlisp-Stat* (see [22] in this issue) makes this feasible. These approaches do, however, require a non-trivial investment in a distributed configuration and raise issues such as network resource management and security.

  - We might persuade others to use the same statistical applications as used to develop the algorithms if we can provide customized graphical user interfaces (GUIs) which hide the underlying environment. These systems must provide flexible and integrated facilities with which its users can develop GUIs. This approach is significantly more general than the developers of the system providing a single graphical interface, and amounts to providing meta-tools for the system users. The argument for augmenting the programming environment with graphical interfaces is outlined in [5] and is an important characteristic of such future systems.

- **Internet.** The explosion of the Internet has prompted many organizations to provide dynamic Web-based reporting services to their employees, members and customers. Many of these systems involve the integration of different database management systems and statistical software. As this approach becomes more common, the need to support multiple users, with remote, persistent and potentially collaborative sessions will grow. [1] describes one example how an existing systems have been used in this manner. Importantly, the computational model of these systems is at odds with this use. For example, they provide neither multi-user capabilities nor a security model.

- **Efficiency.** Most of us are familiar with the decision to implement an algorithm in a high-level, easy-to-use interactive language or the more efficient, but complex, approach of using a compiled language such as *C*. Most of the high-level scripting languages have led to a large separation between interpreted and compiled code. As the volume of data increases and efficiency becomes an issue, we need a smoother way to transition from these high-level languages to compiled code. The system must actively support this. The ability to augment user functions with information so that it can be compiled would allow users to program in a single language while benefitting from the advantages of another. The dynamic nature of most of these languages makes it impossible to compile generally and faithfully. The need for compilation for efficiency reasons is related to the need to export libraries to other systems mentioned above.

- **Integration.** We need to be able to rapidly and seamlessly deploy 3rd party libraries written for other systems (e.g. in *C*/*C++*, Fortran, Java, etc.) to take advantage of the most recent technology as it becomes available. While many of the existing systems facilitate the integration of "foreign" libraries into the system, it is far from simple. Often, there is a tension between the assumptions of the overall system and the new library. Common examples of this are name conflicts, memory management and initialization procedures. Even in the absence of these incompatibilities, developing the user functions that interface to the native routines is a time-consuming task. Debugging across the two languages with dynamically loaded native code also proves complex, especially for inexperienced users. The result is that we do not make as much use of available libraries as we might. We are also hesitant to commit any time to investigate such a library if we are not certain we will use it in the long term. The opportunity costs of such experimentation are excessive. As a result, we lose potential advantages and end up without certain facilities or reinventing them ourselves.

Faced with these potential future limitations to our systems, we have two obvious ways to proceed. We can enhance the existing systems by providing facilities that will support these and other scenarios that arise

in the future. We mention existing work to provide these extensions in Section 5. Alternatively, we can build new systems which incorporate modern needs and technologies into the model from the earliest stage in the development. The next two sections discuss our approach to such an endeavor. While we believe that building a new environment is necessary, we recognize the need to utilize the existing systems and so are pursuing both approaches in parallel.

3 Creating an Environment

When developing an entirely new environment, a good start involves identifying the potential elements of the system. Since the limitations of the existing environments outlined in Section 2 relate to the computational model and internal structure of the system rather than the user facilities, we can benefit from observing what these systems provide. The most important aspect of the system is to provide high-level interfaces to the different tools needed by users. The first of these is an interactive language. Interfaces to other systems such as *C*, *C++*, Fortran, databases, XML (eXtensible Markup Language) and distributed computing via CORBA or Java's Remote Method Invocation (RMI) should also be considered at an early stage and incorporated into the design in a natural way. Statistical modeling, probability and simulation facilities and (interactive) graphics capabilities are other obvious modules to provide. But, these are modules and should not be built directly into the system. Instead, the system should provide access to them as it would any third party library. While this internal architectural difference may seem negligible to the user, the effect is significant. These modules become individually accessible components that can be used in other applications, promoting the exchange of software with other disciplines. Indeed, if the existing systems were designed in this component-based manner, many of the challenges mentioned earlier would disappear. Other than this simple point, we will defer discussion of these modules to future papers and describe the Omegahat language in Section 4 below.

3.1 The Choice of Implementation Language: Java

The next step in the system design is to determine in what language to implement both the computational environment (the user-level language) and the modules that provide the user-level functionality. A prudent selection of the implementation language(s) can simplify the work for the developers and the users. In the past decade and longer, *C* has been a natural choice for the implementation language of a statistical computing environment. One of the reasons for this is the availability of good libraries for linear algebra, optimization, etc. Additionally, other tools such as compilers, parser generators, and so on have been available for a variety of different platforms. Code developed with *C* is considered efficient. And developer familiarity and preferences are important factors when choosing a language.

In spite of its popularity, *C* does not seem to be an ideal language for statisticians to develop algorithms or environments. While it allows one to write code for almost any task, it does not provide much assistance for avoiding common errors. There are several other languages that one might consider using which do provide more direct assistance while still providing adequate control to the developer. These include *C++*, Lisp, Eiffel, Sather, Dylan ([9]), Ada95, Oberon and Java. Rather than comparing these languages, we will discuss some of the aspects that make Java a useful environment, contrasting it with *C*, and illustrate why we have chosen to use it as the primary development language for Omegahat. The ideas that we discuss are not dependent on Java, however.

An important distinction between Java and *C* is that the term "Java" usually refers to three separate components that are commonly employed together. The Java programming language is used to develop applications, applets and libraries (known as packages), which are compiled into Java byte-code. The Java

virtual machine (JVM) runs these applications, interpreting the commands in these byte-codes. The commands refer to fields and methods provided by other packages including the Java Core packages. These packages constitute the third component of the Java environment and are typically implemented with support from the JVM.

While Java byte-code is typically compiled from "programs" written using the Java language, this is not a necessity. One can use different languages and compilers to generate Java byte-code which can make use of the JVM facilities. There are Java compilers for the C, Oberon and Ada95 languages. As we will see below, we can even use Java classes to generate Java byte-code that can be used directly by that application. Hence, we can exploit the benefits of "Java" using programming tools with which we are currently familiar. However, these tools are not as evolved as we might wish and additionally writing C-like code that runs in a JVM may not be the best way to use the Java facilities.

One of the advantages of using Java to implement a statistical computing environment is that many of the facilities that a statistical computing environment must support are provided directly by Java itself. However, the needs of the end user must also be taken into account. This is especially true since the Omegahat system provides greater customization and extensibility at the system implementation level than existing systems. We expect and encourage users to modify the system itself and work closely with it, but of course it is not required. Thus, the features that system developers want from a language must be balanced with those used by users who program at the lower level. Fortunately, these facilities that Java provides help in both system and user-level code development and help to provide a uniform interface for both aspects. The important facilities are described below.

1. **Memory management**. The most common cause of errors in *C* code is memory management which is under the complete control of the programmer. Yet few developers of native algorithms within *S*, *R*, *Xlisp-Stat*, etc. take advantage of this potential. Hence, it is a high price to pay. On the other hand, Java handles all memory management automatically, freeing resources when they are no longer referenced by an application. It provides adequate control for most purposes and the loss of certain facilities (e.g., pointer arithmetic) is more than compensated for by the comparative simplicity and robustness of code. Advanced facilities in Java such as weak references allow sophisticated system management with little additional effort from the developers.

2. **Portability**. Java byte-code can be run, without the need for recompilation, on any operating system and hardware that provides a JVM. Byte-ordering, etc. is taken care of by the language and virtual machine. This greatly simplifies debugging and distribution of user-level libraries, as well as the system itself. The complaints about the non-portability of Java are more accurately described as anomalies in the core libraries (specifically the graphical user interface (GUI) libraries) and not deficiencies of the portability of the Java language or byte-code.

3. **Multi-threading Support**. As described in Section 2, the ability to have multiple tasks executing concurrently is important if we are to make use of multiple processors, handle more than one user session within the environment and simply integrate modules that are essentially independent but run simultaneously. Unlike *C* and *C++*, Java provides support for threads in the JVM *and* in the language. The synchronization mechanisms are remarkably simple, facilitating their use.

4. **Object persistence**. One of the advantages of environments like *S* and *R* is that variables generated within a session can be easily saved to disk for use in future sessions. Much work is done within these systems for writing the contents of user variables to disk and for reading them back into the system. In some systems, one cannot transfer the resulting files between machines because of different low-level representations. Again, Java provides a mechanism for serializing arbitrary objects

onto different types of "streams" and overcomes the issue of portability. While the default serialization mechanism applies to most situations, the design of Java allows developers to provide their own serialization mechanism when needed. An advantageous side-effect of using Java's serialization facilities for inter-session persistence in Omegahat is that other applications can directly read Omegahat objects and vice-versa. This is a result of Omegahat objects being Java objects and stored as such. Other applications read them as Java objects, unaware of their origin. Thus these other applications do not need additional code to understand Omegahat object formats, as is necessary when directly exchanging objects between the existing systems.

5. **Input & Output**. Java provides a high-level and uniform interface to input and output operations. It allows developers to use connections to URLs, sockets, files and pipes in exactly the same way. Facilities for handling compression, secure connections, and so on, can be connected within an object pipe-line. When combined with high-level text manipulation and formatting facilities, the result is a portable, easy-to-use I/O system that frees developers from many tedious details associated with developing an interactive statistical computing environment.

6. **Foreign Routines**. Java provides a relatively simple facility for using foreign – *C*/*C++* or Fortan – code. This allows existing code to be used, perhaps with efficiency advantages. There are some disadvantages in terms of portability and access from Web browsers, but the flexibility may be important in some contexts.

7. **Exceptions**. The ability to generate and catch different types of errors as regular Java objects provides an expressive framework for error handling. Exceptions are used to indicate anomalous events or data values within a Java method. The Java language allows programmers to identify where and how different types of errors should be handled. Anomalies such as division by zero, missing files, and so on, can be treated as fatal errors or processed in context–specific ways and with greater resolution than similar constructs (long jumps) in *C*-like languages.

In addition to these fundamental low-level system facilities that Java provides, the tight integration via the core libraries of numerous packages makes development of a complete system feasible at a much higher level than in *C*. These packages include:

- input and output;
- networking facilities;
- graphical interface components and event model;
- high quality graphics with built-in support for printing and conversion to Postscript;
- database connectivity;
- distributed computing via RMI and CORBA; and
- security.

Other packages are available from Sun and third party developers that provide additional functionality. These include 3D graphics, encryption, a help system, and many more. The simple point is that Java has a significant momentum and we are likely to inherit a great deal of functionality and at a very high-level. We will be able to focus on the use we can make of that technology in our domain, rather than on the technical aspects of integrating it with our system. This is one step in allowing us to foster connections with

other disciplines and their software. It is also a reason that we do not favor selecting other environments such as Dylan or Oberon as base languages, even though they may represent better technology. The lack of significant commercial and community support for these makes them less attractive than Java.

Java is not a panacea that dominates all other languages. There are several disadvantages to using it for building a statistical computing environment. Firstly, the extensive collection of Java libraries does not include good, stable numerical libraries that are readily available for *C*-based systems. For example, the support for key areas such as numerical linear algebra is still evolving: some promising work exists (Colt [10], Jama [12], Jampack [19]) but more research is needed (some of it by us) to provide adequate support and take advantage of Java in these computations.

Secondly, a major concern regarding Java is its performance. Java byte-codes are interpreted by the virtual machine and are in general slower than machine code generated from source written in C and other languages. This is undoubtedly a problem considering we spend much of our computing effort on large numerical computations. Just-in-time (JIT) compilation is an effort to improve the performance of Java. This approach uses fast, native machine code that is created when Java byte-code is first executed. In this way, operations that are repeated frequently can be performed at approximately the same speed as that derived from *C* and Fortran code. Because of the rise in importance of Java, we expect this and improvements to the JVM to get more attention in the future. Additionally, Java provides a more structured interface to *C*/*C++* than most of the existing systems we use. Hence, we can avail of well-tested, fast code by using it to implement Java methods natively. This is relatively transparent to the user. However, we have given up the simplicity, portability and security that we have gained from using pure Java code. On another front, efforts such as GCJ, the GNU compiler for the Java Programming Language [8], to allow Java byte-code to be (re) compiled to machine code allow users to balance the benefits of using pure Java approaches against the need for "efficiency" gained from machine code. And finally, we must understand that short computation time must be balanced against lengthy development (including debugging) time introduced by using low-level languages.

Finally, another considerable disadvantage of using Java is that it is relatively new and few statisticians have taken the time to learn about it. This is true of object-oriented programming in general. As a result, the number of potential users and developers in the immediate future is smaller than if we used a more traditional language or environment. However, this is no reason to avoid using good, modern technology. Also, as we describe in Sections 4.1 and 5, we have provided familiar interfaces to this new environment and its functionality. Additionally, the need for good software engineering approaches is becoming more important. The statistical community must expend more effort in learning the day to day tools and paradigms of our profession.

3.2 The Java Language & Object Oriented Programming

Having explored the immediate advantages of using the Java virtual machine, we now look at the Java language, focusing on how it assists in the development of tools for a statistical computing environment. We focus on three programming techniques that allow developers and users to customize and extend libraries in simple ways. The first is the concept of inheritance or class extension and is discussed in this section. The general paradigm of encapsulation and programming to the interface is discussed in Section 3.3. Finally, the observer or event notification design pattern is outlined in Section 3.4.

Java is a member of the object-oriented family and many of the features that we exploit to create an extensible system are shared by all object oriented languages. An advantage of Java, however, is that it is much simpler than languages such as *C++* and allows one to focus on expressing object oriented ideas rather than complicated syntax and linking. Java is more constraining in form than *C++*. It requires that all routines be part of a class and so there are no "global" methods or variables. This encourages developers to think

in object-oriented terms and also simplifies programming by presenting a uniform structure. Whether the simplicity of Java is a limitation generally, and specifically in our specialized domain, remains to be seen.

The fundamental unit in Java is a *class*. A class is a collection of data fields (variables) and methods (functions) that each instance of the class – an object – has when it is created. An object's methods are tightly coupled with its variables. These methods operate on these variables and those of the objects passed as arguments to the methods. Good programming practice encourages encapsulation so that variables in an object need not be directly accessible to other objects, but indirectly via methods made publicly available by that source object (or more specifically its class definition). In contrast with a language like *C* in which routines can be defined only at the global level, classes immediately provide extra resolution for specifying methods that are to be used by different objects in the system. And the scope of these methods (i.e. the collection of variables visible within the body of the method) naturally includes the object's variables.

The real advantage of classes and the object-oriented approach comes from the concept of inheritance. Suppose that we have a class for representing a matrix. It would have variables for storing the values within the matrix, and might provide methods for inversion, transposition, Cholesky and QR decompositions, and so on. Now consider that a user wants to utilize an alternative implementation for the QR method. We might, for example, use an algorithm that is faster (perhaps using multiple processors), or that has better numerical properties or exploits structural properties of the matrix. In Java, we would *add* this to the system by defining a new class, say `FastMatrix`. We would like for this new class to *inherit* all the properties from the `Matrix` class (e.g. storage of the elements and the arithmetic and inversion methods), and to be able to provide a replacement implementation of just the `QR()` method. We do this by having `FastMatrix` *extend* the original or base class, `Matrix`. Instances of the original `Matrix` class use the original QR method, whereas objects of class `FastMatrix` call the new one.

The fact that extension adds or overrides methods rather than replaces them, and also defines a relationship between the base (`Matrix`) and derived (`FastMatrix`) classes allows the new methods to use the original ones. For example, a multi-threaded version of the QR method might initially check the number of available processors. If there is only one, it might invoke the method it overrides, i.e. the one from the `Matrix` class.

The fundamental benefit of inheritance is the ability to incrementally add to a system. We perturb the system locally and with minimal impact on the existing structure. This reduces the need for re-testing all aspects of the code. Object-oriented languages encourage code reuse offering language constructions for specifying inheritance. Another simple example of how we utilize this is defining different extensions to the database classes that are used to store variables in the interactive Omegahat language described below (see Section 4). The base class simply associates names and values. A simple extension of this class overrides the assignment method so that the value is also written to disk. This derived class is itself further extended so that the serialization to disk uses compression. Essentially, inheritance simplifies a potentially complex system making it easier to understand and customize.

To see the flexibility that inheritance provides, consider how one might introduce the alternative QR method in a *C*-like language. We could define this new method by giving it a different name, say `fastQR`, and substituting *all* references to `QR` with this new name. Alternatively, we might just replace the body of the original version with the new algorithm. This, of course, removes the existing method, forcing all users to employ the new one. Additionally, the new method cannot defer to the original one if it determines that its approach is less valuable (i.e. the call to an inherited method via `super` in the Java example above). Replacing the original method also introduces an instability into the system. Alternatively, more complicated methods using function pointers can be used. However *C* passively tolerates this rather than providing any specific support for it since it is not an object-oriented language. None of these approaches are simple. Each places the burden on the developer while still introducing the instability mentioned above.

In addition to being useful for the developers, the object oriented approach benefits users of the system.

Any user can derive an extended class such as `FastMatrix` and use it directly. In this way they can specialize the computations in a context-specific manner. An example of where this is useful is the optimization mechanism being developed in Omegahat. At each iteration of a loop, the user-specified algorithm is asked to move to another position in the parameter space in an effort to locate an improvement in the objective function. The user will choose from a collection of algorithms provided by Omegahat or some other package that have different techniques for performing this step. Some algorithms may use derivatives to compute the next location, while others might use a nested optimization procedure to look locally, and others might perform some kind of parallel search. Many of these algorithms will inherit code for managing the state of the algorithm across steps and so on from a common base class.

Next suppose that the user knows that the value of the parameter θ in the model of interest should not exceed the value 10 based on a physical understanding of the process that generated the data. While some form of constrained optimization may be available, a simpler approach to the problem utilizes inheritance. Suppose we start with an implementation of a suitable optimization algorithm implemented via the class `BasicAlgorithm`. The user can define a new class that inherits all the methods from this class. Then she redefines the `step()` method (used by each iteration of the optimization loop) by having it call the original version of that method and then checks the value of the parameter of interest. This might be specified using the following code which defines the inheritance relationship between `ConstrainedAlgorithm` and `BasicAlgorithm`, the replacement method for the `step()` and the use of **super** within `step()` to invoke the inherited method.

```
class ConstrainedAlgorithm extends BasicAlgorithm {
   ...

 boolean step(Parameters currentValue) {
   super.step(currentValue);
   if(currentValue.theta > 10) {
    stop("Parameter value exceeds user defined range");
   }
 }
}
```

Using inheritance in this way allows the users to treat the components of the system as a library which they can use directly. They can easily customize computations according to their current needs dictated by the problem and resources. This level of customization of existing systems is only available in systems that provide publicly available source code and to users who are prepared to undertake the necessary *C*-level development needed to introduce such extensions.

An important aspect of this object-oriented style of programming is that both system developers and users can customize a class. Omegahat has been developed using the philosophy that its developers are merely advanced users. We have attempted to capture the development process within the software itself and provide the same facilities to other users. The result is that other users can create their own environments and specialize classes using the base Omegahat packages. This will encourage people to explore enhancements to existing environments and entirely new interfaces by removing the significant obstacles involved in experimenting with system development.

3.3 Java Interfaces

In the optimization example above, we have suggested that the user extend an existing algorithm class so that she can change the `step()` method. If the implementation of the new approach uses little or no code

inherited from the base class, the use of inheritance is actually constraining. Java does not allow a class to inherit from more than one other class. But strongly-typed languages like Java require that the class of the algorithm variable be specified at compile time. (Note that while the resulting byte-code references the step method of the class specified at compile time, the step method of the object used at run time is called, allowing methods supplied by extended classes to be called). So there is a tension between extending one class so that we can inherit its type, and extending another so that we can inherit its code. What is really needed is a way of specifying that a class has a particular collection of methods. After all, in our example, the optimization loop only requires that the user's algorithm provide a `step()` method. To this end, Java provides the concept of an *interface*.

Interfaces act as contracts between an object (and its class) and callers of methods of instances of that class. They specify what an object can do and not how it does it. When a class *implements* an interface, it agrees to provide implementations for all the methods enumerated in that interface. The interface is simply a collection of method declarations (i.e. names, argument types and return type) but with no implementation. A class can implement multiple interfaces. It does not inherit code from any of them since these are simple declarations. The advantage of interfaces is that they are valid classes when declaring the type of an object in a Java program. In the optimization loop example above, rather than using the abstract class approach, we can declare the algorithm to be an interface class, say `StepAlgorithm` which declares the `step()` method. In this way, the body of the loop is written entirely in terms of that interface. Developers are free to uses classes that extend any other class provided they implement the method(s) of the `StepAlgorithm` interface.

A second example of the use of interfaces arises again in connection with the database classes. Omegahat allows "foreign" databases to be used as the source of variables in the Omegahat languages. For example, a table in an Oracle database or a CORBA naming context can be treated as an Omegahat database. These do not need to inherit the hash-table code from the basic classes used to implement the name-value pairs in Omegahat. Instead, all code relating to generic databases need only provide methods for listing the available variables, retrieving a variable, querying whether a variable exists and assigning and deleting a variable. This is described by a `Database` interface used by Omegahat.

The commitment to programming to an interface rather than a class is a powerful idiom. It encourages developers to define the minimal set of methods required to perform a task. This encapsulates the method from the implementation details of the objects on which it is operating. This encapsulation provides an enormous benefit by allowing users to substitute arbitrary implementations that the original developer may never have imagined. And since one can extend interface classes, the same flexibility is provided as with regular classes. Methods written in terms of interfaces give their users more freedom than those written in terms of concrete classes. This ability to substitute arbitrary implementations of an interface provides an extra layer of extensibility and customizability in addition to the object oriented inheritance. It is this that makes the Omegahat classes flexible for use in contexts other than the Omegahat system by itself.

We should note that programming to an interface does not limit the development of the body of a method to use only the methods of an object's interface. Java differs from some compiled object-oriented languages by allowing a method to dynamically query the class of an object at run time. In this way, we can check whether an object declared in terms of an interface is an instance of a particular class and then exploit that knowledge. In this way, a method can by default use the methods available from an interface and yet take advantage of specializations offered by particular classes that implement that interface.

Unlike many other object-oriented languages, Java does not permit multiple inheritance whereby a class inherits code from two or more base classes. This is a tradeoff between, on the one hand, the complexity both at the user and system level introduced by this mechanism and, on the other, the ability to express dependencies between classes and code reuse. Java uses interfaces as a compromise to provide a limited form of multiple "class" extension without the code reuse. This is a potential weakness of the Java language

for the development of large systems.

Interfaces are the corner-stone of the Omega project. Ideally, the results of the project will be a collection of packages consisting of interfaces for statistical data structures and operations, models, graphics, simulation, etc. and a parallel collection of classes which provide implementations of these interfaces. This will provide the maximum flexibility for the users and allow the components of the system to be used in different ways and scenarios.

3.4 Events & Interfaces

The *observable* (or publish-subscribe) design pattern [7] is another mechanism that we use to facilitate customization and specialization of the Omegahat classes. Similar to the Model/View/Controller idiom ([6], Chapter 1), the concept of the observable pattern is that the observable object notifies listener objects when certain events occur. These listeners react in whatever manner is appropriate based on the changes to the observable object. This approach is commonly used to implement linked plots and other event-based dependency actions.

The observer approach can be used in the example of the class `ConstrainedAlgorithm` above. Rather than specify the check on the parameter value in the `step()` method using inheritance, we can specify the check as the body of a listener and separate these two concerns into different code segments. This listener registers with the `Optimizer` object that invokes the `step()` method within its loop. At the conclusion of each iteration, it notifies all the listeners of that event by calling the method specified in the interface defining that listener. At this point, the bounds on the parameter value can be enforced. The `Optimizer` can agree to examine the return value from calling the listeners method and terminate the loop if any listener indicates that this is appropriate. Again, considering how one might provide this customizability generally in Fortran or *C* algorithms illustrates the benefit of the observable approach for easy customization and extensibility.

The flexibility provided by this mechanism may not seem compelling in our simple example. As more complex object dependencies evolve, this separation makes for more understandable and dynamically modifiable applications. A system-level example again relates to the basic database classes. Observers can register interest in a variety of different database events such as assignments and deletions, attachments to the search path and so on. One use of this is to update graphical browsers that display the variables within the system. A second type of listener is a debugger that monitors assignments to a particular variable. An important aspect of this approach is that the code of the observable class does not know either the number or the nature of the listeners at development time, only their interface. The listeners are in some sense anonymous with respect to the observable. This type of generic notification mechanism can be used for arbitrary purposes and provides interesting possibilities for developing new statistical methodology in dynamic systems such as manufacturing, finance, Web based environments, etc.

In Section 4.8, we will show how listeners can be specified interactively as interpreted Omegahat expressions or functions. This allows one to avoid compilation of any classes, facilitating rapid development of once-off code.

The observer concept is taken one step further in the Omegahat *Optimizer* class. A listener can be registered as either a regular listener or as a background listener. The latter type is notified of the step event by the optimizer, but in a separate thread. In this way, the optimizer continues performing its duties of notifying the regular listeners and completing the next step. This separation of listeners allows those such as the one that constrains the value to work synchronously, while an other that plots the values of the parameters and is not vital to the step-wise actions of the algorithm does not slow the algorithm down.

In this Section we have illustrated the advantages of object oriented programming, specifically using the Java language concepts. This leads to a collection of classes that is highly customizable in several ways.

Inheritance and programming to an interface rather than a particular class definition allows for substitution of classes with implementations not known to the developer of the original class. Identifying useful events and broadcasting these to anonymous listeners allows users to further customize the environment with simple programming techniques that do not require deep understanding of the environment. Omegahat puts these system development tools within easy reach of the data analysts to experiment with and implement even more ambitious analysis tools

4 The Omegahat Interactive Language

So far, we have focussed on how we might better develop the system components so that they can be used in different applications and easily customized. Now we turn our attention to how end users might interact with these components. The value and need for a high-level, interactive language is discussed in [5] and exhibited by the existing systems. In this section, we discuss some of the features of the Omegahat interactive language and its computational model. We illustrate how we provide *direct* access to Java from the interactive language. Also, we discuss aspects of the language that reduce the divide between an interactive language and a compiled, general purpose language.

4.1 Seamless Interactive Access to Java

An important aspect of the early *S* system was the ability to call the underlying *C* and Fortran routines directly from within an interactive environment. Inputs for calls to these routines were created via earlier commands and assigned to variables. In some sense, this is the same model as the canonical UNIX shell where the concept of routines is replaced with applications. The important characteristic of these two interactive languages is that they provide direct access to the internal primitives of the underlying system – routines and applications, respectively. They also provide basic facilities for managing variables and flow control via loops, conditionals and so on. Another important characteristic is that they support both casual, interactive use and also more structured development within the same framework.

These languages provide a user-level language for connecting the internal components of the system. One difficulty with these however is that the language is unaware of the details of the system primitives. The shell cannot determine whether a call to an application has the correct form for the arguments. Similarly, *S* cannot automatically know how many arguments a *C* routine expects or what data types these should have. Suppose, however, that a system was able, at run time, to locate an internal routine or method by name and determine information about the number and type of its parameters. In such a system, users would be able to interactively invoke arbitrary routines that were not known to the developer of the language when it was created. New libraries could be added and their contents immediately accessed. There would be no need for writing functions that connected the interactive language to the low-level routines. The system itself would consist only of a generic mechanism to identify the appropriate routines, report potential argument mismatches and provide variable/object management facilities. Coupled with a centralized memory management facility, such a system would support an interactive language with *seamless access* to the implementation language and its primitives.

An immediate consequence of such a system is that system-level functions would be handled in a manner identical to third-party packages. They need not be specially built into the system. This allows one to completely customize the user's environment including replacing the language and interpreter, even after the system has been initialized. Indeed, the default environment can be used to define a replacement system, allowing different user interfaces to be created using the standard Omegahat language. Additionally, the same routines used by the system to discover routines can be exploited by the user. This allows the user to

explore libraries dynamically and learn about the available functionality. The source of the information is the actual class definitions, rather than potentially out-of-date documentation.

Java supports the vital method and parameter discovery by what is called *reflectance*. As we mentioned in Section 3.2, the fundamental unit in Java is the class. Reflectance allows us to retrieve, from within a running application, virtually all the information about any Java object. This information is dynamically made available in the form of regular Java objects within the application and can be used directly to invoke methods and constructors and assign and retrieve the value of a field.

Similar to the Java scripting languages Pnuts [13] and BeanShell [2], we use the reflectance of Java classes to implement an interactive language. This is the Omegahat language and it is a superset of the Java language, with all of its rich facilities for computation and control and the valuable extensions to *C* described in Section 3.1. Additionally, facilities and constructs borrowed from the languages of the existing statistical computing systems have been added to make it more suitable for interactive statistical usage. Because Omegahat is more closely aligned to the Java language, it has several differences from the interactive languages offered by all of these other systems. Firstly, functionality primarily comes from methods in objects, rather than functions. Thus, it is common to use syntax such as

```
device.plot(x,y)
```

to operate on the `device` object, rather than pass `device` as an argument. Secondly, and more importantly, Omegahat uses pass-by-reference semantics rather than copying the value of objects when they are used as arguments to functions or methods. This means that the contents of objects can be changed when calling different methods, even when the command results in an error. This is very different from the functional approach used in *S*, *R* and *Xlisp-Stat.* It does however make more efficient use of memory and makes certain programming tasks more natural to express.

Objects can be created via the **new** operator such as

```
new Histogram(new RandomNormal(1000))
new util.Vector(10)
```

Such expressions are evaluated by the Omegahat interpreter by resolving the class names (e.g. Vector and Histogram) to the appropriate class. Then, we use reflectance on the class to retrieve the list of constructors. We compare each of the arguments in the user's call to the corresponding parameter of each of the constructor methods and determine the closest match. Dynamic determination of an object's class allows Omegahat to simplify interactive use. It may coerce a user's argument to the class of a corresponding parameter (e.g., scalars to arrays of length 1), freeing the user of this detail. Having identified the appropriate constructor, the interpreter calls it via its `invoke()` method, supplying the user's arguments. This same approach works for method calls. Accessing an object's field is even simpler as it does not involve matching parameters and arguments. Optional type specification (Sections 4.3 and 4.4), casting and calling methods of a base class (i.e. **super**) provide the user with further flexibility to identify different methods.

The simple interpreter, based solely on reflectance, provides an interactive version of Java, allowing the user to evaluate individual Java-like expressions with immediate results. This is significantly simpler than compiling entire Java classes and applications. We hope it will encourage those unfamiliar with Java to explore its many libraries and to investigate the object oriented approach. The additional features we discuss in the following sections illustrate that Omegahat is more than just a simple language for learning Java, but an interactive computing and development environment.

The idea of reflectance is central to Omegahat and is also familiar to users of *Xlisp-Stat* and *S*. It allows us to operate on the language objects by programming within the very same language. This allows us to do such things as write functions that write functions, dynamically determine the free variables referenced in a function call, determine which methods will be invoked in different circumstances, and so on. However, the

significant distinction between reflectance in these interpreted languages and those in Omegahat is that the reflectance is coming from the internal implementation language, Java. The result is a seamless connection between the interactive language and the implementation language, allowing trivial integration of other libraries, partial compilation of interpreted code (see Section 4.4), and other interesting possibilities.

The simplicity of an interpreter based on reflectance encourages its reuse in varied circumstances, and with different user interfaces. Different syntaxes, graphical interfaces and visual programming tools can map different language constructs to the same internal expressions used by the evaluator. The same expressions can be created programmatically to emulate these environments. The use of a modern parser generator tool such as Antlr (Another Tool for Language Recognition) [16] allows us to use object-oriented techniques when creating different user languages (grammars). These techniques will help us to experiment with different styles of user interaction and integrate specialized cross-language constructs such as modeling formulae and SQL databases commands within the Omegahat language (see Section 5).

4.2 Class Resolution

Beginning users of Java are often frustrated by the technicalities of correctly resolving class references (e.g. setting the class path, importing specific classes, and using fully qualified class names). Omegahat removes many of these details. Classes are referenced using enough of their package-qualified name as is necessary to avoid ambiguity. Omegahat performs the appropriate matching by maintaining a list of the classes in each of the packages to which it, and the user, has access. This allows the order in which it matches class names to be dynamically modified. Also, new class packages can be introduced into the system during the Omegahat session. This not only makes using Java significantly simpler for new users, but also more appropriate for an interactive environment.

4.3 Optionally Typed Variables

Variables (names) are used in Omegahat, as in other interpreted languages, to refer to and assign objects. Unlike many of these other languages, Omegahat allows users to associate a *type* or class with a particular variable name. This is optional however, making Omegahat different from strongly typed languages such as *C* and *Java*. This mechanism allows the creator of a variable to provide additional information about it, essentially declaring a constraint on that variable which the system should enforce. This can be used to ensure that subtle errors caused, for example, by assigning an erroneous value to a variable in lengthy scripts are caught immediately at the point of the error rather than in some apparently unrelated segment of the code or method call. This error handling happens because the Omegahat interpreter checks potential assignments to variables that have been given type information by the user/developer and ensures the value of the assignment is compatible with that class. An exception identifying the incompatibility is thrown when a mismatch is detected. As well as benefiting from this system check, the type information about a variable is of value to the user of the code. It makes it easier to understand the intentions of the author and the constraints for the different variables.

Since type information for a variable is optional, there are three different ways to create a variable in the Omegahat language: regular assignments, declarations and definitions. For example,

```
x = "An non-typed variable";
String x;
String x = "declaration and definition";
```

all result in the creation of a variable name `x`. The first imposes no constraint on subsequent values assigned to the variable. The other two restrict future assignments to be `String` objects. In the absence of a type

specification, `Object` is assumed, or more correctly no type checking is performed. Type information can be added to a variable after it has been created.

The *optional type specification* is a very powerful compromise between the strongly-typed compiled languages and the untyped scripting languages such as Perl, TCL, *S*, *R* and *Xlisp-Stat.* Rather than urging the programmer to choose between using a member from either of these two families [15], Omegahat allows users to initially employ the rapid prototyping style associated with scripting languages. As programs evolve, additional type information can be incrementally added, deferring tedious checks to the system to ensure validity of the inputs at each step. This gives users the best of both worlds. Since the system is dynamic, type specifications can be changed programmatically, allowing modifications to be easily incorporated.

The current implementation of the optional type specification is also an example of the benefits of object-oriented programming described in Section 3.2. We extend the basic *Database* class that is used to store variable names and their values. The new class provides additional capabilities for storing the type information, and hides the details of the validity checking in its `assign()` method. The use of inheritance allows us to transparently introduce this new capability, but only for those who need it. Others who do not use the type specification interact with the basic *Database* class and do not incur performance penalties associated with unnecessary checking for the existence of type information.

4.4 Functions

Evaluating individual Java expressions and developing Java classes is a useful facility for casual exploration. However, for convenient interactive use, Omegahat supplements these with interpreted functions, which are simpler to define, modify and use than general Java methods and allow groups of Omegahat expressions to be invoked simply. Omegahat Functions are regular Omegahat variables and can be defined using simple assignments as described in Section 4.3. As one might expect, variables created within the function are local to that function. Unlike *S*, **static** variables can also be defined within a function with the effect that its value persists across all calls of that function. Putting all these together, a simple example of a function is

```
random = function(n=10) {
  static int seed = initializeSeed();
  return(new Random(n, seed));
}
```

The language supports default values for parameters, argument matching by name and variable number of arguments (i.e. "..."), all borrowed from *S*. (Note that the body of the function uses other functions and Java classes and methods available via the seamless access.)

It is often convenient and more flexible to be able to call Java methods as if they were Omegahat functions. Omegahat allows this by maintaining an additional list of objects or classes it searches when resolving a function name. This allows existing Java methods to be used directly and naturally as part of the Omegahat language and users to benefit from the improved performance from using compiled methods. This mechanism also supports variable number of arguments and lazy evaluation (see Section 4.5), and is used commonly for creating functions such as *Switch()*, *time()*, *Quote()*.

In the same way that variables can have associated type information (Section 4.3), so too can each parameter of a function, and this is also optional. As with type specification for variables, the interpreter automatically checks the arguments are compatible with the declarations as it assigns them to the call's database. Errors are signaled when mismatches are detected. Leaving such duties to the system simplifies the development and understanding of the code by removing the need for explicit checks often found in the body of functions when using *S*-like languages. An example of how this might be used is

```
histogram = function(double[] data, int numBins = 10,
```

```
                              title="Histogram")
{
  h = new Histogram(data, numBins);
  h.setLabel(title.toString());
 return(h);
}
```

Here, we constrain only the first two arguments. The last argument can be of any class, defaulting to `Object`, since it need only support the `toString()` method available for all Java objects.

Functions can also be defined using a slightly different syntax (more common in Java) that allows the class of the return value to be specified:

```
Histogram  function histogram(double[] data, int numBins = 10,
                                    title = "Histogram")
{
  ...
}
```

Alternatively, this information can be specified programmatically after the initial definition using the `returnType()` method of the function object. Again, the system ensures that the actual return value from a call to this function is compatible with this declaration.

As mentioned earlier, the optional type specification leads to greater clarity in the code and avails of automatic system level checking to catch errors when they occur. There are additional benefits that come from providing this information in the context of functions. We can make the expressions in the function's body more efficient by processing the function when it is defined. Knowing the class of a parameter or variable allows us to replace calls to its methods, constructors and fields just once, at definition time, rather than during each call to the function. For example, in the *histogram()* function above, we can substitute the generic call to the `Histogram` constructor with a call to the particular constructor

```
org.omegahat.Graphics.Plots.Histogram(double[], int)
```

and avoid matching the arguments at evaluation time. This provides significant performance gains for the interpreted language. Since it can be performed at any time after the function has been defined, the user has control over when to migrate from prototyping to production level code and back.

We can go one step further than resolving methods when the function is defined. We can compile the Omegahat expressions into Java byte-code. This will then allow us to respond to the two challenges mentioned in Section 2. Providing functions developed in Omegahat as Java byte-code allows us to export statistical methodology for use by others in more common programming environments. Also, compilation provides a convenient route for generating efficient code from interpreted prototypes. We think this is a powerful mechanism by which we can also reduce the separation between the interactive user and package developer by connecting these two ends of the programming spectrum.

The Omegahat extensions and interpreted evaluation necessarily introduce some semantic differences from "pure" Java, which may be important in compiling functions. Fortunately, most user functions will have reasonably clear semantics, and compilation will produce the correct result. Resolving the tension between faithfulness to Java and interactive convenience will be part of our future research. The user's wide choice of options for compiling, optimizing and mixing Omegahat and Java code (Section 4.8) allows for different strategies to be used in different contexts.

4.5 Optional Lazy Evaluation

The *S* language uses lazy evaluation in all function calls. The effect is that in-lined expressions passed as arguments are not actually computed until needed. This avoids unnecessary computations if the argument is not actually used because of conditional branching, an error, or whatever. While this is desirable in many cases and can also simplify specification of certain default values for arguments, lazy evaluation can also lead to difficulties in (quickly) understanding the effect of certain expressions. At its ugliest, it can force programmers to circumvent it by simply referring to variables in order to cause the computation to occur. However, in interactive use, it is a powerful mechanism for avoiding un-needed computations.

The Omegahat language provides a compromise between using lazy evaluation in all circumstances and not supporting it. Instead, programmers can chose which approach to take on a function by function and parameter by parameter basis. The keywords **lazy** and **eager** are introduced into the language to qualify parameters. The definition

```
foo = function(lazy x, y) { .... }
```

specifies that the argument **x** is to be evaluated only when it is first used, but that the argument **y** is to be evaluated when the call is made. As a convenience, the default argument mode for a function can be specified for the entire function as in the equivalent specifications

```
lazy  function(x, eager y) { .... }
eager function(lazy x, y) { .... }
```

Since Java does not directly support lazy evaluation, introducing such facilities in Omegahat complicates the compilation of Omegahat functions to Java byte-code. Whether it is sufficiently useful remains to be seen. However, it is an illustration of the experimentation we can do in this new system with little effort.

4.6 Methods and Interpreted Classes

The ability to define functions and also specify type information for individual parameters can be combined to support overloaded or generic functions. By this, we mean that we can have several functions, all with the same name, but which are differentiated from each other based on what types of arguments they expect. For example, we may have two *plot()* functions, one which produces a single scatter-plot from a pair of variables, and another which creates a scatter-plot matrix for all pairs of variables within a dataset. Using the same name, **plot**, for both functions simplifies the interface for the user and is more consistent with Java. In Omegahat, we can define such methods as

```
plot = function(x, y) {
  new Scatterplot(x,y);
}

plot = method(DataFrame data) {
  ctr = 0;
  plots = new Scatterplot[(data.size()-1)^2];
  for(i in data.names()) {
   for(j in data.names()) {
     if(i != j)
       plots[ctr++]  = new Scatterplot( data[[i]], data[[j]]);
   }
  }
}
```

(Note the use of the keyword method in the second function to indicate that we should not overwrite the value of the existing variable, but add this function to the collection of methods named **plot**.) For a particular call to an overloaded function, the Omegahat interpreter is able to identify which function is appropriate. It uses exactly the same approach as that used for the method dispatching for regular Java objects, by dynamically creating a Java class with methods corresponding to the different functions.

Overloaded functions lead naturally to defining interpreted classes, and having functions within a class access variables local to an object of this interpreted class. Omegahat provides two approaches for interpreted classes. One approach uses interpreted tables to store an object's fields and methods. The other uses extensions to the idea of dynamically compiling new Java classes and implementing the methods with calls to Omegahat functions. More details are available at [14]. The important idea is once again that the Omegahat language provides a smooth way to migrate from casual interactive use to software development via a route that allows, but does not require, structure to be specified.

4.7 Operator Overloading

S provides many convenient ways to subset different objects in different ways. The operators [and [[are powerful features of the language. Users can add their own methods for this operation that act on different types of objects and so extend the system. This is true for other operators such as the basic arithmetic operators and others such as $\%*\%$. This ability to treat these language operators as regular methods which can have different actions for the different classes of their operands is known as operator overloading.

While *Java* does not (yet) provide support for operator overloading, Omegahat does, and at several different programming levels. Users can specify Omegahat functions or compiled Java methods to be used for this purpose. Additionally, developers of new classes can elect to implement certain Omegahat interfaces to allow them support the Omegahat language operators. This style of extensibility is discussed in [18] and again illustrates the meta-programming style of development with which Omegahat was itself developed, allowing other systems to be developed from the same tools.

4.8 Accessing Omegahat Objects from Java

The goal of seamless access between Omegahat and Java implies the ability to use Omegahat language objects from Java methods, and not just accessing Java from the Omegahat language in Section 4.1. Specifically, we want to allow users to pass Omegahat functions, expressions, and interpreted classes as arguments to Java methods and to have Java operations on these be performed within Omegahat. Being able to use objects from either of these two languages in the other gives the user freedom to program in whichever is most appropriate given immediate constraints. She can gradually migrate code from the interpreted language (Omegahat) to the compiled one (Java). This gradual progression from interpreted code to Java provides performance gains and also independence from Omegahat. Using Omegahat functions as event handlers for graphical interfaces is a good example of the flexibility gained from mixing the two languages. The handler functions can be modified without the need for recompilation, leading to a very rapid development cycle.

Since Omegahat is dynamic and interpreted in senses that Java is not, attaining this bi-directional seamless access between the two languages requires explicit support. The Omegahat environment provides ways to "wrap" functions and expressions in order to make them act as instances of arbitrary Java classes. An example will illustrate this approach. Consider the optimization algorithm and the use of an observer described in Section 3.4 to test the value of a parameter exceeding a user-specified value (10). This test can be easily written as an Omegahat function as

```
function(parameters) {
  if(parameters.theta > 10)
```

```
    stop("Parameter outside user-specified range.");
}
```

It would be convenient to register this function directly with the `Optimizer` object so that it would be invoked at the end of each iteration of the optimization procedure. Unfortunately, the `Optimizer` object knows nothing about Omegahat functions, and the function does not implement the appropriate interface expected by the `Optimizer` object. Let's suppose that this interface is named `OptimizerListener` and has a single method `iterationPerformed()`.

To be able to treat the function as an `OptimizerListener` object, we need to define a new class. The class would store the user's function and implement the `iterationPerformed()` method of the required interface. The body of this method would simply call the function with the arguments it received from the `Optimizer` object's call. This method would be as simple as

```
public void iterationPerformed(Parameters params) {
 Object[] args = new Object[1];
   args[0] = params;

    /* Now call the user's function with
       the arguments to this method call.
     */
  eval(params)
}
```

Note that we would have to create such a class for each interface, and implement each method of those interfaces in the same way. Omegahat provides a tool that does this dynamically. It uses reflectance on the target interface classes to determine the methods it must implement. It implements the methods using the same approach as the example above, but by dynamically generating Java byte-code to define the method's body and the entire class. And finally, the Omegahat interpreter loads this new class and creates an object of the appropriate type to contain the user's function or expression. This is sufficient to provide seamless bi-directional exchange between the Omegahat and Java languages.

4.9 Extending the Evaluation Model

The object-oriented paradigm described in Section 3.2 allows us to extend the Omegahat language as well as the core statistical classes. Omegahat provides several interpreter classes that offer different functionality that can be used in different sessions and contexts. Indeed, multiple interpreters can be used concurrently in a single session, supporting multiple-users working either separately or sharing different variables, graphics devices, etc. Two simple extensions to the interpreter illustrate the flexibility of the different interpreter classes, and specifically the capability of transparently integrating distributed computing into the Omegahat environment.

Omegahat allows a table within a remote database application to be treated as the source of Omegahat variables. Each column in the table is a separate Omegahat variable. Let's suppose that we have a table with two columns, `x` and `y`. Omegahat commands such as

```
x[x < y]
y[x == 1 && y > 10]
```

would usually be evaluated by fetching values for `x` and `y` (potentially multiple times) and then performing the logical operations and sub-setting. A special SQL-oriented Omegahat evaluator can infer that the two

variables are located in a remote database table. It can then generate an appropriate SQL command to perform each expression entirely within the remote database. This approach leads to fewer communications with the database and a reduction in the quantity of data transferred between the database and Omegahat. The details are transparent to the user, making it easy both to use and replace with another implementation. Omegahat allows us to easily experiment with more complicated scenarios, such as when variables are distributed in different tables, databases and on different machines.

Similar distributed facilities have been added to an Omegahat evaluator for allowing remote server objects in other applications to be treated as Omegahat variables. Named servers are collected in CORBA naming contexts or folders, and a context can be used as an Omegahat database. Methods can be invoked on these servers using the same syntax as regular, local Java methods calls. For example, an Omegahat user might instruct an XGobi [20] server to load a new data set via an Omegahat command such as

```
xgobi.setData("myDataset.dat").
```

The Omegahat interpreter recognizes the special nature of the `xgobi` variable, and dispatches the call using an appropriate mechanism.

The ability to invoke methods on servers on other machines and in different applications can be used to implement parallelized versions of loops and other computations. Using the reflectance of Omegahat's own interpreted language, under certain circumstances, an Omegahat interpreter can identify parallelism in a user's command and execute several expressions concurrently. Again, this can be done transparently and as an extension to the default system.

5 Extending Existing Systems

In addition to the Omegahat environment, the Omega project will focus on the concept of multiple systems that freely interact and share resources – the networked "multiple agent" paradigm. In particular, we do not intend Omegahat, at this stage, to be a complete statistical environment that can be used as a replacement for any of the existing systems. A number of current efforts, many of them within the Omega project, will help give existing systems added functionality and allow Omegahat to share services with them.

- John Chambers and I have developed facilities in both *R* and *S* for providing interactive access to Java classes and methods in much the same way *C* and Fortran routines are accessed. By combining a JVM and an Omegahat interpreter into the running sessions, users can take advantage of the Java libraries from within these high-level interactive languages. Network access, database connectivity, graphical interfaces and so on can all be used directly and portably within these systems. This will be described in a future paper.

- Facilities for supporting the inter-process communication or client/server paradigm have been developed for *Xlisp-Stat* using Microsoft's Component Object Model (COM) [22] and *R*/*S* using CORBA. These allow users of these high-level environments to easily integrate services from other applications as well as provide services using functions within these interactive languages.

- Tools within *R* and *S* to read and write XML files will potentially simplify exchanging data and object files between these and other systems, including Omegahat. This format may allow easier integration of other tools that use variants of XML, such as MathML for mathematical expressions, GML for geographical map specification, SVG for vectorized graphics descriptions. Also, it provides us with another interface for users who don't wish to use traditional programming languages.

- Several projects are currently working on providing database connectivity within *R* and *S* for different database management systems. As these progress, dynamically accessing data from distributed sources and deferring computations to more appropriate applications may be feasible.

Again, the idea is to provide more choices for developers so that they can use the most appropriate tool for their task. All of the efforts above provide useful extensions to this end and need to be incorporated into common use to determine the properties and merits of each.

6 Conclusion

We have discussed the need for enhancing existing systems. It is essential that these support interactions with other systems, sharing data and functionality both statically and dynamically. Additionally, we need to research new statistical computing paradigms. Development of new systems that utilize the new tools available to us both now and in the future is important. We have presented some reasons for using Java as an implementation language for new computing systems and components and hope these contribute to a productive discussion on the value of different approaches. We have also outlined some advantageous features provided by the Omegahat interactive language that provide significant assistance in many aspects of the software use and development processes. Specifically, optional type specification allows structural information to be used by the developer, read by the user and processed by the system in an incremental, evolutionary style different from either scripting or compiled languages. The use of reflectance is also important for providing bi-directional seamless access between the low-level implementation language used for components and the higher-level user languages.

It would be frustrating to see the limited resources of the statistical computing community continue to focus on existing tools that are unlikely to evolve in ways that allow them to handle new demands. The general statistical community hopefully will encourage and reward research on these issues, and be keen to learn the "new" paradigms such as object-oriented and event-driven programming, parallel and distributed computing, and basic software engineering principles.

I believe that the architecture described in this paper provides a strong foundation on which we in the statistics community can build future software to meet not only our own needs, but those of other disciplines who would benefit from sophisticated statistical techniques. The strength of this approach relies heavily on new tools and technology that have only become available recently. The Omega Project provides a unique umbrella under which different ideas can be explored by, and for, the entire community of interested users.

7 Acknowledgements

Many of the ideas and the content of this paper have evolved from my work with John Chambers. Both he and David James have helped considerably in presenting the ideas here. Also, thanks are due to the editor for organizing this special issue.

References

[1] Jeff Banfield. Rweb: Web-based Statistical Analysis. *Journal of Statistical Software*, 4, March 1999. http://www.stat.ucla.edu/journals/jss/v04/i01.

[2] The BeanShell: Lightweight Scripting for Java. http://www.beanshell.org/docs.html, February 2000.

[3] John Chambers and Duncan Temple Lang. The Omegahat Interactive Language. http://www.omegahat.org/Environment/Introduction.ps, September 1999.

[4] John M. Chambers. *Programming with Data*. Springer Verlag, 1998.

[5] John M. Chambers. Users, Programmers, and Statistical Software. *Journal of Computaional and Graphical Statistics*, 9(3), September 2000.

[6] Robert Eckstein, Marc Loy, and Dave Wood. *Java Swing*. O'Reilly & Associates, Inc., 1998.

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, and Grady Booch. *Design Patterns : Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing. Addison Wesley, 1995.

[8] The GNU Compiler for the Java Programming Language. http://sourceware.cygnus.com/java/, current Feburary 2000.

[9] Robert Gentleman and Ross Ihaka. Java and Dylan. *Journal of Computational and Graphical Statistics*, 9(3), September 2000.

[10] Wolfgang Hoschek. The Colt Distribution. http://nicewww.cern.ch/~hoschek/colt/index.htm, current February 2000.

[11] Ross Ihaka and Robert Gentleman. R: A Language for Data Analysis and Graphics. *Journal of Computational and Graphical Statistics*, 5(3), 1996. pages 299–314.

[12] JAMA Home Page. http://math.nist.gov/javanumerics/jama/, current February 2000.

[13] John H. McCoy. Scripting for Pnuts. *Dr. Dobb's Journal*, 25(308), January 2000.

[14] The Omegahat Home Page. http://www.omegahat.org, July 1998.

[15] John K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3), March 1998. pages 23–30.

[16] Terence Parr. The Antlr Home Page. http://www.antlr.org, current February 2000.

[17] Güenther Sawitzki. Extensible Statistical Software: On a Voyage to Oberon. *Journal of Computational and Graphical Statistics*, 5(3), 1996.

[18] Guy Steele. Growing a Language. *Journal of Higher-Order and Symbolic Computation*, 12(3):221–236, October 1999.

[19] G. W. Stewart. The Jampack Home Page. ftp://math.nist.gov/pub/Jampack/Jampack/AboutJampack.html, current February 2000.

[20] D. F. Swayne, D. Cook, and A. Buja. Xgobi: Interactive Dynamic Data Visualization in the X Window System. *Journal of Computational and Graphical Statistic*, 7(1), 1998.

[21] Luke Tierney. *LISP-STAT: An Object-Oriented Environment for Statistical Computing and Dynamic Graphics*. J. Wiley & Sons, New Your, NY, 1990.

[22] Luke Tierney. Connecting Lisp-Stat to COM. *Journal of Computaional and Graphical Statistics*, 9(3), September 2000.