In the past, seismic processing was best organized as UNIX filters that were programmed in procedure-based languages, such as C or Fortran. Today, the replacement of traditional processing flows by more complicated geophysical inversion formulations requires more powerful object-oriented languages. Additionally, the arrival of the World Wide Web offers new collaborative models between the in-house departments of oil companies, outside contractors, and software companies.

As a response to these challenges and opportunities I wrote *Jest*, a Java library for seismic estimation that aims to enable programmers to prototype and test complex geophysical processing problems on field data sets. The Java implementation offers the distribution, execution, and maintenance of the seismic processing flows through the World Wide Web. Java itself is a popular, portable and powerful object-oriented programming language. If efficiency requires it, Java facilitates the easy and transparent substitution of compute intensive subroutines by native C-routines. The Jest library is freely-available at the web site of the Stanford Exploration Project[1].

Separation of solver and application. Jest separates optimization and application software without imposing limitations on either. Consequently, experts independently implement complementary and reusable optimization and application software, irregardless of the algorithms' internal complexity. Off-the-shelf optimization packages usually implement their vectors in simple generic data structures such as one-dimensional Fortran arrays. Unfortunately, generic data structures are often inconvenient or unacceptable in the case of complicated vectors or vectors too large to hold in-core. Alternatively, laboratories develop their own in-house optimization software that they adapt to their local data structure and application type. This approach, however, locks a laboratory into a fixed data structure and application style while simultaneously preventing any collaboration with laboratories of a different standard. Additionally, any new procedural-based routine has to comply with the calling sequences of existing routines. A new routine can easily take advantage of old programs by invoking their known calling sequence. However, a new routine is not easily substituted for an existing alternative unless the two programs share identical calling sequences: a coincidence that is not easily maintained among a diverse group of contributing programmers.

Mathematical interfaces. Instead Jest is centered around a set of abstract mathematical interface classes – *Jam* (Java for mathematics) – that resolve the implementation- and invocation-compatibility problem of traditional optimization libraries. Jam defines names and method invocations for basic mathematical entities (vectors, operators, and solvers) and their standard operations. For example, a vector includes a `add()` and `scale()` method. Jam interfaces do not prescribe any

[1] `http://sepwww.stanford.edu/oldsep/matt/jest`

particular implementation, such as a vector representation.

Collaboration among classes relies on the method invocations defined by the Jam interface. A Jam-compatible class implements a Jam interface and limits itself to use only Jam-defined interface methods. A programmer of a Jest class is not concerned with the implementation or invocation of other Jest classes but only with the invocation of the relevant Jam interfaces. Furthermore, the Java compiler ensures that any class that implements a Jam interface does indeed supply all necessary methods. Consequently, Jam offers a plug-and-play quality: a programmer can accept or replace any part of Jest and preserve compatibility by implementing the corresponding Jam interfaces. A programmer may even extend Jam with additional mathematical object interfaces.

Ultimately, the Jam interface is as complete and general as the original mathematical definition. Furthermore, the mathematical framework of Jest is shared by researchers in numerical analysis, engineering, and natural sciences.

Example solver. A conjugate-gradient solver, for example, is programmed in terms of the general Jam interfaces such as a vector and an operator with adjoint. The interface terms allow a literal translation of the mathematical algorithm 1-5 (Gill et al., 1981) to the Java routine below. More importantly, the program's instructions are as general as the algorithm, since the program's objects – vectors and operators – are of the same abstraction as the algorithm's mathematical symbols. The solver is compatible with any classes that implement the Jam interface of a vector and an operator.

$$p_k = -r_k + \beta_{k-1} p_{k-1} \tag{1}$$

$$\alpha_k = \frac{|r_k|_2^2}{p_k^T B p_k} \tag{2}$$

$$x_{k+1} = x_k + \alpha_k p_k \tag{3}$$

$$r_{k+1} = r_k + \alpha_k B p_k \tag{4}$$

$$\beta_k = \frac{|r_{k+1}|_2^2}{|r_k|_2^2}. \tag{5}$$

The example's simple solver interface defines a single method, `solve(operator B, vector y, vector x)`. The arguments stem from the equation $\mathbf{Bx} = \mathbf{y}$. The `solve()` invocation requires a starting solution for $\mathbf{x}$ which it then updates to its best estimate. The solver's termination criteria are usually set by the solver's constructor.

```
package juice.solver;
import  jam.solver.*;
import  jam.vector.*;
import  jam.operator.*;
/** Conjugate Gradient Solver */
public class CGSimple implements LinearSolver {
  private int maxIter;   /** @param maxIter maximum number of iterations. */
  public CGSimple(int maxIter) { this.maxIter = maxIter; }
  /**
   * @param A operator to be inverted
   * @param b known vector
   * @param x starting estimate of unknown vector
   */
  public void solve(Operator A, Vector b, Vector x ) {
    if (! (A instanceof LinearOperator))
      System.err.println("Error: A is not a LinearOperator");
```

Package name	User Group	Class type	Vectors	Operators	Solvers
Jam	**Anyone**	Interfaces of abstract math objs	Vector	Operator CompoundOperator	Solver IterativeSolver
Juice	Mathematicians	Classes of numerical analysis	Rn	MatrixMultiplication	CG solver, Newton-Raphson ..
Jag	SEP geophysicists	Classes of seismic processing	Rsf, Isf	Nmo, convolution ...	

Table 1: Package hierarchy. Jest packages (class libraries) are separated by user community. The top package, Jam, is shared by all users and only includes interfaces for abstract mathematical entities, such as a vector and its operations (add, scale, etc.).

```
  Vector r = A.getRange().newMember(); // init r = b - Ax
  A.residual(x,b,r);
  r.neg();
  Vector p = (Vector) r.copy();        // init p = r
  Vector v = A.getRange ().newMember();// init v = 0
  float rtrNew = r.norm2();

  for (int iter=0; iter< maxIter; iter++) {
    A.image(p, v);                     //  v = Ap
    float ptv   = p.dot( v);           //ptv = p^t Ap
    float alpha = rtrNew/ptv;          //  a = |r|^2 / p^t Ap
    if (ptv == 0f) System.err.println("Error: ptv=0.");
    x.addScale(  alpha, p);            //  x = x + a p
    r.addScale( -alpha, v);            //  r = r - a Ap
    float rtrOld = rtrNew;
    rtrNew       = r.norm2();
    float beta = rtrNew/rtrOld;        //  b = |r|^2 / |r|^2
    if (rtrOld == 0f) System.err.println("Error: rtrOld=0.");
    p.addScale( beta, r, p);           //  p = -r + b p
}}}
```

Packaging. Jest packages (class libraries) are separated by user community (see Table 1). The top package, Jam, is shared by all users and only includes interfaces for the abstract mathematical entities, such as a vector and its operations (add, scale, etc.). User groups author packages, the classes of which implement Jam interfaces. The *Juice* package, for example, contains many elementary numerical analysis classes (e.g., space of real numbers, matrix multiplication, compound vectors and operators, conjugate gradient solver). The *Jag* package contains my laboratory's multi-dimensional, regularly sampled, physical function – a traditional *SEPlib data cube* – that implements the Jam vector interface. An array of axis objects describes the corresponding vector space. Overall, Jag contains a budding family of seismic processing operators, e.g., convolution, poststack migration, edge detection, nonstationary patching, and missing data estimation.

Any additional package written by a third-party programmer is bound to be compatible if the classes implement the corresponding Jam interfaces. Consequently, any given user group only needs to implement the classes that are unique to its user community. Jest's extendible structure and mathematical objects aims to garner collaboration across disciplines.

Java. Jest's abstract mathematical classes require a full-blooded object-oriented language. We found Java much easier to learn and program than C++. Especially, Java's secure design and its diverse library of high-level routines reduces the time spent debugging. The same secure and abstract capabilities make Java less efficient than C or Fortran. Over the past years, however, Java's

efficiency has been improving steadily, and I expect it to be comparable to C or C++ soon.

Unfortunately, Java lacks a complex primitive type and does not allow the overloading of the index operator, which requires scientific programmers to use awkward work-arounds. I hope, however, the designers of Java will overcome these shortcomings in future releases.

Since Jest is implemented in Java, it potentially facilitates reproducible electronic documents (Schwab et al., 1996) on the Web. Such reproducible Web documents could change the way geophysical researchers and practitioners publish and collaborate, since they make computational results as accessible as a button click.

Literature. Historically, Jest is the latest in a series of optimization libraries. The Stanford Exploration Project developed CLOP (Nichols et al., 1993), an experimental geophysical inversion library implemented in C++. Nichols and Dye redesigned CLOP's class hierarchy to express abstract objects of vector algebra. Independently Gockenbach and Symes of the Rice Inversion Project developed a similar C++ library. In 1994, the efforts of the two laboratories merged into the Hilbert Class Library (HCL) (Gockenbach and Symes, 1996; Gockenbach, 1996). Jest began as a Java implementation of HCL. Fall 1997, Jacob and Karrenbach (1997) implemented a multi-threaded velocity estimation based on Jest.

Lydia Deng et al. (1995) developed a C++ optimization library for geophysical inversion that addresses similar problems as Jest. Fomel (1996) focused on performance and implemented a geophysical optimization library in Fortran90 using its modular features.

REFERENCES

Deng, L., Gouveia, W., and Scales, J., 1995, The CWP object-oriented optimization library: The Leading Edge, **15**, 365–369.

Fomel, S., and Claerbout, J., 1996, Simple linear operators in Fortran 90: SEP–**93**, 317–328.

Gill, P. E., Murray, W., and Wright, M. H., 1981, Practical optimization: Academic Press.

Gockenbach, M., and Symes, W., 1996, The Hilbert Class Library: a library of abstract C++ classes for optimization and inversion: submitted for publication in Computers and Mathematics with Applications.

Gockenbach, M., 1996, The Hilbert Class Library: A library of abstract classes for C++ optimization and inversion: `http://www.trip.caam.rice.edu`.

Jacob, M., Philippsen, M., and Karrenbach, M., 1997, Large-scale parallel geophysical algorithms in Java: A feasibility study: `http://wwwipd.ira.uka.de/~jacob`.

Nichols, D., Urdaneta, H., Oh, H. I., Claerbout, J., Laane, L., Karrenbach, M., and Schwab, M., 1993, Programming geophysics in C++: SEP–**79**, 313–471.

Schwab, M., Karrenbach, M., and Claerbout, J., 1996, Making scientific computations reproducible: SEP–**92**, 327–342.