Chapter 2 The BioJava Core Interfaces

BioJava is intended to provide Java based APIs for common bioinformatics tasks. It also strives to be a convenient basis for writing potentially computationally expensive algorithms. To reduce the learning curve, and to decrease maintenance overhead, individual APIs must be complete enough to allow them to be used algorithmically, but slim enough that they can be easily implemented and used. The balance between making the APIs not only powerful but also small, is sometimes difficult to maintain, but has, in my view, fostered a high degree of elegance in the underlying object design.

There are two clearly different cases where code reuse is beneficial. The first, and most commonly thought of case is the reuse of library code by invoking it from multiple applications. For example, it is very common to reuse a matrix mathematics library during numeric programming. We could call this the “new using old” case. The other reuse case is when library code can execute a tried-and-tested procedure that in turn calls some application specific code. For example, in Java, a listener can be registered with a window to handle mouse movement events. In this case, the library code is responsible for drawing the window and for invoking the listener of mouse movements, but the exact behaviour of the library is customized by the listener. We could call this the “old using new” case.

The design and implementation of the BioJava libraries has primarily been an exercise in computer science, not biology. Throughout, we have striven to foster a high degree of code reuse both by providing APIs that can be used in a wide range of contexts (new using old), and by providing opportunities for developers to drop in new implementations of these APIs without affecting existing code (old using new).

The APIs relevant to this dissertation, and for which I have been primary designer and implementer, are the following:

- Nested Exceptions and Assertions
- Changeability
- Symbols, Alphabets and SymbolList
- Sequence, Feature and SequenceDB
- Distribution
- MarkovModel
- Query

Wherever possible, the API is defined in terms of Java interface definitions, allowing for the seamless integration of multiple implementations. Indeed, it has been the reliance on interfaces that has made the development of the BioJava library relatively rapid and robust. We have discovered that nearly all core data types can be implemented in multiple ways depending upon a host of factors, so the entire toolkit makes as few assumptions about implementation as are possible.

2.1 Java as a Language for Bioinformatics

Java (Gosling, Joy et al. 2000) is a language created by Sun Microsystems, originally for use in imbedded systems such as mobile phones, watches and ABS systems. It relies upon a definition of a virtual machine (VM) (Lindholm and Yellin 1999) that is responsible for thread and memory allocation, byte-code instruction execution and enforcing security restrictions. The byte-code is naturally object-oriented and has support for advanced features such as exception handling and thread synchronization. The byte-code acts upon a stack of working variables, an arbitrarily large set of virtual registers and the object (or class) that is currently in scope. There is no pointer type in Java, or in the byte-code, making it impossible to write byte-code that addresses arbitrary memory. Theorem provers can be used to validate that a given portion of byte-code is safe to execute, avoiding some of the issues with other languages (such as invalid memory allocation, executing instructions on inappropriate types and validation that the execution stack is always in a consistent state).

The Java VM is responsible for interpreting the byte-code and for environment functions such as memory allocation and garbage collection (freeing objects from the memory pool once they are no longer within scope), system calls (IO, process execution, thread management) and managing peers to the native operating system graphical user interface (GUI). With a VM of a given version (e.g. 1.2.2) and any platform (e.g. Sun for Windows, Compaq for Tru64), executing a portion of byte-code should produce exactly the same results, even if the performance differs[13]. This code portability by design is historically one of the major benefits of Java. In practice, platform incompatibilities nearly always arise from platform-specific portions of the VM, such as graphical peers, rather than bugs in the execution of byte-code.

[13] Since version 1.4 of Java, some floating-point mathematical operations may be implemented by processor-specific instructions that do not conform to the IEEE floating-point maths required by the Java virtual machine specification. In the vast majority of cases, this does not change the result of a calculation greatly enough to alter program behaviour. The keyword `strictfp` can be applied to any class or method that must uses the IEEE-compliant math operations, such as numerically intensive code that needs particular overflow/underflow and rounding semantics.

Pure Java byte-code interpretation has historically been slow relative to native code (compiled from C/C++ or FORTRAN, for example), but has always compared favourably to other interpreted languages such as Perl. Recently, with the move of Java from toy examples and small graphical applications to large, demanding applications such as web-server back ends (especially J2EE[14]) and large-scale numerical processing (for example, the Colt matrix mathematics library[15]), a number of technologies have appeared to improve the performance.

Initially, just-in-time (JIT[16]) compilers increased performance of large blocks of numerical code to that comparable with C++ by compiling each byte-code function into native code for the physical processor once a class was first loaded (present even in many Java1.1 VMs). Some Java byte-code instructions could be represented cleanly as one or more simple native instructions (for example, the arithmetic operations). However, many Java byte-code instructions have no direct representation (such as object allocation, or method invocation), so must be converted into calls to the virtual machine. JIT compilers tend to do a good job of increasing the performance of numeric code that resembled more classically procedural programming styles. JITs proved insufficient for many tasks as many Java methods are small, and very often are executed as virtual calls which can not be resolved at compile-time. In addition, due to the highly polymorphic nature of much Java code, it is often impossible to perform optimisations because the simple type-based system

[14] See http://java.sun.com/j2ee/ for information related to the J2EE standard

[15] Colt is distributed from http://tilde-hoschek.home.cern.ch/ ~hoschek/colt/index.htm

[16] See http://java.sun.com/docs/jit_interface.html for more information about Sun's JIT compiler

can't give enough information to know the context within which code will be executed.

The latest family of virtual machines are based upon Sun's Hotspot Virtual Machine architecture[17]. This uses a mixture of several techniques to remove performance bottlenecks and optimise code execution. Firstly, a large portion of Java execution time can be spent in object allocation and garbage collection. This is especially expensive for objects that are allocated and then discarded within inner loops. Hotspot initially flags objects with a creation time, and places them into a nursery area. When more memory is needed, Hotspot first attempts to free objects within the nursery rather than completing a full garbage-collection cycle.

As an anecdotal example of how this can impact performance, I wrote some code that unnecessarily allocated a large number of objects within a tight loop, and then ran the application on a PIII 800MHz with the Hotspot Virtual Machine and a Compaq DS40 with Compaq 1.2.2 Fast VM. After one and a half days, the process on the DS40 had still failed to complete. On the PC, it completed after 110 seconds. Once the unnecessary objects were not being created, the Compaq server took just 20 seconds to execute the code, and the PC took 56 seconds. This clearly indicates the affect of the VM implementation upon performance.

After careful memory management, the second truism of code optimisation is that 5% of the code will account for 95% of the execution time, so if you wish to focus efforts upon optimisation, this is the portion to target. The hotspot VM continually

[17] See http://java.sun.com/products/hotspot/docs/whitepaper/ Java_HotSpot_WP_Final_4_30_01.html for more information about Sun's hotspot VMs

profiles the application, and concurrently optimises each of the execution hot spots. This results in applications increasing in performance as they are run (often by factors of greater than 10 times).

Method invocations make many optimisations impossible, especially if the method is bound at execution time rather than at compile time (for example, virtual method invocations). This is because the optimiser does not know what the side-effects of the invoked code will be, so it can't really perform aggressive optimisations to eliminate redundant code or to reorder instructions across function calls.

In traditional languages, this has been tacked with tactics such as macro-expansion, inlining and by defining many methods as not being over-ridden by subclasses, making the linkage static. The hotspot VM takes another approach by dynamically inlining functions to produce multiple context-dependant compiled and optimised versions of a given portion of an application. Most small functions, such as get/set pairs can be trivially inlined, removing the method invocation overhead completely. Increasingly complex methods may be inlined, allowing loop variables to be merged and larger blocks of code to be optimised. Certain types of objects can be proven to decompose to the set of their fields and methods only (i.e. their object reference is never explicitly used to test for identity), in which case the fields can be allocated on the stack. This is similar in spirit to having the power of a templated method that can be parameterised with the template type during run-time. The result of these optimisations is that hotspot-interpreted Java code that is polymorphic or uses many small methods can often execute at speeds comparable to or faster than similarly polymorphic `C++` code. Currently, dedicated procedural style `C` or `C++` may out-perform similar Java code, but even here Java is making inroads. For example, the

Colt matrix maths library in Java now has comparable performance to the FORTRAN matrix math libraries. There is undoubtedly more work to be done for high-end computation in Java, but it is no longer an insurmountable obstacle to the acceptance of Java for the hard end of Bioinformatics.

Bioinformatics is a field that is constantly redefining itself. Some problems are clearly defined, such as the alignment of two proteins using the Smith-Waterman algorithm (Smith and Waterman 1981). However, many other issues are moving targets. There is also the constant pressure to produce results quickly. Traditionally this has caused a polarization between the development of handcrafted applications in languages such as C for specific tasks like the BLAST applications (Altschul, Gish et al. 1990), and the use of rapidly developed 'throwaway scripts' in scripting languages such as Perl[18] and Python[19]. In practice, 'throwaway scripts' often become the basis of sequence analysis pipelines that have a lifetime of months or years, and are maintained by a succession of individuals. Eventually, these must be re-written to improve performance, to fix bugs inherent in the initial design, or to allow the application to perform tasks that were not part of the original design aims.

Java is a suitable language for rapidly developing Bioinformatics applications. It can be used to write the computationally expensive as well as the flow-control portions of Bioinformatics scripts. If libraries of biological functionality are developed, and these are easy to use and extend, then it becomes possible to achieve rapid development of throwaway scripts. If these short-term applications become part

[18] The Perl web site can be found at http://www.perl.org/

[19] Python is distributed from the http://www.python.org web site

of pipelines, the object-oriented nature of Java code means that it is potentially possible to salvage much of the intellectually expensive code, and to quickly isolate design faults.

The Java compilers are much more pedantic than `C` or `C++` compilers, disallowing many unsafe constructs that can generate strange runtime behaviour. For example, casts are checked where possible, and arbitrary pointers do not exist. Memory allocation and de-allocation are handled by the VM. This means that many errors that would show up as a program crash in other languages cause the Java compiler to generate error messages.

BioJava is intended to provide the functionality needed to rapidly develop effective Java applications for bioinformatics. The design of the language, compiler and virtual machine help greatly in quickly developing robust applications. BioJava builds upon this strong foundation by providing APIs for common biological objects and tasks, such as biological sequences, and reading these from files. Additionally, a number of classes provide basic functionality that increases both the encapsulation and the robustness of BioJava's highly polymorphic code. The rest of this chapter describes the core classes and interfaces that provide this functionality, and for which I was solely or primarily responsible for the design and implementation.

The conventions adopted here for referring to Java types and methods are those used in the Java documentation. When referring directly to types and methods, the type used is `fixed width`. When methods are referred to, the usual form will be to name the method followed by ellipses as in `someMethod()`, regardless of the actual arguments accepted by the method. If it is necessary to describe the parameters accepted by a method, either for the clarity of the text, or to disambiguate over-loaded methods, the types of the arguments will be included as in `anotherMethod(String, int)`. In a few very rare cases, the names of the arguments must be included as in `substring(int start, int length)` so as to make the semantics more clear. In general, once a method is introduced, it will be referred to using the shortest unambiguous form.

2.2 Nested Exceptions and Assertions

Both during the development of applications and their deployment, failures occur. These may be due to the application being implemented incorrectly, being used with data that it was never designed to be used with, or by some external failure, such as a break in network communication. Programmatically handling failures gracefully and informatively is a key to developing robust software rapidly.

Java supports the handling of error conditions by the throwing of exceptions. The built in exceptions have a constructor that takes a message `String` only. Java methods can be defined as throwing a list of `Exception` types. This means that the method can raise any one of these exceptions if it is unable to complete processing, and that it is limited to this list of checked exceptions. Some exceptions, such as `OutOfMemoryException` are unchecked as it would be difficult to guard against the many places where they may be raised without bloating both the volume of source code and impacting upon run-time performance. During invocation, a method may choose to not raise any of the listed exceptions (indicating that it was successful). Function calls in `C` generally return an error code to indicate error status. In Java, the method would return a value if it was successful or throw an exception if it was unable to complete. The basic exceptions are applicable to the case where the error is caused by a failure in the program that is clearly attributable to a single action, such as accessing a file, or an array index being out of bounds.

When complex applications are composed of multiple ‘black box’ modules, failures in one module may cause failures in another module. With classical exceptions, the original cause would either percolate up by allowing the `Exception` to be thrown from all methods in the first module that invoke methods in the second one, or would have to be caught, and a new `Exception` thrown to describe the failure. The first alternative tends to lead to methods throwing very large numbers of specific exceptions defined in other modules or it leads to methods throwing extremely general exceptions that provide poor programmatic control over error handling. This problem becomes even more pronounced when there are multiple implementations of a given interface. The interface author can not possibly foresee all of the ways the interface may be implemented or the range of potential failures, so can not declare all of the exceptions that may be raised by all implementations. For example, if an interface is implemented using a Common Object Request Broker Architecture[20] (CORBA) peer in one case and file access in another, the implementations may fail due to CORBA-specific exceptions or problems with file access, but the original interface author could not have known this, so would not have listed exceptions specific to these two failures in the methods.

BioJava provides subclasses of `Exception` and `Error` (the base-class for unchecked exceptions) that have an extra field that contains a reference to a causal exception. These are called `NestedException` and `NestedError` respectively. In the above

[20] http://www.omg.org/ is the web-site for the organisation that manages the CORBA standards

example, the interface author would declare the methods as throwing sub-classes of `NestedException` describing the type of the failure (something like `RetrievalFailedException`). The CORBA implementation would catch the CORBA-specific exceptions and then construct and throw a new `RetrievalFailedException` instance that refers to the CORBA exception. Similarly, the file-based implementation would catch `IOException` instances and throw new `RetrievalFailedException` instances that refer to the `IOException` that caused the failure. `NestedException` and `NestedError` can be nested to arbitrary depths, allowing a complete 'chain of evidence' to be collected about the cause of errors without requiring modules to have knowledge of all of the exceptions raised by indirect dependencies.

This ability to have both a complete chain of evidence for any failure while respecting encapsulation has made it much easier to develop portions of the BioJava library independently of one another while still allowing them to rely on functionality provided by other BioJava modules. In our view, this has strongly contributed to the rapid development of the libraries.

Because of a combination of Java not supporting sub-classing by restriction, and the Java compiler being pedantic about ensuring that each error condition is accounted for in the code, exceptions must be caught even when they are logically impossible to generate. For example, if counts are being collected for a probability distribution over the DNA alphabet, and the sequence is known to be DNA, then it should be impossible to raise an `IllegalSymbolException`. However, because the component-based APIs don't have this information, the compiler will expect the exception to be handled. The recommended way to handle this is to catch the exception and throw a

`NestedError` instance indicating that an assertion has been violated. The `NestedError` instance will cause the stack to unwind until it is explicitly caught. If not caught, the thread will exit with an error message. In theory, this case should never happen, but in practice, incorrectly implemented objects manage to invalidate these checks, particularly during development, and the assertion failures clearly pinpoint the source of the errors.

Nested errors and exceptions are used throughout the BioJava libraries and in many of the applications that use these libraries. They have proven to be invaluable in writing robust code. Sun has added the concept of nested exceptions to the latest version of Java (1.4), and we look forward to merging our system with theirs.

2.3 Changeability

When developing complex applications, and in particular those which may contain multiple threads of execution, it is important to control which resources may change and which can not. Often, it is also very important to be informed if something does change so that some action can be taken. Unfortunately, the exact details may need to be decided at run-time, so can not be implemented at compile-time as a mutable or immutable interface implementation or by using keywords (such as `C`'s `const` modifiers). Robust applications that are built in a modular manner need strong guarantees about which resources will and won't be modified, and expect these to be enforced.

An object may change state because a method is invoked that would directly modify it. Alternatively, it may change state because some other object, that it delegates state maintenance to, is modified. For example, a `List` instance may be modified by invoking the add method to append an item to the list. A view on the list returned by

`Collections.unmodifiableList(List)` cannot be modified directly, but if the underlying list is altered, then the unmodifiable view will reflect this change. This illustrates the difference between modifiability and changeability. The unmodifiable list is changeable, as there is a legal way for its data to be altered, even though it can't be modified directly.

BioJava contains a complete object model for tracking the changes made to objects, and for allowing changes to be prevented, without breaking object encapsulation. The `Changeable` interface defines methods to add and remove listeners which will be informed when the state of an object alters. These listeners are informed before the object attempts to change, and have an option to veto the change by throwing a `ChangeVetoException`. If none of the listeners throws an exception, then the object updates its state and then informs each listener of the change. At this stage, each listener can synchronize state to ensure data-integrity. In principal, this is a very similar design pattern used by Java Beans to implement 'bound properties', but offers several advantages akin to those provided by a simple cascading transaction processing framework.

Each BioJava interface that extends the `Changeable` interface has public final static fields that hold `ChangeType` instances that encapsulate one way that the Interface implementations may change state. For example, `FiniteAlphabet` defines a static field called SYMBOLS that represents a modification that changes which Symbol instances are contained within the alphabet. Each implementation of `FiniteAlphabet` is required by the Changeability API to allow `ChangeListener` instances to be registered. The `ChangeType` class supports the idea of a hierarchy of change types.

The root of this hierarchy is UNKNOWN. When listeners are added to a changeable, they will be informed of all events that descend from that type.

It is required by the contract described in the Changeable documentation to inform all listeners registered for a given type and all of its descendents whenever state changes by invoking the `preChange()` method on each listener before it changes state If any one of the listeners vetoes the change, or if for any other reason the state cannot be updated, then an exception will be raised. If the change can go ahead, it will commit the state change and then invoke the `postChange()` method on listeners to inform them that the state has been successfully modified. There is no guarantee made about the relative orders that different listeners will be informed either before or after a change is made. It is legal to inform different listeners in different threads, and in some situations this may be a sensible thing to do, if for instance, the listeners need to communicate with network resources.

The `preChange()` method indicates that a change should not be made by raising a `ChangeVetoException`. `ChangeVetoException` extends `NestedException`, allowing a change to be prevented because of a failure elsewhere in the application to publish this information. A common cause for this is when an object stores its state in a delegate. If a method is invoked on the object that would modify the delegate, and the change on the delegate is vetoed then the delegate will raise a `ChangeVetoException`. This will be caught by the object and a new `ChangeVetoException` will be thrown indicating that the requested modification could not be made. This new exception will nest the original exception, allowing the complete reason for the failure to be maintained. One very useful `ChangeListener` is the AlwaysVeto instance. This always throws a `ChangeVetoException` in `preChange()`, ensuring that the change cannot go ahead. In this way, a mutable implementation of an interface can be instantiated, populated and then locked. From that moment on, no more modifications can be made to it.

The way that the Changeability API can be used to dynamically constrain what data can change, and what data is fixed, can be illustrated by the dynamic programming code. While an HMM is being used for an alignment, `AlwaysVeto` is registered as a listener to all of its parameters, preventing them from changing. Once the alignment has been completed, the `AlwaysVeto` listener is removed, allowing it to be modified again. This kind of fine-grained and dynamic control of which properties can be modified is key to developing robust and modular BioJava functionality.

`ChangeEvent` contains a field to store another `ChangeEvent`. This is used in the case when a change in one object leads to a change in another. For example, if a probability distribution that encapsulates some transition probabilities in an HMM changes then the HMM will no longer have the same parameters. The HMM will listen to each distribution, waiting for them to change. Whenever they do, it will inform listeners that the model parameters have changed with a `ChangeEvent` instance that refers back to the event fired by the probability distribution. Again, by maintaining references back to the event that caused the new event to be fired, a complete ‘chain of evidence’ can be built up about why an object wishes to change, without invalidating the objects encapsulation. This potentially allows listeners to accept or reject a change according to one of the underlying causes.

To avoid the expense of maintaining the entire changeability infrastructure all of the time, several BioJava objects only build the support objects once listeners have been registered. In this way, the cost of having changeability support in an object with no listeners is the cost of one field in the object that has a null value. Once listeners are added, this field would be filled with a reference to the support data-structures. Mutator methods can easily chose not to perform expensive operations while there are no listeners, such as protecting the listener list with a synchronized block. In addition, if some listeners have been registered, but none have been added that need to be informed of changes to state-delegation objects, then there is no need to instantiate the change-forwarding apparatus. By implementing the classes that will take part in a network of Changeable objects carefully as described here, it is possible to avoid almost all unnecessary overhead.

The Changeability API is used extensively throughout the BioJava libraries. It has proven to be effective at guarding against design flaws and has prevented countless bugs that could have been caused by assuming the involatility of data. It has also been leveraged within the DAS[21] (Dowell, Jokerst et al. 2001) and GUI packages to implement efficient data-caching schemes. In the future, it may be necessary to implement a full transaction-processing framework. Until then, the Changeability API will continue to be an invaluable tool.

2.4 Symbols, Alphabets and SymbolList

Although BioJava must deal with DNA and protein sequences, the underlying interfaces for defining sequences is extremely flexible, allowing almost any signal to be represented as a stream of symbols. This allows all code defined in terms of these APIs to be applied to a wide range of use-cases. For example, the FASTA file format object can be used to read and write state labels from HMMs without any change,

[21] The DAS standard, and associated information is published at the http://biodas.org/ web site

simply by parameterising it with the alphabet of states for that HMM, and the sequence viewing APIs can be used ‘out of the box’ with this data. This section describes these APIs and how they can be used to represent sequences of complex data structures in addition to DNA.

The interfaces borrow heavily from the concepts of an entity, a set and a string. A set is an item that contains some number (possibly zero) of entities. A string is an item that can be represented as an ordered list of entities (possibly zero in length). If all of the entities in a string belong to a particular set, then it is described as a string over that set. For Java `String` objects, the entities are `char` instances, the set is the Unicode Character Set and the string is the `String` class. In BioJava, the `Symbol` interface represents an individual entity, `Alphabet` represents a set of `Symbols` and a `SymbolList` represents a string over an `Alphabet`. These interfaces are designed to be as mathematically elegant as possible, as over time this has made them very useful for seamlessly implementing algorithms. This has had the unfortunate side effect that a large amount of documentation is needed to describe what all of the API does. Fortunately, users of these APIs usually do not need to know the finer details to perform all common tasks.

To reduce the amount of special-case code required, the Symbol interface represents ambiguity symbols, such as ‘n’ or ‘x’ and gaps ‘-’ as well as concrete symbols such as the nucleotides ‘a’, ‘g’, ‘c’ and ‘t’. Sometimes the natural alphabet to work in can be represented as the cross product of other alphabets. When writing code to translate a region of RNA, it is convenient for the code to work with RNA triplets. The natural alphabet for this is RNAxRNAxRNA, which contains symbols that represent entities like [a, u, g] and [c, c, c]. To allow all of this to be represented consistently, three symbol interfaces extend one another.

`Symbol` is the most generic interface. `Alphabet` and `SymbolList` are defined in terms of in terms of this interface. `Symbol` has two methods. A textual representation is provided by `getName()`, which returns a human-readable string like 'Adenine' or 'gly'. The `getMatches()`returns an `Alphabet` that contains all of the `Symbols` that are valid matches to this one. The matches `Alphabet` will by definition contain the symbol itself, as it must match itself. For a `Symbol` that is ambiguous, such as 'n', this alphabet will contain multiple items. For a `Symbol` that has no ambiguity, such as 'a', this will return an `Alphabet` containing just that single `Symbols`. Gaps are represented by symbols that return an empty alphabet for `getMatches()`, representing the idea that there is literally nothing there, even though space must be reserved for it. Two `Symbol` instances are considered equivalent if their `getMatches()` alphabets contain exactly the same set of `Symbols`. An `Alphabet` does not contain a `Symbol` if there are any members of `getMatches()` that are not also members of the `Alphabet`.

`BasisSymbol` extends `Symbol` and adds the method `getSymbols()` that returns a `List` of `Symbol` instances. Any column of any alignment can be represented as `BasisSymbol` instance, as it is a list of individual symbols, one from each sequence in the alignment. If a `Symbol` comes from some `Alphabet` 'a' that can be represented as the cross product of a list of alphabets, 'A', then it is a `BasisSymbol` if it can itself be represented as a list of `BasisSymbols`. All one-dimensional `Symbol`s are `BasisSymbols`, as clearly an alignment of a single sequence contains columns with single symbols in it.

The codon [a, a, t] is a `BasisSymbol` because it is represented by the list 'a', 'a' and 't' from the DNAxDNAxDNA alphabet. The codons {[a, a, t], [a, c, t]} can be represented as [a, {a, c}, t], which again is a list of three symbols. The codon {[a, a, t], [a, c, t], [a, a, g]} can not be represented as any single list of symbols, so it is not a `BasisSymbol`. However, {[a, a, t], [a, c, t], [a, a, g], [a, c, g]} can be represented as [a, {a, c}, {t, g}], so is a `BasisSymbol`. An `Alphabet` does not contain a `BasisSymbol` if there is any member of `getMatches()` that it does not contain. Additionally, an `Alphabet` does not contain a `BasisSymbol` if it is of a different order to the `Alphabet` (for example, the basis symbol is of length 3, but the alphabet is the product of two other alphabets).

`AtomicSymbol` extends `BasisSymbol` but adds no methods. These symbols actually make up an alphabet, and are never ambiguous. There is an `AtomicSymbol` for 'a' or the codon [a, t, c]. Since `AtomicSymbol` instances can't be ambiguous, `AtomicSymbol` adds the constraints that `getSymbols()` must return a `List` that only contains `AtomicSymbol`s. For the same reason they also add the constraint that `getSymbols()` must return an `Alphabet` that contains exactly one `Symbol`, and that should be the instance itself. Two `AtomicSymbol` instances are considered to be equal if they are referred to by the same Java reference, that is, they are comparable by the == operator. This constraint is not mathematically required, but is necessary to implement efficient algorithms. A Java reference is of the same size as a pointer, and the == operator will have the same overhead as pointer comparison. On many architectures, this will be as efficient as comparing integers or characters.

In practice, most user code never need know that `Symbol` instances can be cast to `BasisSymbol` or `AtomicSymbol` because the API is complete enough that any APIs can be defined to work with `Symbol` directly and hide any casting inside library code methods. For example, the `TranslationTable` interface defines a method to translate a `Symbol` from one `Alphabet` into a `Symbol` in another `Alphabet` (representing the concept of a function with the domain being the first `Alphabet`, and the codomain being a subset of the second `Alphabet`). This can be implemented by maintaining a table for each `AtomicSymbol` in the source alphabet and the associated `AtomicSymbol` in the target alphabet. Given any `Symbol`, it would first check if it was castable to `AtomicSymbol`. If it is, then the return value can be found directly by looking it up in the table. If it is not, then each `AtomicSymbol` instance in the `getMatches()` alphabet can be translated in turn and a new ambiguous symbol can be made representing this set of translated symbols. In either case, the code calling the translate method need not know anything about the actual type or implementation of the symbol instance.

The gap symbol needs special treatment to avoid various logical problems. The purest version of gap would represent a perfectly empty set, which is dimensionless. Indeed, the EMPTY_ALPHABET constant contains just this entity. The pure gap is a `Symbol` that returns EMPTY_ALPHABET in response to `getMatches()`. All `Alphabet` instances contain this gap. This is because there is no `AtomicSymbol` instance that matches the gap, so there can never be one that matches the gap that is rejected by any `Alphabet` instance.

In addition to the pure gap, there are gap symbols that represent columns in alignments that contain gaps themselves. We can refer to a `BasisSymbol` that is a list containing as [gap]. The gap and [gap] symbols are distinct entities. The pure gap takes up no space in any direction. The [gap] symbol takes up space in one direction. Alphabets like DNAxDNA would contain both gap and [gap, gap], but it would not contain [gap], as [gap] is a 1-dimensional `BasisSymbol`, and DNAxDNA is two-dimensional. Using this notation, we could represent a column in an alignment between two sequences with a gap in the first sequence as [gap, sym]. Symbols like this have no AtomicSymbol instances that could possibly match them, so their `getMatches()` alphabet is empty. In geometrical terms, this is similar to finding the volume of a solid that has one dimension of size zero. If there is some alphabet that is the cross product of other alphabets, some of which are themselves cross products of other alphabets, the gap symbol respects this. For example, the alphabet DNAx(DNAxDNA) would have the gap symbol [gap,[gap,gap]]. Although this looks complicated, it is in fact necessary to correctly maintain all available information about a `Symbol`. It allows algorithms such as the dynamic programming recursions to distinguish between insertions in each sequence, cells that are at the start or end of one sequence, and cells that lie outside the range of the sequences.

The `Alphabet` interface represents a set of `Symbols`, and can therefore be uniquely represented as a set of `AtomicSymbol` instances. It follows that any ambiguity symbol is a member of an alphabet if its `getMatches()` `Alphabet` is a subset of the alphabet. The `contains(Symbol)` method returns true if the argument is a member of the `Alphabet`, and false otherwise. As a convenience to code that uses `Alphabet`, it also has the method `validate(Symbol)` that throws an `IllegalSymbolException` if the symbol is not contained in the alphabet, and silently returns otherwise. This is in concept similar check to a run-time class cast check.

`Alphabets` have a name retrieved by `getName()`. This is intended to be human-readable. It is also used as a unique identifier for the alphabet, allowing alphabets to be serialized between different virtual machines and resolve to a single unique instance.

To convert from text to `Symbol` instances, an `Alphabet` can provide access to multiple `SymbolTokenization` instances via the `getTokenization(String)` method. The tokenization registered under the string ‘token’ will allow `String`s to be parsed into `Symbol`s using some well-known single character codes. Alphabets may provide other tokenizations.

When the product is taken of a `List` of `Alphabet` instances, `getAlphabets()` for the resulting `Alphabet` will returns an equivalent `List`. It follows that `getAlphabets()` returns a `List` of `Alphabet`s that when multiplied together in that order would generate that `Alphabet`. The EMPTY_ALPHABET constant is equivalent to the product of a zero length list. One-dimensional `Alphabet`s such as DNA, return a single element `List` containing themselves. The `Alphabet` DNAxDNAxDNAxPROTEIN would return the list [DNA,DNA,DNA,PROTEIN], and so on.

Two factory methods allow `Symbol` instances to be retrieved from an `Alphabet` while allowing it to maintain internal state and manage memory efficiently. The `getAmbiguity()` method takes a `Set` of `Symbol` instances. It returns a `Symbol` instance that has a `getMatches()` value that contains all of the `AtomicSymbol` instances matching any one of the `Symbol` instances in the `Set`. The `getSymbol()` method takes a `List` of symbols and returns a single `Symbol` that represents the product of these. Both methods validate the input collections to ensure that the resulting `Symbol` is a legal member of the `Alphabet`.

These methods could potentially need to do some fairly involved processing. They must make sure they return the most specific type of symbol possible. If an equivalent symbol exists in the virtual machine, it should return that instance. The `AlphabetManger` class provides several methods to help in this process, simplifying the implementation of `Alphabet`.

`FiniteAlphabet` extends `Alphabet` and represents the case when there are a finite number of `AtomicSymbol` instances that are contained in the `Alphabet`. Because the set is now finite, we can meaningfully define some more methods. The number of `AtomicSymbol` instances in the `Alphabet` is returned by `size()`, and `iterator()` returns an `Iterator` over these. Additionally, it adds the mutator methods `addSymbol()` and `removeSymbol()`, which allow the set of symbols contained to be altered.

Alphabets such as DNA and PROTEIN are represented by instances of `FiniteAlphabet`. There are non-finite `Alphabet` instances for things like the set of all integers and doubles. The `Alphabet` representing all `Symbol`s for integers between 1 and 100 will be a `FiniteAlphabet`. Otherwise, they are non-finite, and just implement `Alphabet`. There are a range of package-private implementations of `Alphabet` and `FiniteAlphabet` that implement these rules. `AlphabetManager` provides the main API for manipulating these entities.

`SymbolList` represents a list of `Symbol` instances from a single `Alphabet`. The `getAlphabet()` method returns the `Alphabet` it is over. The `symbolAt(int)` method returns the `Symbol` at that index. The length of the `SymbolList` is returned by `length()`. Arguments to `symbolAt()` must be between 1 and `length()` inclusive. A portion of the `SymbolList` can be retrieved by invoking `subList(int start, int` `end)`, where start and end must also be legal indexes. The sub-list returned is inclusive of both start and end.

The default implementations of `SymbolList` in BioJava over finite alphabets all use the ‘flyweight’ design pattern (Gamma, Helm et al. 1994)[195] to keep memory consumption to a minimum. Internally, the `SymbolList` maintains references to the small number of `Symbol` instances in their `Alphabet`. This means that in a `SymbolList` that is a million DNA symbols in length will be represented as a list of one million references to the four DNA `AtomicSymbol` instances in the DNA `Alphabet`. Because the `SymbolList` interface places no requirements on the actual storage of the data, it is possible to implement many different storage mechanisms. For example there are implementations that fetch portions of the underlying data from files or databases on demand.

Alphabets for DNA, RNA and protein are in use daily with the BioJava toolkit. Additionally, subsets of the alphabet of all double values are used to represent DNA physical properties (such as curvature or flexibility), and higher order alphabets are routinely used to encapsulate everything from multiple-sequence alignments to the results of alignment algorithms to 3-D coordinates. The apparent complexity of the underlying symbol model has more than paid for itself by the vast increase of potential applications now available to objects and algorithms which purely rely on these interfaces, not the underlying data.

2.5 Locations, Sequences and Features

The `Symbol`, `Alphabet` and `SymbolList` APIs were primarily designed to be a good basis for developing algorithms. In contrast, `Sequence` and `Feature` are designed for representing bioinformatics concepts such as database IDs, repeat regions and exons.

`Sequence` represents an entire biological sequence, be it a chromosome, a clone or a primer. A `Feature` represents a region of a `Sequence` that is annotated as being interested for some reason. It may, for example, be a repeat, an exon or a protein active site. The position of a `Feature` is specified by a `Location` object.

The `Location` interface represents an immutable set of indices. A `Location` may be a single index (such as 73), or the range of indices (like [1000..1100]), or all indices that are odd, or any other arbitrary set (for example, {73, [1000..1100]}). `Location` defines the methods `getMin()` and `getMax()` to return the lowest and highest index contained within that `Location`. The method `contains(int)` indicates whether an index is contained. For all locations other than EMTPY_LOCATION, both `getMin()` and `getMax()` are contained by the `Location`.

There are specific implementations for special cases, such as `PointLocation` for a single index and `RangeLocation` for a contiguous range, and `CompoundLocation` for dis-continuous regions. The `equals()` method will return true if two instances contain exactly the same set of indices, regardless of the concrete class of the instances.

The methods `isContiguous()` and `blockIterator()` work together to expose the state of the `Location` without exposing the storage. The `isContiguous()` method returns true if there are no indices above `getMin()` and below `getMax()` that are not contained. The `blockIterator()` method returns an `Iterator` over a minimal set of `Locations` that are guaranteed between them to contain each index exactly once, and are themselves contiguous. For a contiguous `Location`, this will return an `Iterator` that just returns that instance.

There are several methods that compute new `Locations` from old ones; `translate(int dist)`, `intersections(Location l)` and `union(Location)` perform the obvious functions. The methods `overlaps(Location)` and `contains(Location)` return true or false depending on whether the argument overlaps or is entirely contained within the `Location` respectively. Using these operations, it is possible to build arbitrary `Location` instances without ever needing to know how a `Location` is implemented. This makes code using the `Location` APIs very easy to maintain, while enforces ridged encapsulation.

`Locations` are used within the context of the APIs described here. They have, however, been found useful for a range of other applications including bookkeeping to store which pixel indices have been used in GUIs, and also for prime-number searching algorithms.

The `FeatureHolder` interface represents a collection of `Feature` instances. It has methods to count how many features it contains, return an iterator over them, and to return a `FeatureHolder` containing all the Features that match a filter criterion.

There are implementations of `FeatureHolder` that directly store features. Other implementations encapsulate views of over `FeatureHolders` (for example, performing a translation and strand-flip operation) or just store the rule necessary to fetch the underlying data when needed (quite common when implementing adapters to high-latency storage, such as databases).

The `Annotatable` interface specifies one method, `getAnnotation()`, which returns an `Annotation` object. The `Annotation` object is just an associative array (key to value mapping) where arbitrary information can be stored.

`Sequence` extends `FeatureHolder`, making it a container of features. It also extends the `SymbolList` interface so that it can represent the primary sequence. In addition, it adds a name and URI property for naming the sequence uniquely and also extends `Annotatable` so that arbitrary information that pertains to the entire sequence can be stored.

There are implementations of `Sequence` that store the sequence and features in memory. Other implementations include those that manipulate whole-chromosome assemblies or data from Ensembl (Hubbard, Barker et al. 2002) and DAS. The interface-centric design means that an implementation of `Sequence` that suits a particular situation can usually be trivially composed from a suitable implementation of `SymbolList`, `Annotation` and `FeatureHolder`. As Java does not support multiple inheritance of implementations, this is achieved by storing references to objects implementing each of these interfaces and explicitly forwarding method invocations as needed.

Other than the effort required to initially enter the code that forwards method calls, this actually has some benefits over inheritance. Firstly, the implementing objects may in some cases be expensive to initialize or use a lot of memory. As they are private state of, and not directly part of (by inheritance) the implementation, they can be lazily instantiated. Secondly, it is possible to choose a specific implementation class at run-time, for example, by choosing an implementation of `Annotation` optimized for efficiently storing very small numbers of values, or for retrieving values associated with a very large number of properties.

`Feature` extends `FeatureHolder` and `Annotatable`. In addition, `Feature` has source, type, location, parent sequence and symbol properties. Source and Type are equivalent to the GFF source and type fields. Source should represent the program or process that produced the evidence for the feature (such as a particular gene finder), and Type should indicate what the feature is meant to represent, such as CDS, or transcription start site. The `Location` represents which region of the `Sequence` the `Feature` is attached to. The parent is the `FeatureHolder` that directly contains this `Feature`. The Sequence property always refers to the `Sequence` that ultimately contains the `Feature`. As `Feature` extends `FeatureHolder`, it is possible to build up arbitrary hierarchies of features (but always as a tree). For example, a gene feature may contain zero or more child features that represent transcription factor binding sites or perhaps exons. The parent of the exon would be the gene feature, and the parent of the gene would be the sequence. However, both exon and gene objects will return the same `Sequence` for `getSequence()`. Lastly, the symbols property returns a `SymbolList` that represents the `Symbols` contained within the `Feature`. The exact semantics of this method is left up to the `Feature` implementation. Sub-interfaces of `Feature` are provided which add more specific properties such as strand information, frame and protein translations.

So that the feature hierarchy on sequences can be modified, there must be an API for adding features to other features and to sequences. So that a given implementation of `Sequence` and `Feature` can maintain implementation integrity, there must be some sort of factory method (Gamma, Helm et al. 1994)[107] defined in the interfaces. The original implementations had methods like `createFeature()`, `createStrandedFeature()` or `createExon()`, but as the number of interfaces grew, it became obvious that this would not scale well as the interface would have to be modified every time another type of feature was added.

This was solved by using a single `createFeature()` method that takes a polymorphic argument of type `Feataure.Template`. The template has public fields that hold the properties of the feature to make, such as location, type, source and the like. This mirrors the memento design pattern (Gamma, Helm et al. 1994)[283]. In each interface that extends `Feature`, a public static inner class extends `Feature.Template` called by convention Template. For example, to instantiate a `StrandedFeature`, you invoke the `createFeature()` method with an instance of `StrandedFeature.Template` as the only argument. It is then the responsibility of the `Sequence` and `Feature` implementation to create a `StrandedFeature` implementation with the same information as the template. If the particular `Sequence` implementation can't provide an appropriate implementation of `Feature`, it should either instantiate the closest one it can and put the missing information into the annotation bundle, or throw an exception. This approach allows sequence implementations to support an arbitrary sub-set of the available feature types without requiring the feature creation interface to grow in complexity with the number of feature types defined.

Sequences and features represent the bulk of information manipulated by most applications. By designing the APIs from the foundation to support polymorphism and encapsulation, we have produced a design that allows the underlying data to be represented in any one of a number of different ways. There are implementations that are backed by relational databases (Ensembl and BioSQL adaptors), CORBA (BioCorba adaptors) and by web services (DAS and XEMBL clients) in addition to those using Java objects directly. The feature creation API provides a uniform and easy to implement way to create features conforming to a range of interfaces with a range of different concrete implementations. The result is that developers can interact with a single API and have access to a wide variety of different information without needing to know anything about the implementation details of how this is achieved.

2.6 Probability Distributions and Hidden Markov Models

Hidden Markov Models (HMMs) (see Section 1.3.2) are a popular method of analysing biological sequences. BioJava contains APIs for representing and working with probabilistic HMMs. This includes code for representing models, as well as implementations of the common dynamic-programming (DP) algorithms for evaluating alternative state-paths and training model parameters. All of these APIs build upon the `Symbol`, `Alphabet` and `SymbolList` APIs (Section 2.4), allowing them to be applied without change to the wide range of signal types these data-structures can be used to represent.

To model HMMs effectively, it is useful to provide a mechanism for representing probability distributions. There are a wide range of other contexts within which probability distributions can be used, such as in modelling weight matrix columns, so to aid in their reuse there is a separate Java package dedicated to their representation and implementation.

The `Distribution` interface encapsulates a probability distribution over an `Alphabet`. The method `getWeight(Symbol)` returns the current probability of observing the symbol from the probability distribution. This is notionally equivalent to integrating or summing a probability distribution out over the range of all `AtomicSymbol` instances in the symbol's `getMatches()` `Alphabet`. The manufacture of distributions is usually performed by a `DistributionFactory` object. This allows particular implementations of `Distribution` to be tailored to a particular `Alphabet` without client code needing to know the details. For example, the default implementation of the factory returns `Distribution` implementations that use either linear lookup or binary search lookups based upon which is most time-efficient for the alphabet size. In addition, `OrderNDistribution` extends the Distribution interface, and defines that it will be a probability distribution over one alphabet conditional upon another. For example, given the `Alphabet` DNAxDNA (containing all ordered pairs of nucleotides), an `OrderNDistribution` could be built that was four independent `Distribution`s over the second `Alphabet` conditioned upon the first (i.e. the probability of the second nucleotide appearing given that we knew what the first one was).

Background database probabilities of the amino acids in Swiss-Prot (Boeckmann, Bairoch et al. 2003) could be represented as a probability distribution over the PROTEIN alphabet. The probabilities could be estimated by counting the frequencies of each amino acid in the database and then normalizing these counts to give a probability distribution.

The `MarkovModel` interface encapsulates state-emitting HMMs. `MarkovModel`s contain one or more `State` objects. The `State` interface extends `AtomicSymbol`. `EmissionState` specializes `State` by having an associated emission `Distribution`. In generative model terms, `EmissionState`s emit the symbols within sequences. `DotState` extends `State`, and represent non-emitting, silent states. These are useful for rationalising the architecture of models.

The `MarkovModel` interface has a `stateAlphabet()` property that returns a `FiniteAlphabet` containing every `State` in the model. It also has an `emissionAlphabet()` property that is the `Alphabet` that matches the `Alphabet` of the `Distribution` objects associated with the `EmissionState` instances. In addition, there is a method `getWeights(State)` that returns the transition probabilities from a `State` as a `Distribution`. The `Alphabet` of the `Distribution` will be a sub-set of the states `Alphabet`, representing every `State` that is reachable from that `State`.

`MarkovModel` also has a property `getHeads()` that represents how many `SymbolList` instances are aligned to each other and the model. A single-head model emits a single `SymbolList` of sequence and a single `SymbolList` of `State`s. These co-linear lists of symbols and states can be used to label a sequence, which may, for example, mark up features like repeat regions, protein domains or exon boundaries. Models with two heads perform pair-wise alignment. These can be used to align pairs of sequences based upon evolutionary relationships, or to find portions of two sequences that are more similar than would be expected by chance. Models with three or more heads align that number of `SymbolList` instances to one another and label the alignment with the states used.

The `EmissionState` interface defines one other property named advance, which is an array of integers (usually 0 or 1) that indicate how much each head of the model is advanced by the emission. For example, in pairwise alignments, the states that emit aligned regions will advance in both directions, having an advance property of [1, 1], where as the insert states will have an advance of [1, 0] and the delete states will have an advance of [0, 1]. The `Distributions` associated with emission states should emit gap `BasisSymbols` that have a `BasisSymbol` in each dimension that is 1, and a gap in each dimension that is 0. Gaps are used to represent the concept that there is a gap in the list of symbols for that dimension, or equivalently that although the global index has advanced, the index of the underlying data being viewed has not.

The `MarkovModel` interface is the data-structure that defines how some sequences could be emitted. These HMMs are purely data, and have no algorithms associated with them. The recursions that align sequences given a `MarkovModel` are defined by the `DP` interface. `DP` defines methods to calculate the forwards, backwards and Viterbi recursions (see Equation 1-13). In addition, it defines a method to generate sequences from the model. The efficient implementation of these recursions depends upon the structure of the model and the number of heads the model has.

To hide implementation detail from the user, the `DPFactory` interface defines how to get a `DP` implementation for a given model. For models with one or two heads, there is a `DP` factory implementation that returns `DP` implementations that invoke interpreters. For two head models, there is also a `DP` implementation that generates Java byte code optimized to the architecture of a particular model. The `DP` compiler outperforms the `DP` interpreter significantly, particularly for models that contain many states that have transitions from only one source state. The interpreter is more suited to situations where the model architecture is being altered, as the compiler would have to produce new code each time the model architecture is modified.

For pairwise alignment, using the notation of Equation 1-13, $\bar{i}$ is a 2-tuple of the index for the first and second sequences respectively. We can impose a partial ordering upon the set of 2-tuples that follows the natural ordering of each component. To calculate the cell at $\bar{i}$, we must first have calculated all cells that are before $\bar{i}$. For the 2-dimensional case, this means calculating the cells at $\bar{i}-(1,0)$, $\bar{i}-(0,1)$ and $\bar{i}-(1,1)$. The naive way to ensure this is to construct an in-memory matrix to store results, and to perform a nested loop over two index variables starting at 0 and going to the first and second sequence length respectively. There are many ways to loop over all possible values of $\bar{i}$ while guaranteeing this ordering. As long as all of the values that are needed for uncalculated cells to be calculated are present, it is not necessary to store any of the other values.

Assuming that the entire dynamic programming matrix is not required, we can write a space-optimised implementation of pairwise DP that uses space proportional to the length of the shortest sequence. Given an index a into the first sequence (the shortest) and b into the second (the longest), we can have an outer loop over the values of b. A single row of the dynamic programming matrix (indexed by a) containing the results of the previous iteration (at $b-1$). Then, a new row can be calculated for the row at b. At the end of the iteration, the column at $b-1$ and be discarded, and that at b becomes the array used as the known results for the next iteration (at $b+1$).

The back pointer structure is a matrix of the same shape as the Viterbi matrix, but storing the state used to reach that cell. This contains all the information necessary to trace back from the final cell to the beginning of the alignment to retrieve the highest scoring alignment. However, this introduces a space cost proportional to the product of the sequence lengths. It is clear that many sub-optimal paths will exist through the matrixes that are not needed for the eventual trace back.

Instead of holding a reference to the previous state in the matrix, BioJava stores a `BackPointer` object. This stores a reference to the previous `BackPointer`, a step-wise score and reference to the `State` associated with that position. Because `BackPointer` instances refer directly to the previous entry in the chain, there is no need to store the entire matrix in memory explicitly. The Java virtual machine will take care of garbage-collecting all `BackPointer` instances that are not reachable from the current states. It is then only necessary to explicitly hold in memory the

`BackPointer` objects associated with the cells that still have uncalculated dependencies.

In this way, the memory cost for the back pointer data-structures can be reduced to something that is at a maximum proportional to the memory used to store the values of the dynamic programming matrix, and which converges to something proportional to the length of the final alignment (which can never be longer than the sum of the lengths of the sequences). The `BackPointers` will form a directed a-cyclic graph. The trace back path must be from one of the leaves of this graph to the root. While calculating this directed acyclic graph (DAG), the garbage collector will drop entire branches from memory when they are no longer reachable. The more fully connected the model is, the quicker the `BackPointer` graph will converge towards being linear on alignment length.

Space-saving versions of the Forwards and Backwards recursions can be similarly constructed. Since these algorithms consider all possible paths, there is no need to consider the `BackPointer` data structures, so these algorithms require memory proportional to the length of the shortest sequence and the number of states only.

This allows very large pair-wise alignment problems to be considered without memory resources becoming the limiting factor. Clearly, this does not remove the need to evaluate every part of the recursions, so the algorithms still scale computationally on the product of the lengths of the sequences being aligned.

The parameters of a `Distribution` (and by extension the emission and transition probabilities in an HMM) can be estimated using the `DistributionTrainer` interface. This provides a transactional framework for associating observed counts with a `Distribution`, for aggregating these, normalizing them and resetting them to zero. Given labelled data, parameters are estimated by adding whole counts to the `DistributionTrainer` proportional to the observations and then invoking the train method to update the `Distribution` parameters.

In many cases, a collection of distributions will need to be trained simultaneously. The `DistributionTrainerContext` interface encapsulates such a set. This allows all of the distributions within an HMM to be trained simultaneously. Since `Distribution` is an interface, there will often be cases when implementations do not actually store the values directly, but rather perform some calculation on the parameters of another `Distribution`. For example, there is an implementation of `Distribution` in BioJava that takes an underlying `Distribution` and a table that maps input `Symbol` instances to a `Symbol` for an underlying distribution. This allows, for instance, a `Distribution` over DNA symbols to have emission probabilities equal to those of the complementary symbols in another `Distribution`. When the distributions are registered with a `DistributionTrainerContext`, the implementations will ensure that counts for the complementary view are routed on to the underlying `Distribution` instance, and once the context is asked to train all the parameters from the aggregated counts, the complementary view will reflect the new parameters of the newly trained distribution.

Training distribution parameters via `DistributionTrainerContext` allows very complex parameterisation of models to be explored, both in terms of the emission probabilities and for the transitions as well, without altering the dynamic programming recursions or the routines used to collect observation counts.

HMMs have been constructed with this API to model 3-D DNA structures (`BasisSymbols` with one dimension per physical property and `Distributions` that model multinomial Gaussians over these properties), align pairs of protein secondary structure, find transcription factor binding sites, perform GIBBS sampling of expression data, find eukaryotic and prokaryotic promoters, as well as a host of other tasks.

2.7 Query

2.7.1 Motivations

Fairly early on in the use of the `Feature` interfaces, there was the need to find features of a particular type, or with particular properties, or some combination thereof. Initially we started adding many `getFooByBar()` methods, but it quickly became apparent that this would not scale.

2.7.2 Initial Implementation

After reading the Dragon compiler book (Aho, Sethi et al. 1985), we developed a small language for describing constraints for accepting or rejecting feature types, and added the method `filter(FeatureFilter aFilter, boolean recurse)` to the `FeatureHolder` interface. The `FeatureFilter` interface has the single method `accept(Feature)` which returns true if the feature is to be included in a return-set and false otherwise. There are implementations of `FeatureFilter` for accepting features based upon their properties (type, location, annotations and the like). There are also several logical filters. For example, `FeatureFilter.And` is an implementation of filter that will accept a feature if two other filters both accept it. There are implementations for the logical Operations ‘and’, ‘or’, ‘not’ and ‘nand’. Using these logical Operations and the basic filters, it is possible to build up quite sophisticated constraints. The ‘recurse’ flag in the filter method indicates whether to apply the filter to the current collection of features only or whether to apply it to those features and all features that they contain recursively. This provides coarse-grain control over how to navigate the feature hierarchy.

The filtering language allows collections of features to optimise the processing of requests as they can interrogate the query to find portions that they can easily process. For example, a given `FeatureHolder` implementation may know that it only contains features of a particular type. It can then optimally handle any filter expression that contains a `ByType` expression by just comparing the two types and either accepting or rejecting the entire set of features. This is much cheaper than comparing every feature in turn with the filter expression. Filters are used in nearly all library and script code that manipulates features. For data-specific implementations such as the DAS or Ensembl bindings, the ability to compare filters can be used to implement reasonable lazy-fetching strategies to avoid loading unnecessary information into memory from high-latency storage (the web, or an SQL database for example). For GUIs, the rules for deciding which features to display can be stored in these flexible filter objects and modified at will.

2.7.3 Limitations of This System

This scheme has served us well over the last two years. However, there are several shortcomings with this approach. The first one is that it can only be applied to features. To extend this to all BioJava objects would require many sets of filters to be written, each with rules about how to interpret them. In addition, in practice it would be nice to be able to retrieve sequences from a `SequenceDB` instance based upon whether they do or don’t contain a given type of `Feature`. This would require the ability to specify filters that span multiple object types. The other main shortcoming is the way that the recursion through the feature hierarchy is performed. For example, when retrieving features of a particular type within a region of a human chromosome we must recurse down through each level of the assembly pruning it as we go according to region and then search for features of that type at each level. This process is not easy to represent as a single filter. In practice, we end up constructing recursive function calls that each do non-recursive filters to prune the selection by location, find the features of the appropriate type to return, and recurs down to each feature with children.

Because the filter statement does not represent the entire process of finding the features, it is impossible to perform optimal searching and data-caching strategies for these complex cases. This causes potential inefficiencies to creep in to an otherwise elegant system.

To address these issues, we are evaluating a range of approaches for modelling complex queries, including finite state machines, ontology languages and graph grammars (refs).

2.8 Recent Developments

Since late 2001, BioJava has continued to be developed, expanding far beyond the original group of two individuals. There are now over thirty developers, five of which form the core development team. In this time, existing APIs have been consolidated, and new ones have been added. In this section we will discuss some of the areas where I have personally been the primary developer, as well as some of the functionality which the community has contributed.

The three primary areas of personal contribution are in the design and implementation of the tag-value parser framework, flat-file indexing, and a constraints-based type system for `Annotation` objects.

Major community contributions include improvements to the `FeatureFilter` language, the 'change hub' mechanism for managing large numbers of change listeners, bit-packed sequences, parsers for blast (and other sequence similarity search formats) and an emerging API for representing and manipulating ontologies. To a greater or lesser extent I have had personal involvement in each of these, but the bulk of the design or implementation has been undertaken by others.

2.8.1 The Tag-Value Parser

A large proportion of the data analysed by bioinformaticians is stored in text files. Commonly, these are structured as lists of records. Each record is composed from one or more lines that contain a tag specifying its type and an associated value. For example, EMBL entry files have entries separated by lines consisting of '//', and each entry has one or more lines with a two letter line-type identifier code (such as 'AC', 'OC' or 'FT') with a value present in columns 6 to 80. Genbank files have a similar structure, but in this case the record separator is '///' and the different line types are identified by full names (such as 'ACCESSION', 'ORGANISM' or 'FEATURE'). There are a large number of file formats that closely resemble either EMBL or Genbank files, but contain different tags and represent different types of information, such as classes of enzymes (Bairoch 2000), taxonomies (Benson, Karsch-Mizrachi et al. 2003) and protein families (Falquet, Pagni et al. 2002).

In our experience, developing custom parsers for these file formats is an error-prone task. One system that has implemented a general approach to parsing biological flat files is the SRS system with its language Icarus[22]. Here we describe a similarly generic framework for parsing these files within BioJava.

The tag-value framework is an attempt to provide a unified way to abstract out the common parts of the parsing task (such as recognizing record boundaries and dividing lines of text into tags and values) while allowing the exact details to be customised as needed. The approach we took was to use a mixture of the strategy (Gamma, Helm et al. 1994)[315] design pattern and liberal use of listeners. Strategies are used to encapsulate the variable portion of a process into an interface on its own, so that the unchanging portion can handle the unchanging functionality and delegate to the strategy where needed. All data is treated as Java `Object` instances rather than `String` instances, allowing the same framework to work un-changed on on-textual information.

Over all, the flow of parsing events is very similar to that in the Simple API for XML (SAX[23]). In the XML analogy, the text files are like XML files, the tag-value listeners are like SAX events, and the `Annotation` API is the equivalent of the Document Object Model (DOM[24]). There are also similarities with the Boulder IO package[25], as well as the way that the BioPerl SearchIO has been designed.

[22] We have been unable to find documentation about icarus on the LION bioscience web site. However, the EBI is currently providing documentation, which can be found at http://srs.ebi.ac.uk/doc/icarus.pdf

[23] The SAX standard is coordinated through the http://www.saxproject.org/ web site

[24] The DOM specification can be found at http://www.w3.org/DOM/

[25] BoulderIO is described at http://stein.cshl.org/software/boulder/

The class `Parser` has a single method `read(BufferedReader, TagValueParser, TagValueListener)` that reads all of the text from the buffered reader, uses the tag-value parser to process this into tags and values, and informs the tag-value listener of these pairs. For users of the API, this is the main method that they would invoke.

The `TagValueParser` interface encapsulates the process of splitting each line of input into a tag and a value, and also of deciding if the tag is new or not. If the tag is different from that on the previous line, then the parser assumes that it is new. In the case where it is the same, the tag-value parser can indicate that it should be treated as a new instance of that tag, rather than as an additional value for the current tag. For example, rather than SWISS-PROT comments being treated as just one series of values for a single comment tag, the tag-value parser could force a new comment tag event to be fired for each logical block of comments. There are implementations of `TagValueParser` that split lines into fixed-width areas (with two pre-built instances, for files formatted similarly to EMBL and Genbank), and one that splits according to a regular expression.

The `TagValueListener` interface has five methods. These are all invoked by a `Parser` instance, and it is the `Parser` that is responsible for ensuring correct nesting of these method invocations. The two methods `startRecord()` and `endRecord()` signal that records have started and ended respectively. All other events are emitted within the scope of this pair of events. The `startTag(Object)` and `endTag()` methods indicate that a tag has been started and ended respectively. These are called within the scope of the record. Tags are never directly nested. That is, for every `startTag()`, there is never a directly nested `startTag()` invocation, and for every `startTag()` there is exactly one matching `endTag()`. The `value(TagValueContext,` `Object)` method is used to inform the listener of values associated with the current tag. The `value()` method is only ever invoked within the context of a tag, and never directly within the context of the record. For each value associated with a tag, there will be a separate invocation of `value()`, and it is up to the listener how this should be interpreted.

Values can be replaced with `Objects` that are not `String` instances. For example, while parsing an EMBL entry, the lines relating to organism information could be transformed by a listener into a single taxonomy value. It is common to transform textual representations of things like URLs and Enzyme Classification (EC) numbers into light-weight objects. This greatly enhances the richness of the data consumed by the ultimate listener.

Some of these tag-value file formats have embedded sub-documents. For example, EMBL and Genbank files have an embedded feature table document. The tag-value framework supports these by using the context passed in as the first argument to `value()`. The listener can uses the `pushParser(TagValueParser, TagValueListener)` method to indicate to the `Parser` that all values of the current tag should themselves be split into tag and value pairs. The pushed tag-value parser will be used to split the values of these lines, and the results will be passed onto the pushed listener. Once the outer tag ends, the pushed parser and listener pair are popped back off the processing stack, and the original listener will be informed of an `endTag()` event as normal. The listener pushed will receive the full set of start/end record and tag events associated with the sub-document, and may itself choose to push new listeners for embedded documents.

This framework allows flexible processing of files into event streams. However, it is useful to further process these events. To support this there are a range of listener implementations that wrap other listeners, and pass on altered evens. For example, the accession lines of EMBL and Genbank files can contain a list of accession numbers, separated by semi-colons. A listener would receive one value event for each accession line. The data becomes easier to interpret if one value event can be produced for each accession number. A `RegexSplitter` instance could be used to recognize each portion of the accession line that is an accession, and then fire one value event to the wrapped listener for each accession.

The `ValueChanger` listener implementation is the class responsible for changing values associated with particular tags. It is responsible for either replacing a value with some other value, or for firing off multiple values. Again, the strategy pattern is used, in this case to factor out the mapping between tags and actions into a separate class named `ChangeTable`. A `ChangeTable` instance maintains a table of which actions are associated with which tags. This greatly promotes code reuse and modularisation. The `ValueChanger` code just manages the flow of events. The actions themselves are trivial to implement as little Java classes (often in practice as anonymous classes). We have found that this kind of composition and parameterisation is far superior to inheritance-based methods of customizing behaviour.

The `TagMapper` listener is used to systematically replace tags with other tags. For example, it would be possible to configure a `TagMapper` instance to map all EMBL tag names to Genbank tag names. This allows event streams to be transmuted into those accepted by standard listeners and factory objects. For example, by transforming reference information in to tags and values that resemble those emitted from the EMBL parser, the same reference handlers used for EMBL processing can be reused.

Using the built in tag-value classes, and by supplementing these where needed with some application-specific code, it is possible to rapidly develop parsers for tag-value formats of nearly any kind, and transform the information in these files into that required for a particular application, while achieving a very high degree of code reuse.

2.8.2 Flat File Indexing

A related problem to that of parsing these files is that of retrieving one entry among potentially the many hundreds of thousands of entries in a single or multiple files. It was decided by members of the OBF that it would be useful for all of the projects to share a mechanism for indexing these files. There exist a number of indexing strategies (for example, emblcd[26]). However, these tend to pose problems when accessed from multiple different languages and on multiple platforms as they are binary file formats. The OBDA flat file indexing specification[27] defines an indexing strategy that just uses plain text files to store the indices.

BioJava contains a full implementation of this specification, allowing a wide range of file types to be indexed, and for individual records to be fetched in time

[26] Applications for manipulating embl CD index files can be found at http://www.hgmp.mrc.ac.uk/Software/EMBOSS/Apps/dbiflat.html

[27] See http://cvs.biojava.org/cgi-bin/viewcvs/viewcvs.cgi/obda-specs/flatfile/indexing.txt?rev=HEAD&cvsroot=obf-common&content-type=text/vnd.viewcvs-markup for the most recent version of the specification.7

proportional to the cost of a binary search through the index. Records can be retrieved either by primary ID or by secondary IDs. The BioJava implementation is wholly compatible with the other OBF implementations (in Perl, Python and `C`).

Sequence files can be indexed using the standard BioJava classes for reading sequences from a stream. Additionally, any format that can be parsed with the tag-value framework can be indexed. In the future, we will be continuing to enhance support for secondary IDs, and provide simple APIs to allow different flat-file formats to be linked to one another by IDs in a manner similar to SRS[28].

2.8.3 Annotation Types

Since the beginning, BioJava has supported free-form associations of keys and values through the `Annotation` interface. Ironically, many applications need to be able to guarantee that certain property keys will be present in an `Annotation`, or that certain values will be present. The project started to gain a lot of repetitive and error-prone code that first checked an annotation to see what properties and values it had, and then acted accordingly.

In order to reduce the need to write this error-prone and repetitive code, we chose to develop a dynamic type system for `Annotation` instances, based around the new `AnnotationType` interface. The main two methods are `instanceOf(Annotation)` and `subTypeOf(AnnotationType)`. Both compare the argument to the current type. The `instanceOf()` method returns true if the argument is an instance of the type, and `subTypeOf()` returns true if the argument is a sub-type of the type. An `Annotation` is

[28] Mode information about SRS can be found at the http://www.lionbioscience.com/solutions/products/srs web site

an instance of an `AnnotationType` on the basis of what properties and values it has. One type is a sub-type of a super-type if every annotation that is an instance of the sub-type is also an instance of the super-type. Two `AnnotationType` instances are equivalent if exactly the same set of `Annotation` instances are accepted by the `instanceOf()` methods of both types.

Restrictions are placed upon the range of values that can be associated with properties by using instances of the `CollectionConstraint` interface. This has two main methods. The `accept(Object)` method returns true if the argument is acceptable to the constraint. The `subConstraintOf(CollectionConstraint)` method returns true if all items acceptable by the sub-constraint are also acceptable to the super-constraint. The `AnnotationType` `instanceOf()` and `subTypeOf()` methods are implemented purely by using these methods. There are implementations of CollectionConstraint that perform the normal logical operations, as well as checking properties of the item under consideration.

There are many utility methods in `AnnotationTools` that deal with the common operations that may be performed upon `AnnotationType` and `Annotation` instances. This use of the façade design pattern (Gamma, Helm et al. 1994)[185] insulates users of the API from its necessary complexities. `AnnotationTools` implements a wide variety of logical operations upon `AnnotationType` directly, such as computing types that are the logical union, intersection and difference of two types. It is very common to compare the results of these to the `AnnotationType` constants that accept or reject every `Annotation`. Additionally, it can be used to generate new `Annotation` instances from an old one and a type, for example, by retaining or removing all keys defined by the type.

The AnnotationType APIs differ from the way the Java type system works. In Java, every `Object` maintains a reference to its `Class`. This `Class` maintains references to all `Class`es that it inherits from, both implemented interfaces and extended classes (the Java introspection APIs use the same type to represent both classes and interfaces). With `AnnotationType`, `Annotation` instances maintain no such reference. As properties are added and removed, or the associated values are altered, the `Annotation` may change which annotation types it is an instance of. Code that wishes to check types should always use the `AnnotationType.instanceOf()` method. We tend to think of annotations as being 'castable to' an annotation type, rather than inheriting from or implementing them.

The `AnnotationType` interface works synergistically with annotations and the tag-value parsers. If the tag-value framework is like SAX and the `Annotation` API is like DOM, then `AnnotationType` is like XML-SCHEMA[29]. Annotation types are also used extensively in the recently enhanced implementation of feature filters.

2.8.4 Enhanced Feature Filters

The `FeatureFilter` APIs have since been developed by the community into a fully functional constraint language for `Feature` hierarchies. In addition to adding implementations for nearly all conceivable feature properties, there are now implementations that accept or reject a feature based upon the type of the annotation associated with the feature. Additionally, there is now much finer grained control of searches through the hierarchy, using filters like `ByAncestor` and `HasChild`.

[29] The XML Schema and related standards can be found at http://www.w3.org/XML/Schema

There is a façade class named `FilterTools` that implements many common operations upon feature filters. This includes composing new filters that are the union, intersection or difference between two filters and a range of factory methods. The results of these are often compared to the `FeatureFilter` constants that accept or reject all features. Another very common operation is to compare two `FeatureFilter` instances to see if they accept a disjoint set of features.

A range of `FeatureHolder` implementations now publish FeatureFilter instances as schemas describing what features they contain. Given a query, it is possible to efficiently see if the query is disjoint from the schema and potentially avoid comparing the contained features to the filter.

Some Feature and Sequence implementations now look at the FeatureFilter instances being passed into the filter() method, and check the filter for known types of annotation. For example, if a FeatureFilter is used to filter Ensembl, and it is constraining upon the "ensemble_id" property in the feature's annotation bundle, the Ensembl code will be able to recognize this and do optimized database lookups rather than looping over all possible feature instances.

For database and high-latency applications, such as Ensembl and DAS, this constraints-based language has allowed the existing Sequence and Feature APIs to scale gracefully to queries that potentially scan many hundreds of thousands of entities. By careful comparison of the filters with known schemas, and by introspecting the filters for constraints that can be optimised, we experience performance comparable to that for special case code that manually plans search strategies.

2.8.5 Change Hubs

Another bottle-neck upon scalability was the implementation of how change listeners were registered with resources such as database objects. Sequence databases that contain mutable implementations of `Sequence` need to behave as if the complete network is in place to forward events from every one of the sequences it contains. In the case of database implementations like BioSQL and Ensembl, there may be anywhere from a few hundred to several hundreds of thousands of sequence and feature instances that theoretically need to be listened to. However, in a typical application, only a few of these are ever directly accessible in memory.

In these special cases, there are implementations of the Changeability support classes which we call "change hubs" that maintain data-structures to keep track of which listeners logically exist. As each sequence is instantiated in memory, the change hub registers the required listeners to it. As the sequence goes out of scope, it takes care of removing the listeners. In this way, users of these databases can appear to be listening indirectly to a vast number of objects, while the implementation cost in terms of memory, and the time needed for event notification, can be kept to a minimum.

This is one of the many examples of where designing BioJava from the start in terms of interfaces has allowed us to drop in a complex replacement for some standard functionality without altering the APIs exposed to users.

2.8.6 Bit Packed Sequences

As it became more common to work with complete genomic information, the original implementation of `SymbolList` became impractical. It stores references to `Symbol` instances in an array. References in Java are the same size as a pointer. On

32-bit platforms, this is a clear overhead compared to storing bytes (8 bits), and this is even more pronounced on 64-bit platforms.

To allow very large sequences to be loaded into memory, an API was developed for mapping between `Symbol` instances and bit patterns. The `Packing` interface encapsulates the mapping between all `Symbols` in an `Alphabet` and unique bit patterns. To encode all atomic symbols, the bit patterns must be wide enough to encode the size of the alphabet. For example, the DNA alphabet has four members. This can be packed into two bits. The PROTEIN alphabet has 20 members. This will require 6 bits as 5 bits can only represent 16 combinations, but 6 bits can represent 32. To encode all symbols, including the ambiguities, the bit pattern will have one element per atomic symbol in the alphabet. This allows the presence or absence of that atomic symbol to be indicated. DNA therefore requires 4 bits, and PROTEIN requires 20 bits.

For small alphabets, like DNA and RNA, the memory saving associated with packing the sequence is considerable (2 bits vs. 32 or 64 bits). The performance penalty is approximately a factor of two compared to the implementation that accesses references directly for linear scans. However, the code that is shared between the packed and unpacked implementations has been tuned now to be over three times faster than it originally was, so that the current packed implementation is in fact faster than the original unpacked implementation.

Long symbol lists are implemented by storing a `List` of fixed-length symbol list instances. When a `symbolAt()` request comes in to the symbol list, it calculates which child list it is in, extracts that symbol list and passes the request on (after adjusting the index accordingly). One benefit of this is that if a sub-list has been taken of a region, and that region falls totally within the range of one of the child symbol lists, the sub-list will only maintain a reference to this child symbol list, allowing the large one to be garbage collected once it goes out of scope.

Another benefit is that different child lists can be implemented using different `SymbolList` implementations. Most sequences present today have a very small number of ambiguity symbols, and when they are present, they are usually runs of 'n' characters. Child lists that have no ambiguity at all can be packed using the most efficient packing possible. Child lists that do have ambiguity can be packed using an ambiguity-capable packing.

The bit-packing APIs have facilitated the implementation of a pure-java implementation of SSAHA (Ning, Cox et al. 2001), as well as making it feasible to store complete human chromosomes in memory. It demonstrates again the flexibility afforded by interface-based design. No methods in the `Alphabet` or `SymbolList` interfaces needed to be modified, meaning that all existing applications benefit from these improvements without alteration.

2.9 Conclusions

The BioJava APIs outlined here are designed to be extremely flexible, while imposing minimal restrictions on how the interfaces are implemented. This is achieved by pervasively using Java interfaces rather than abstract classes to define APIs, and leveraging nested exceptions to handle errors. Potentially variable behaviour is systematically encapsulated by strategy objects. The Changeability API allows programmers to maintain tight control over which object may be modified and under what circumstances this will be allowed, while also facilitating the synchronization of objects' state and simultaneously enforcing the principal of strong object encapsulation. The Symbol and Location APIs provide examples of how careful object modelling can make software disproportionately powerful by ensuring that the interfaces have a complete but minimal set of operations that allow for all conceivable uses of the objects. The `DistributionFactory` and `DPFactory` interfaces demonstrate how tailor-made implementations of interfaces can be instantiated without the users of APIs needing to know about these implementations. The creation of features by using `Feature.Template` instances demonstrates how two-dimensional polymorphism (interface and implementation) can be implemented without pushing responsibility for data-integrity to the users of the API. The `FeatureFilter` and `AnnotationType` APIs demonstrate how data structures can be queried efficiently without violating encapsulation.

Because of the strongly interface-centric design, it is fairly easy to view underlying data in several forms by defining only the transformation to be applied. For example, `OrderNSymbolList` views an underlying symbol list as the n^{th} order view. A 1^{st} order view of a DNA sequence would produce a `SymbolList` with the alphabet DNAxDNA, containing symbols that represent each overlapping pair of symbols from the original sequence. Similarly, distributions can be constructed that represent a translated view of another distribution, such as the complementary distribution. This paradigm allows very elegant data-structures to be built up without duplicating either the underlying data or the code that performs view transformations. Indeed, BioJava strongly discourages data duplication. As an extreme example, to reverse-complement an entire chromosome, BioJava would construct a `ComplementarySymbolList` that views a `ReverseSymbolList` that views the underlying `SymbolList` for the chromosome. The total additional cost in memory for complementing the chromosome is two Java object instances, rather than the memory for the entire