BioSmalltalk v 1.0

Hernán Morales

26/12/2016

Contents

Introduction

BioSmalltalk is a library for doing bioinformatics with the Smalltalk programming system. This guide's goal is to be useful for bioinformaticians, biologists, or life scientists, with little or no experience at all in programming or object technology. It will explain everything needed to understand how to use BioSmalltalk to reach your objectives, also contains the specific context and vocabulary to communicate properly in mailing lists when you need help. Object-Oriented Programming concepts are not required for basic tasks (like building sequences, parsing files, and exploring results) but are specially necessary to build more complex objects and/or debugging code, for example.

Notes and Disclaimers

BioSmalltalk is free of charge and far from perfect. The developer does not provide any guarantees for quality, availability, or fitness for particular purpose. BioSmalltalk is licensed with the MIT License.

If you feel that you can contribute to **BioSmalltalk** development, please do not hesitate to contact through the available communities.

Installation

BioSmalltalk is currently implemented in Pharo, a Smalltalk dialect. It is distributed in several forms depending of the target platform:

- For Microsoft Windows as a .exe installer.
- For Mac/OS X as a compressed ZIP file.
- For GNU/Linux as a compressed .tgz file.

Once installed, all distributions contains:

- A virtual image file (.image extension)
- A changes file (.changes extension)
- The Smalltalk Virtual Machine (VM) (executable file)
- A group of folders including examples and test data.

The usual way to run BioSmalltalk is to open the "Environment", a run-time Integrated Developement Environment (IDE) where all coding activities are performed. This means launching the VM and selecting a virtual image. At first you will have just one image - the one you downloaded in the BioSmalltalk distribution - but as you gain more experience with Smalltalk you will find having several images for different purposes.

You may also work by executing scripts from .st files (explained in the Executing Scripts section) however you would miss powerful Smalltalk tools like Browsers, Refactoring Tools, Code Critics, Finders, etc.

Smalltalk Basics

You may evaluate scripts or code in BioSmalltalk as in most Smalltalk environments: Selecting the code, right button, and choose one of the following operations:

- **Print it** (**Show it**, or **Display it**): To print the results as a simple String next to the selected code.
- **Inspect it**: To open an Inspector window. You may observe and send messages over the selected object in the left pane
- **Explore it**: To open an Explorer window. You may expand the tree items, and send messages over the selected object.

At least you should work with two tools. A ”System Browser” and a ”Workspace”. The famous System Browser, Class Hierarchy Browser, or simply Browser, is the most important tool for development in Smalltalk and enables to explore and modify the categories, classes, methods in the system. Actually is so powerful that you can do more advanced actions like finding class references, perform code critics and refactorings, find method senders and implementors, execute tests, group classes, manage breakpoints, find method examples and past versions, among other actions.

Additional References

A comprehensive description of any Smalltalk system would deserve a complete book on its own. Fortunately, there are several good books devoted to describe the Smalltalk language and environment. I recommend the following ones as starting points:

- If you are very new about object technology, a **must** read and one of the best books is “Object Technology” (1997, ISBN: 0-201-30994-7) from David Taylor.
- The Smalltalk Best Practices Patterns is a good book for coding conventions and style: http://www.amazon.com/Smalltalk-Best-Practice-Patterns-Kent/dp/013476904X

- There are free on-line books in the Stéphane Ducasse personal web site: http://stephane.ducasse.free.fr/FreeBooks.html
- Pharo Books: http://files.pharo.org/books/

Finally, a useful advice often done by experienced Smalltalkers is to actually experiment with the environment, make mistakes, and ask in the mailing lists or forums in case of doubt.

Scripting Languages versus Smalltalk

Table 1: Comparison of coding taks

Python/Perl/Ruby	**Pharo Smalltalk**
Write a script (in a file) called foo.py/foo.pl/etc	Write a script in some environment tool (Workspace, Script Manager, etc.)
Specify a shebang line	Not necessary
Code navigation	Open the System Browser
Search import modules and how to ”invoke” them	Not necessary
Search for senders/callers of a function: Configure ctags for your language	In a method, press the Senders button
Place your source code in directories or/and files	Not necessary, the snapshot system will persist your source code
Execute your program : Find a way to execute the command line interpreter/compiler specifying with your script name as input	Select any piece of valid source code and press the mouse right button, then do it or print it

Usually depends of a web browser on the generated pages from cross-linking the source code and comments by an autodoc tool. This setting usually doesn't allow targeted navigation.

Executing Scripts

As mentioned, you can execute scripts in .st files from command-line. Just cd into the directory where the ”pharo” script is found and evaluate the following test expression to check everything is working properly:

```
./pharo Pharo.image eval ”3 + 4”
```

To run a script in a file you can pass the file name as argument to the image:

```
./pharo Pharo.image my_script.st
```

Bioinformatics with BioSmalltalk

Developing and using Object-Oriented technology in BioSmalltalk is about finding your objects of interest in the (virtual) image. Think about the virtual image, or simply the image, as a snapshot of the system current state.

If you want to experiment with sequences, your first task is then to find classes related to sequences (hint: Search for **BioSequence** in the Browser). Once found, you can easily select methods and explore the senders to check how they are used. You will see some methods require a minimum instantiation effort, like a reverse complement, some others will require you to set up a whole context of many objects to figure out how to use them.

Although the above workflow could be valid for all the Bioinformatics libraries, in BioSmalltalk you have probably the most powerfull tools to explore a system through targeted navigation. This is a big change if you are too used to scripting and sweeping raw textual files.

Sequences

A biological sequence is a single, continuous molecule of nucleic acid or protein, and represents a naturally occurring, or experimentally generated, fragment of genetic or protein material or any intermediate product (like the messenger RNA). It is possibly the most important object in bioinformatics and most bioinformatics workflows involve using sequences at some point.

In BioSmalltalk sequences are represented with **BioSequence** object. We will talk about representation later. If you have previous programming experience, a sequence is essentialy a **String** object with an alphabet. It is read-only, if you want to modify a sequence in place you may use **BioMutableSeq**.

Creating a DNA sequence

```
BioSequence newDNA: 'atcggtcggctta'.
```

The above one-liner creates a ”BioSequence instance”. Nucleotide letters are case-insensitive, this is, meaning does not change based on differing use of uppercase and lowercase letters. When the alphabet information of a sequence is missing (as in this case, or when reading sequences from FASTA files), **BioSequence** will try to guess the alphabet type. Notice you can also create a sequence specifying the alphabet, for example:

```
BioSequence newAmbiguousDNA:'
   AAGTCAGTGTACTATTAGCATGCATGTGCAACACATTAGCTG'.
BioSequence newUnambiguousDNA:'
   AAGTCAGTGTACTATTAGCATGCATGTGCAACACATTAGCTG'.
```

Similarily, to create a RNA and Protein sequence, you may use:

```
BioSequence newRNA: 'auugccuacauaggc'.
BioSequence newProtein: 'AGFAVENDSA'.
```

Please note: Next to the following expressions, between quotations, is displayed the result of the "Print it" (or "Show it" depending the Smalltalk flavor you are using) operation

The format of the **BioSequence** printString is:

```
"
aBioSequence( sequence_size ) ([ letters_in_alphabet ] alphabet_class)
    [sequence_letters]
"
```

An easier way to create sequences, without specifying the **BioSequence** class, is converting a **String** into a Sequence object. The following expressions display the contents of the #asSequence message.

```
'ATGCTAGN' asSequence.  " a BioSequence(8) ([GATCN] IUPAC DNA) [
   ATGCTAGN] "

'RSWAFFGH' asSequence.  " a BioSequence(8) ([ACDEFGHIKLMNPQRSTVWY]
   IUPAC –> Protein) [RSWAFFGH] "
```

```
'AGCUCGGCU' asSequence. " a BioSequence(9) ([GAUCRYWSMKHBVDN] IUPAC —>
    RNA —> Ambiguous) [AGCUCGGCU] "

'VPFSNATGVVRSPFEYPQYYLAE' asSequence
" a BioSequence(23) ([ACDEFGHIKLMNPQRSTVWY] IUPAC —> Protein) [
   VPFSNATGVVRSPFEYPQYYLAE] "
```

Mutable Sequences

A biological mutable sequence is like a normal sequence, but it is editable, i.e. not read-only.

Take note that Smalltalk, as in R, index starts counting from 1. One of the benefits of 1-based system is a more human-centered thinking, rather than 0-based counting which are more machine centered. The following script modifies the first position in a RNA sequence:

```
(BioMutableSeq newRNA: 'auugccuacauaggc')
        at: 1 put: $g;
        yourself.
" a BioMutableSeq(15) ([GAUC] IUPAC -> RNA -> Unambiguous) [
   guugccuacauaggc] "
```

You may completely replace the sequence if appropriate, this could be convenient for preserving additional information (annotations like sequence features) still valid for the new sequence.

```
(BioMutableSeq newUnambiguousDNA: 'ATGC')
        seq: 'CGTA';
        yourself.
```

Operations on Sequences

The following subsections describes available operations on sequences.

Basic Operations

Accessing a subsequence and display it as a **String**:

```
((BioSequence newDNA: 'atcggtcggctta') copyFrom: 3 to: 5) asString.  "
    --> 'cgg' "
```

Count the number of bases of a sequence:

```
(BioSequence newUnambiguousDNA:'AAGTCAGTGTACTATTAGCATGGCTG') size.
 " --> 26 "
```

Get the molecular weight (MW) of a sequence:

```
'AGTACACTGGT' asSequence molecularWeightNonDegen  " --> 3435.23"
```

Get the GC nucleotide content:

```
'AGTACACTGGT' asSequence gc.
```

Sequence complement

```
(BioSequence newDNA: 'AAGTCAGTGTACTATTAGCATGCATGTGCAACACATTAGCTG')
   complement

" a BioSequence(42) ([GATCN] IUPAC DNA) [
   TTCAGTCACATGATAATCGTACGTACACGTTGTGTAATCGAC] "
```

Reverse complement

This message answers a copy of the receiver's sequence complemented and with element order reversed. You may need to work with the reverse complement of a sequence if it contains an ORF on the reverse strand.

Upper and lower case are preserved as can be used to mark regions of interest. Spaces and special letters (like '-' and spaces) are left untouched.

```
(BioSequence newAmbiguousDNA: 'AcTG C—NH' ) reverseComplement.
" a BioSequence(9) ([ACGTWSMKRYBDHVN] IUPAC DNA —> Ambiguous) [DN—G
   CAgT] "
```

Ambiguous and unambiguous nucleotide sequences are supported

```
(BioSequence newUnambiguousDNA: 'GGGGaaaaaaaatttatatat')
   reverseComplement
" a BioSequence(21) ([GATCN] IUPAC DNA) [ATATATAAATTTTTTTTCCCC] "
```

Supports BioSequence and BioMutableSeq

```
(BioMutableSeq newDNA: 'AAGTCAGTGTACTATTAGCATGCATGTGCAACACATTAGCTG')
   reverseComplement

" a BioMutableSeq(42) ([GATCN] IUPAC DNA) [
   CAGCTAATGTGTTGCACATGCATGCTAATAGTACACTGACTT] "
```

In the answered sequence, the alphabet is preserved.

```
| mySeq |
mySeq := BioSequence newDNA: '
   AAGTCAGTGTACTATTAGCATGCATGTGCAACACATTAGCTG'.
mySeq alphabet = mySeq reverseComplement alphabet
```

Transcription and back transcription

ToDo

Sequence translation

Translation of RNA could be performed in a one-liner:

```
(BioSequence newRNA: 'AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG' )
    translate.
" a BioSequence(13) ([ACDEFGHIKLMNPQRSTVWY] IUPAC —> Protein) [MAIVMGR
    *KGAR*] "
```

the same could be achieved stopping at the first stop codon:

```
(BioSequence newRNA: 'AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG' )
    translateToStop
" a BioSequence(7) ([ACDEFGHIKLMNPQRSTVWY] IUPAC —> Protein) [MAIVMGR]
    "
```

Translate and concatenate a DNA sequence to Protein sequence:

```
| dna protein |
dna := BioSequence newDNA: 'atcggtcggctta'.
protein := BioSequence newProtein: 'AGFAVENDSA'.
protein , dna translate.
```

It is possible to specify the translation table, according to the NCBI Genetic Codes tables (check the transl_table identifier):

```
(BioSequence newRNA: 'AUGGCCAUUGUAAUGGGCCGCUGAAAGGGUGCCCGAUAG' )
    translateWithTableId: 2.
" a BioSequence(13) ([ACDEFGHIKLMNPQRSTVWY] IUPAC —> Protein) [
    MAIVMGRWKGAR*]"
```

Warning: After you select another translation table, the selection remains until you otherwise specify other table. To see which table is currently selected evaluate:

```
BioCodonTables currentCodonTable
```

which in our last example should answer:

```
” a BioCodonTable [2] a Dictionary('AAA'->'K' 'AAC'->'N' 'AAG'->'K'
  'AAU'->'N' 'ACA'->'T' 'ACC'->'T' 'ACG'->'T' 'ACU'->'T' 'AGC'->'S'
   'AGU'->'S' 'AUA'->'M' 'AUC'->'I' 'AUG'->'M' 'AUU'->'I' 'CAA'->'Q
  ' 'CAC'->'H' 'CAG'->'Q' 'CAU'->'H' 'CCA'->'P' 'CCC'->'P' 'CCG'->'
  P' 'CCU'->'P' 'CGA'->'R' 'CGC'->'R' 'CGG'->'R' 'CGU'->'R' 'CUA
  '->'L' 'CUC'->'L' 'CUG'->'L' 'CUU'->'L' 'GAA'->'E' 'GAC'->'D' '
  GAG'->'E' 'GAU'->'D' 'GCA'->'A' 'GCC'->'A' 'GCG'->'A' 'GCU'->'A'
  'GGA'->'G' 'GGC'->'G' 'GGG'->'G' 'GGU'->'G' 'GUA'->'V' 'GUC'->'V'
   'GUG'->'V' 'GUU'->'V' 'UAC'->'Y' 'UAU'->'Y' 'UCA'->'S' 'UCC'->'S
  ' 'UCG'->'S' 'UCU'->'S' 'UGA'->'W' 'UGC'->'C' 'UGG'->'W' 'UGU'->'
  C' 'UUA'->'L' 'UUC'->'F' 'UUG'->'L' 'UUU'->'F' ) ”
```

To set the default codon table, evaluate:

```
BioCodonTables currentCodonTable: BioCodonTables defaultCodonTable
```

Alphabets

IUPAC

IUPAC is one of the many non-governmental organizations composing the International Council for Science. In practice, for the bioinformatician, serves as a ”standard” for the naming of the chemical elements and compounds.

Amongst other nomenclatures, in bioinformatics the *IUPAC codes* usually identify amino acids and nucleotide bases. These codes can consist of either one letter code or a three letter code. For example DNA letters are A, G, C, T bases, which compose the DNA Alphabet. Alphabets are used to determine the contents of a sequence, because simple String(s) are not enough. Consider the following sequence:

```
'GTCG'
```

it could be interpreted as:

- A DNA fragment of Guanine,Thymine,Cytosine and Guanine.
- A Peptide with Glycine, Threonine, Cysteine and Glycine.

the solution is to disambiguate specifying the correct Alphabet objects when appropriate.

Alphabets classes are located under the **BioAlphabet** class hierarchy. To access the ambiguous alphabets just ask to the IUPAC albhabet abstract class:

```
BioIUPACAlphabet ambiguousAlphabets. --> an OrderedCollection(
   BioIUPACAmbiguousRNA BioIUPACAmbiguousDNA)
```

The following script displays the alphabet letters available in the IUPAC specific classes:

```
BioIUPACAmbiguousDNA codes --> ACGTWSMKRYBDHVN
BioIUPACAmbiguousRNA codes --> GAUCRYWSMKHBVDN
BioIUPACAmbiguousDNA ambiguityCodes --> WSMKRYBDHVN
BioIUPACAmbiguousRNA ambiguityCodes --> RYWSMKHBVDN
```

```
BioIUPACUnambiguousDNA codes --> GATCN
BioIUPACUnambiguousRNA codes --> GAUC
BioIUPACDNAExtended codes --> GATCBDSW
BioIUPACProtein codes --> ACDEFGHIKLMNPQRSTVWY
BioIUPACProteinExtended codes --> ACDEFGHIKLMNPQRSTVWYBXZJUO
BioNullAlphabet codes --> -
```

Notice however the IUPAC encoding lacks of symbols for termination, gap (-), or selenocysteine.

Create and display sequence objects guessing the alphabet:

```
| sequence |
sequence := 'atcggtcggctta' asSequence.
sequence alphabet.

sequence := 'gccuacau' asSequence.
sequence alphabet.
```

Sequence Records

BioSeqRecord stores a sequence with additional information, usually called ”annotations”. You can think of a Sequence Record like a wrapper for a **BioSequence** to attach additional information to a sequence. Equivalent classes in other Bio tookits are Bio::Sequence for BioRuby and Bio.SeqRecord for Biopython. In a sequence record usually the #id method is used to store an unique identifier such as the Accession Number (a **BioAccession**), and the #name is used to hold a **String** with a more descriptive name, such as clone name. The #description method is used to set up a **String** with a textual, i.e. human readable, description of the sequence.

A sequence record responds to the #annotations message to retrieve the sequence annotations. Such message will answer a Smalltalk **Dictionary**, however, if you need to customize the annotations you may subclass **BioSeqRecord** and override #annotationsClass method to answer a different kind of **Dictionary**.

Creating a **BioSeqRecord**:

```
| sequence |
sequence := BioSequence newProtein: '
   mdstnvrsgmksrkkkpkttviddddcmtcsacqslkllndfas'.
(BioSeqRecord new: sequence)
        id: 'P20994.1';
        name: 'P20994';
        description: 'Protein A19';
        dbxRefs: #('Pfam:PF05077' 'InterPro:IPR007769' 'DIP:2186N');
        annotationAt: #note put: 'A simple note';
        yourself.
```

Mostly you would want to iterate over a collection of sequences with metadata information, and populate Sequence Records with annotations and features. Some Bio* libraries use to parse a GenBank file (.gbk) which is a textual format intended to be read by humans, even when there exists for years programming utilities like Entrez E-Utils to retrieve information in a machine-readable format, which are easier to parse. One of these formats is XML and the following example retrieves a XML-formatted file with annotations, which are then stored in BioSeqRecord objects:

Work in progress

```
'seq_records.xml'
```

Variable number tandem repeat

Microsatellites and Minisatellites Locus

BioSmalltalk can use VNTR sequences using two objects:

- The **BioMicrosatelliteLocus** or **BioMinisatelliteLocus** are used to represent "static" or unvariable information about a microsatellite or minisatellite respectively.
- The **BioMicrosatelliteSequence** or **BioMinisatelliteSequence** are used to represent a particular sequence **using** a locus (BioMinisatelliteLocus or BioMinisatelliteLocus).

In the following examples we are using as reference the Cattle STR information accessible at http://www.cstl.nist.gov/biotech/strbase/cattleSTRs.htm. Specifically we are going to set up the BM1818 loci through code. A microsatellite locus can have a simple or a compound repeat structure. You can set this with the proper message sends:

```
BioMicrosatelliteLocus
    name: 'BM1818';
    chrLocation: 'D23S21';
    beSimpleRepeat;
    yourself
```

Microsatellite and Minisatellite Sequences

Work in progress

Parsing

About parsing

Parsing is one of the most important parts of developing software in Bioinformatics. Parsing is a process by which a developer makes sense of a sentence (acting as input data, usually a **String**), usually by breaking it down into letters or words.

A parser is an object that scans an input **String**, decomposes it into its constituent parts (which you may define), and then optionally processes the parts in some suitable manner.

Programming literature usually refer to writing a parser *by hand* when you write a parser without any support programming library. Currently taking such approach is rare and not recommended (because keeping track of contingencies caused by different inputs and input paths is extremelly difficult), so developers usually use a parser generator, i.e. a library, and write rules for this generator.

Parsing in BioPharo

BioSmalltalk uses different libraries to simplify and enable parsing of the many formats out there in the Bioinformatics arena. The libraries used are:

- XMLSupport for parsing XML format in DOM or SAX mode.
- XMLPullParser for parsing XML in StAX mode.
- PetitParser for parsing formatted or structured text.

Parsing Sequences

For example, finding if a given **String**, like 'TCGTACGA', is actually a DNA sequence you just evaluate:

```
#dnaSequence asParser matches: 'TCGTACGA'.  --> true
#dnaSequence asParser matches: 'TCGTZZZACGA'.  --> false
```

And for parsing:

```
#dnaSequence asParser parse: 'TCGTACGA'. --> 'TCGTACGA'
#dnaSequence asParser parse: 'TCGTA3344CGA'. --> end of input
   expected at 5
```

The #asParser method actually instantiates a parser, which is then sent the message #matches: to test the given input.

Another common way to use the existing parsers is through the **BioParser class**. In BioSmalltalk we have two modes of parsing: Tokenize and Parse. The tokenize messages always answer a Collection-like object containing other collections or primitive Smalltalk objects (this is, something like #('object1' 'object2') or 1234, 74.32, true, etc.), while the parse messages answer **BioObject**'s.

You may want BioObjects if you need to keep working with the resulting bioobjects from your parsing. Or to learn relationships between BioObjects.

You may work with primitive Smalltalk objects like **Integers** or **Strings** if you want "cheap" objects.

Parsing Accession Numbers

An accession number is a unique sequential number assigned to each record (sequence) in a repository. This number allows for tracking of different versions of a sequence record and the associated sequence over time. Parsing an accession number answers a very simple object which you may use to store or index separatedly from its sequence, thus, reducing storage space and processing time. To parse an accession number:

```
BioParser parseAccession: 'CAF97855.1' -->  a BioAccession CAF97855 1
```

You may also query if the accession number is versioned:

```
(BioParser parseAccession: 'CAF97855.1') isVersioned --> true
```

And query the version number:

```
(BioParser parseAccession: 'CAF97855.1') version --> '1'
```

Another shortcut for the same message is #asAccession

```
'XP_425521.2' asAccession version --> '2'
```

Parsing FASTA files

This is how to parse a FASTA file with multiple sequences:

```
BioParser parseMultiFasta: (BioFASTAFile on: 'my_sequencesHM.fasta')
   contents.
```

Given a header, split its identifiers:

```
BioParser tokenizeFastaHeader: '>gi|198282148|ref|NC_011206.1|
   Acidithiobacillus ferrooxidans ATCC 53993 chromosome, complete
   genome'.

"  #('>gi' '198282148' 'ref' 'NC_011206.1' 'Acidithiobacillus
   ferrooxidans ATCC 53993 chromosome, complete genome') "
```

Print in the Transcript each sequence description with its sequence size from a FASTA String:

```
| fastaString |

fastaString := '>gi|2765658|emb|Z78533.1|CIZ78533 C.irapeanum 5.8S
   rRNA gene and ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGATGAGACCGTGGAATAAACGATCGAGTG
AATCCGGAGGACCGGTGTACTCAGCTCACCGGGGGCATTGCTCCCGTGGTGACCCTGATTTGTTGTTGGG
CCGCCTCGGGAGCGTCCATGGCGGGTTTGAACCTCTAGCCCGGCGCAGTTTGGGCGCCAAGCCATATGAA
AGCATCACCGGCGAATGGCATTGTCTTCCCCAAAACCCGGAGCGGCGGCGTGCTGTCGCGTGCCCAATGA
ATTTTGATGACTCTCGCAAACGGGAATCTTGGCTCTTTGCATCGGATGGAAGGACGCAGCGAAATGCGAT
```

```
AAGTGGTGTGAATTGCAAGATCCCGTGAACCATCGAGTCTTTTGAACGCAAGTTGCGCCCGAGGCCATCA
GGCTAAGGGCACGCCTGCTTGGGCGTCGCGCTTCGTCTCTCTCCTGCCAATGCTTGCCCGGCATACAGCC
AGGCCGGCGTGGTGCGGATGTGAAAGATTGGCCCCTTGTGCCTAGGTGCGGCGGGTCCAAGAGCTGGTGT
TTTGATGGCCCGGAACCCGGCAAGAGGTGGACGGATGCTGGCAGCAGCTGCCGTGCGAATCCCCCATGTT
GTCGTGCTTGTCGGACAGGCAGGAGAACCCTTCCGAACCCCAATGGAGGGCGGTTGACCGCCATTCGGAT
GTGACCCCAGGTCAGGCGGGGGCACCCGCTGAGTTTACGC

>gi|2765657|emb|Z78532.1|CCZ78532 C.californicum 5.8S rRNA gene and
   ITS1 and ITS2 DNA
CGTAACAAGGTTTCCGTAGGTGAACCTGCGGAAGGATCATTGTTGAGACAACAGAATATATGATCGAGTG
AATCTGGAGGACCTGTGGTAACTCAGCTCGTCGTGGCACTGCTTTTGTCGTGACCCTGCTTTGTTGTTGG
GCCTCCTCAAGAGCTTTCATGGCAGGTTTGAACTTTAGTACGGTGCAGTTTGCGCCAAGTCATATAAAGC
ATCACTGATGAATGACATTATTGTCAGAAAAAATCAGAGGGGCAGTATGCTACTGAGCATGCCAGTGAAT
TTTTATGACTCTCGCAACGGATATCTTGGCTCTAACATCGATGAAGAACGCAGCTAAATGCGATAAGTGG
TGTGAATTGCAGAATCCCGTGAACCATCGAGTCTTTGAACGCAAGTTGCGCTCGAGGCCATCAGGCTAAG
GGCACGCCTGCCTGGGCGTCGTGTGTTGCGTCTCTCCTACCAATGCTTGCTTGGCATATCGCTAAGCTGG
CATTATACGGATGTGAATGATTGGCCCCTTGTGCCTAGGTGCGGTGGGTCTAAGGATTGTTGCTTTGATG
GGTAGGAATGTGGCACGAGGTGGAGAATGCTAACAGTCATAAGGCTGCTATTTGAATCCCCCATGTTGTT
GTATTTTTTCGAACCTACACAAGAACCTAATTGAACCCCAATGGAGCTAAAATAACCATTGGGCAGTTGA
TTTCCATTCAGATGCGACCCCAGGTCAGGCGGGGCCACCCGCTGAGTTGAGGC
'.

(BioParser parseMultiFasta: fastaString) do: [ : seqRecord |
                Transcript
                        show: seqRecord name; cr;
                        show: seqRecord size; cr; cr ]
```

Alignments

Underlying sequences are in a BioAlignment are sequence based (contrary to String based). This could enable to access track information like quality score through the alignment itself, instead of maintaining multiple associated matrices.

Creating a Multiple Sequence Alignment

```
align := BioAlignment new.
align
    addSequence: 'ACTGCTAGCTAG';
    addSequence: 'ACT-CTAGCTAG';
    addSequence: 'ACTGCTAGATAG';
    addSequence: 'ACTGCTTGCTAG';
    addSequence: 'ACTGCTTGATAG'.
```

BLAST

Provided by the National Center for Biotechnology Information (NCBI), BLAST is a sequence homology search program that finds all the sequences that match a particular given sequence from a database. The NCBI-GenBank database contains over 150 billion nucleotide bases as of June 2013, and doubling its contents approximately every 18 months. It is considered the most fundamental tool in bioinformatics today, although there are many other similar programs for different cases. In BioPharo you may query programmatically a BLAST at NCBI with the following parameters:

- Query the nucleotide (nr) database.
- Use an ”unknown” DNA sequence.
- Get only the first 200 hits.
- Filter query sequence with low compositional complexity.
- Use the default expect value (10).
- Use the BLASTN programs to search nucleotide databases using a nucleotide query.
- Get the results in XML for further querying

which is translated in BioPharo as:

Make a BLAST from BioPharo

```
| search |
search := BioNCBIWebBlastWrapper new nucleotide
        query: 'TCGAAATAACGCGTGTTCTCAACGCGGTCGCGCAGATGCCTTTGCTCATC
AGATGCGACCGCAACCACGTCCGCCGCCTTGTTCGCCGTCCCCGTGCCTC
AACCACCACCACGGTGTCGTCTTCCCCGAACGCGTCCCGGTCAGCCAGCC
TCCACGCGCCGCGCGCGCGGAGTGCCCATTCGGGCCGCAGCTGCGACGGT
GCCGCTCAGATTCTGTGTGGCAGGCGCGTGTTGGAGTCTAAA';
        hitListSize: 200;
        filterLowComplexity;
        expectValue: 10;
        blastn;
        formatTypeXML;
        fetch.
```

Now the ”search” object contains a BioWebBlastResult. The following line write the XML contents to a timestamped file in the directory where the .image file is located:

```
search outputToFile: 'blast-' , BioObject currentSeconds , '.xml'.
```

”Note: When the Transcript is opened (Tools --> Transcript) you may view a detailed log of message sends.”

Reading and Writing Sequences

Reading FASTA records

The following script count letters for each fasta record in the 'Fasta_example_1.fa' file included in the BioSmalltalk distribution. The file contains multiple fasta records which are parsed and iterated collecting the numbers of letters for each record.

```
| mFasta fPath |

fPath := BioObject testFilesFullDirectoryName , 'Fasta_example_1.fa'.
mFasta := BioParser parseMultiFasta: (BioFASTAFile on: fPath)
    contents.
mFasta collect: #occurrencesOfLetters
```

And of course the output is not nice for the human eye:

```
 an OrderedCollection(a Dictionary($A->24 $C->18 $G->25 $T->23 ) a
   Dictionary($A->33 $C->30 $G->23 $T->34 ) a Dictionary($A->25 $C->30
    $G->15 $T->39 ) a Dictionary($A->17 $C->9 $G->19 $T->15 ) a
   Dictionary($A->26 $C->17 $G->36 $T->41 ) a Dictionary($A->8 $C->4 $
   G->11 $T->31 ) a Dictionary($A->31 $C->16 $G->17 $T->22 ) a
   Dictionary($A->16 $C->14 $G->15 $T->25 ) a Dictionary($A->14 $C->20
    $G->11 $T->21 ) a Dictionary($A->20 $C->29 $G->25 $T->41 ) a
   Dictionary($A->12 $C->14 $G->8 $T->26 ) a Dictionary($A->26 $C->8 $
   G->11 $T->24 ) a Dictionary($A->36 $C->37 $G->16 $T->36 ) a
   Dictionary($A->30 $C->30 $G->33 $T->27 ) a Dictionary($A->14 $C->11
    $G->13 $T->20 ) a Dictionary($A->34 $C->31 $G->20 $T->27 ) a
   Dictionary($A->23 $C->31 $G->29 $T->37 ) a Dictionary($A->30 $C->42
    $G->25 $T->31 ) a Dictionary($A->7 $C->23 $G->5 $T->18 ) a
   Dictionary($A->15 $C->28 $G->6 $T->18 ) a Dictionary($A->23 $C->44
   $G->12 $T->37 ))
```

Suppose we want to display the output in CSV format. Each object could be formatted for CSV by implementing #outputAsCsvTo: aStream. Then modifying the script:

```
| mFasta filePath |

filePath := BioObject testFilesFullDirectoryName , 'Fasta_example_1.fa
    '.
mFasta := BioParser parseMultiFasta: (BioFASTAFile on: filePath)
    contents.
BioCSVFormatter new exportFrom: (mFasta collect:
    #occurrencesOfLetters).
```

It is possible to get a timestamped file like "CSVOutput3536647736.csv" with contents like this:

```
A;24
G;25
C;18
T;23

A;33
G;23
C;30
T;34

A;25
G;15
C;30
T;39

...
```

Entrez

The Entrez API is an interface for accessing to the Entrez Utilities provided by the NCBI.

Entrez Examples

FASTA record from Protein

- Input: Accession number
- Database: Protein
- Output: FASTA record

```
(BioEntrezClient new protein
    term: 'NP_031402.3';
    useWebEnv;
    setFasta;
    fetch) outputToFile: 'NP_031402.3.fasta'
```

SeqID record from Protein

- Input: Accession number
- Database: Protein
- Output: SeqID

```
(BioEntrezClient new protein
    term: 'NP_031402.3';
    useWebEnv;
    setSeqId;
    fetch) outputToFile: 'NP_031402.3.seqid'
```

GenBank records from Nuccore

- Input: List of UIDs
- Database: Nuccore
- Output: GenBank formatted file (.gb)

```
(BioEntrezClient new nuccore
    uids: #(57240072 57240071 6273287 6273291 6273290 6273289
   6273286 6273285 6273284);
    setGb;
    fetch) outputToFile: 'fetchNuccore3.gb'
```

XML records from Pubmed

- Input: List of UIDs
- Database: Pubmed
- Output: XML

```
| result |
result := BioEntrezClient new pubmed
            uids: #( 11877539 11822933 );
            setModeXML;
            fetch.
result outputToFile: 'eFetch1.xml'
```

XML records from Protein

- Input: Search terms
- Database: Protein
- Output: List of UIDs in XML

```
(BioEntrezClient new protein
    term: 'insulin AND homo';
    search) outputToFile: 'insulin AND homo_protein.xml'
```

Swiss-Prot and ExPASy

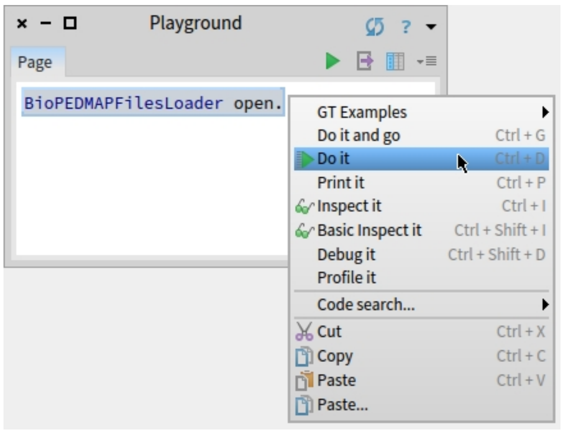
PDB

Population Genetics

Reading PLINK (PEDMAP) files

BioSmalltalk includes a simple utility window to browse and select PED/MAP files. You can use it for your projects and launch it by evaluating:

```
BioPEDMAPFilesLoader open.
```



```
BioPEDMAPFilesLoader open.
```

Figure 1: Opening the PEDMAP loader

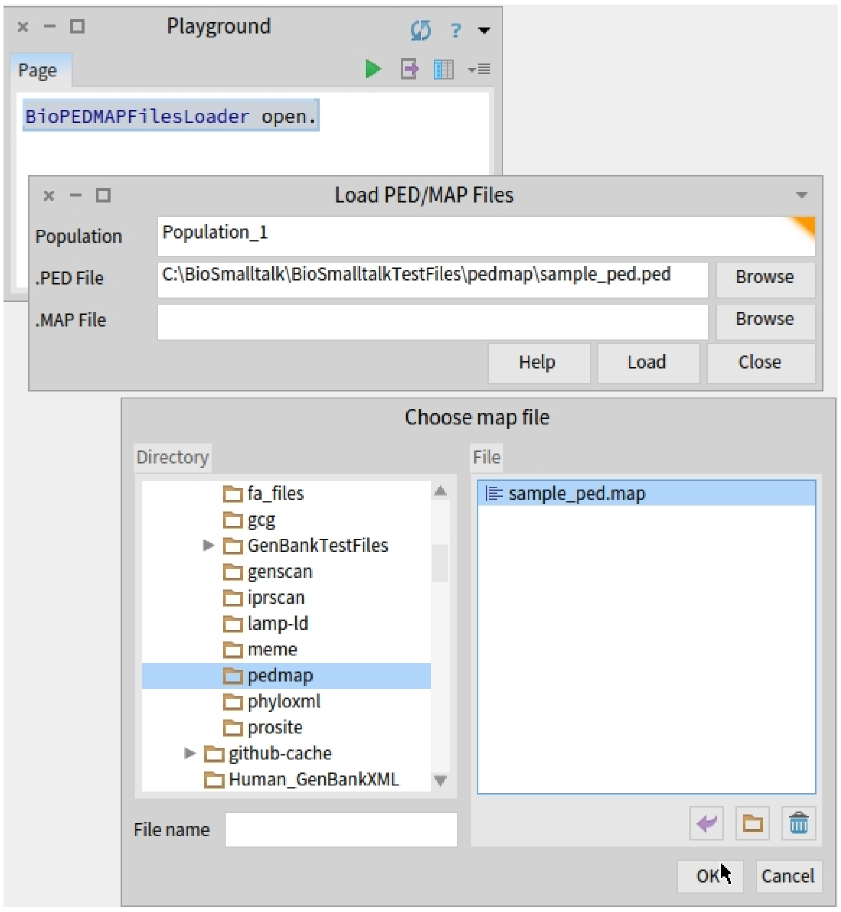
The window will require mandatory population name, a .PED and .MAP files to be selected.

Once opened, you can browse to your PED/MAP files and click Load to perform an action.

Which action is performed by default? If no action was specified, then the Load button will close the PED/MAP loader and return an Association with the file name of the selected .PED file as key, and the name of the .MAP file as value. The answered directory name will have a system dependent path separator (e. g. / for Unix) appended:

```
"'C:\BioSmalltalk\BioSmalltalkTestFiles\pedmap\sample_ped.ped'->'C:\
   BioSmalltalk\BioSmalltalkTestFiles\pedmap\sample_ped.map'"
```

You can also specify which action to realize after the Load button gets clicked.



```
BioPEDMAPFilesLoader open.
```

Population	Population_1
.PED File	C:\BioSmalltalk\BioSmalltalkTestFiles\pedmap\sample_ped.ped
.MAP File	

Figure 2: Selecting MAP file

```
BioPEDMAPFilesLoader open
    actionBlock: [ : pedFilename : mapFilename | ”...” ]
```

Phylogenetics

Clustering

Graphics

Graphics are built and rendered using the Roassal agile visualization engine, which is already installed in the BioPharo distribution. The Roassal engine includes a large set of examples using short Smalltalk expressions.

DotPlots

A simple visualization for the comparison of two sequences (or one sequence against itself) is DotPlot , which provides an overview of the extent of similarity between sequences without diving into details. DotPlots operates by projecting two sequences to be compared along the horizontal and vertical axis of a Matrix. DotPlots requires three main parameters:

- A ”Word Size”, which is a **Number** representing the length of the identical polynucleotide or polypeptide that must be found in both sequences in order to generate a dot on the chart. The bigger the word size, the lower is the probability that the same segment is present in both sequences.
- A ”Scoring Scheme”, specified as a **Matrix**. The simplest scoring scheme is the Identity Matrix, but others can be used, for example the EMBOSS DNA scoring matrix. Other extended matrix is used when sequences contain ambiguity codes, or using proteins.
- A ”Threshold” or ”cut-off score”.

When plotting mummer output, it is necessary to use the lengths of the input sequences to set the plot ranges, otherwise the plot will be automatically scaled around the minimum and maximum data points

Any character or symbol that does not belong to the [a-zA-Z] set is ignored.

Drawing a 2D DotPlot with two random sequences

```
BioRTDotPlot
    x: BioSequence atRandom y: BioSequence atRandom;
    window: 3;
    threshold: 2;
    open.
```

Drawing a 2D DotPlot for single analysis (identify repeats in a sequence)

```
BioRTDotPlot
        sequence: BioSequence atRandom;
        window: 3;
        threshold: 2;
        open.
    open.
```

Bar Charts

@@todo Under construction, please contact mailing list to collaborate.

Histograms

Histograms are frequently used to get an overview of quantitative distributions. Examples where histograms are applied are: Distribution of codon usage, sequence lengths,

Drawing a bar chart of a FASTA file with multiple sequences

```
| fastaSeqs g ds seqRange |
" Read and parse FASTA file with multiple sequences "
fastaSeqs := (BioParser parseMultiFastaFile: BioObject
   testFilesDirectoryName asFileReference / 'ls_orchid.fasta').

" Filter sequences "
seqRange := fastaSeqs select: [ : fastaRec | fastaRec size between:
   750 and: 790 ].

" Build diagram "
g := RTGrapher new extent: 500 @ 200; yourself.
```

```
ds := RTStackedDataSet new.
ds points: seqRange;
        x: #name;
        y: #size.
ds barShape width: 10.
ds histogramWithBarTitle: [ : e | (e sequenceDescription first: 15) ,
   '...' ].
g add: ds.

" Configure axis settings "
g axisY
        title: 'Count';
        noDecimal.
g axisX
        noLabel;
        noTick;
        title: 'Sequence length (bp)'.

" Open visualization in a new window "
g open.
```

Widgets

BioSmalltalk includes widgets to provide user interfaces. These utilities are small re-usable windows which could be easily integrated into your applications.

Chromosome Selection

A small utility to select organisms and their chromosomes for posterior analysis. Each organism has already configured their corresponding list of chromosomes.

```
BioChrSelector open.
```

CSV Explorer

The BioCSVExplorer class builds a widget that allow users to visually explore a CSV file and import its contents to the image.

Work in progress

Matrix Explorer

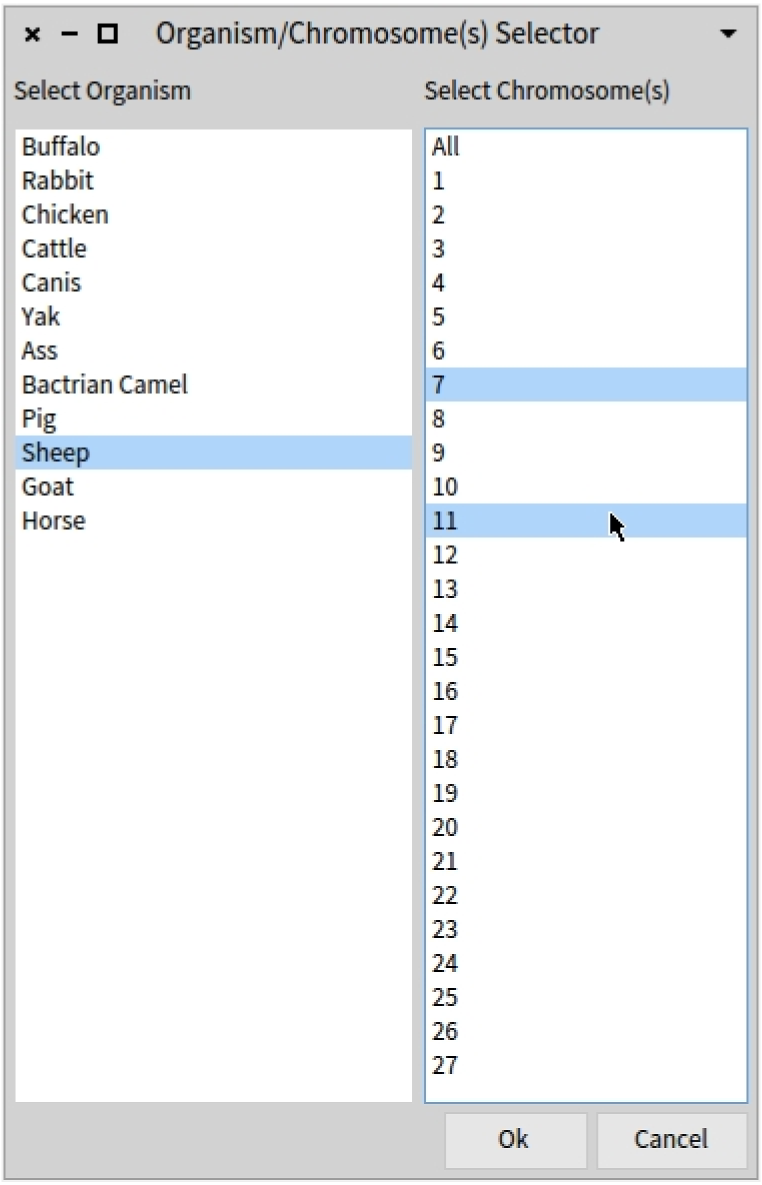
Work in progress

Wrappers

HMMER

To test if the HMMER wrapper is working properly for your platform evaluate:

```
BioHMMERWrapper new
        help;
        execute.
```



Select Organism:

- Buffalo
- Rabbit
- Chicken
- Cattle
- Canis
- Yak
- Ass
- Bactrian Camel
- Pig
- Sheep
- Goat
- Horse

Select Chromosome(s):

- All
- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27

Figure 3: Selecting MAP file

Successfull execution should answer a BioResultObject with results:

Now this is translated code from the HMMER tutorial:

```
| samplesDir hmmFile stoFile fastaFile outputFile |

samplesDir := BioObject testFilesFullDirectoryName.
” hmmbuild globins4.hmm tutorial/globins4.sto ”
hmmFile := samplesDir , ’globins4.hmm’.
stoFile :=  samplesDir , ’globins4.sto’.
BioHMMERWrapper new
        addParameter: hmmFile;
        addParameter: stoFile;
        runHmmBuild;
        execute.

” hmmsearch globins4.hmm uniprot sprot.fasta > globins4.out ”
fastaFile := samplesDir , ’globins45.fa’.
outputFile :=  samplesDir , ’globins4.out’.
BioHMMERWrapper new
        runHmmSearch;
        addParameter: hmmFile;
        addParameter: fastaFile;
        addOption: ’o’ value: outputFile;
        execute.
```

Cookbook

Third Party Libraries

This section includes description of libraries not specific to ”Bio” tasks, but useful for development of any software utility.

Persistency

All current Pharo images includes Fuel, a fast binary serializer that defines a pickle format based on clustering similar objects. It reduces deserialization overhead compared to other approaches. It supports class shape changing (when a variable has been added, or removed, or its index changed), cyclic object graphs, serialization of several special objects like **CompiledMethod** (in a binary way, without compiling code) and Classes, etc.

Mathematics

The package *Numerical Methods* includes common numerical algorithms implemented in Smalltalk. The package library is documented in the free downloadable book Object-Oriented Implementation of Numerical Methods. An Introduction with Smalltalk , which covers all the available algorithms:

- Interpolation: Lagrange, Newton, Neville, Bulirsch-Stoer, Cubic Spline.
- Iterative algorithms
- Integration methods: Trapeze, Simpson, Romberg.
- Series: Infinite Series, Continued Fractions, etc.
- Linear Algebra: Vectors, Matrices, Linear Equations, LUP decomposition, Eigenvalues and Eigenvectors.
- Distributions: Normal, Gamma, etc.
- Statistical Tests: Chisquare-test, F-test, T-test, Linear Regression, Maxi
- mum Likelihood, etc.
- Optimization: Hill Climbing, Powell, Simplex, Genetic, etc.

OS Communication

The package OSProcess provides access to operating system functions, including pipes and child process creation. It is implemented using pluggable primitives in a shared library for Unix or Linux, and a DLL for Windows. The Smalltalk code, including the classes which implement pluggable primitives for Unix or Win32 operating system functions, may be loaded into any Squeak image, but the primitives are only useful on Unix and Windows systems. Placeholder classes are provided for MacOS, OS/2 and RiscOS, but are not yet implemented.

CommandShell is a Smalltalk implementation of a command processor shell and terminal window. It is intended to behave like to a simple terminal window (like xterm) running a Unix command shell (like /bin/sh). It lacks some elements of Unix shell syntax, and does not provide terminal emulation, but it adds some nice Smalltalk enhancements such as a text editor which works in a command pipeline, and the ability to evalute Smalltalk expressions in a command pipeline with unix commands.

Developing BioSmalltalk

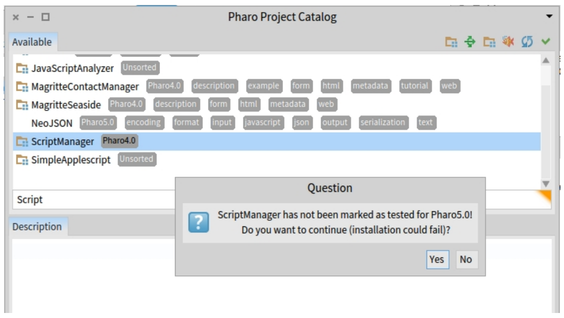
Code Organization

User Code Organization

At this point, you probably have your own scripts and wonder how to maintain your source code. You have several choices, depending on what is your task on course:

The Script Manager

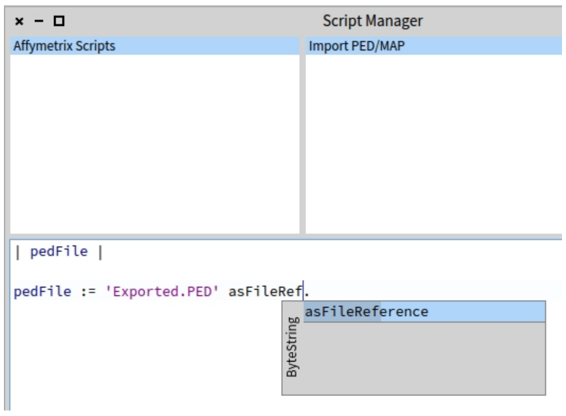
The Script Manager is a script browser tool developed by Torsten Bergman which enables to organize your scripts. It allows to have multiple workspaces organized by categories and within each category, to name scripts, execute, save and export them. It also supports syntax highlighting and autocompletion. Unfortunately the tool was removed from official Pharo images, but you can install it through the Catalog Browser:


Pharo Project Catalog
Available
JavaScriptAnalyzer Unsorted
MagritteContactManager Pharo4.0 description example form html metadata tutorial web
MagritteSeaside Pharo4.0 description form html metadata web
NeoJSON Pharo5.0 encoding format input javascript json output serialization text
ScriptManager Pharo4.0
SimpleApplescript Unsorted
Script
Description
Question
Yes
No

ScriptManager has not been marked as tested for Pharo5.0!
Do you want to continue (installation could fail)?

Figure 4: Installing the Script Manager

To open the tool, go to ”Tools” -> ”ScriptManager”



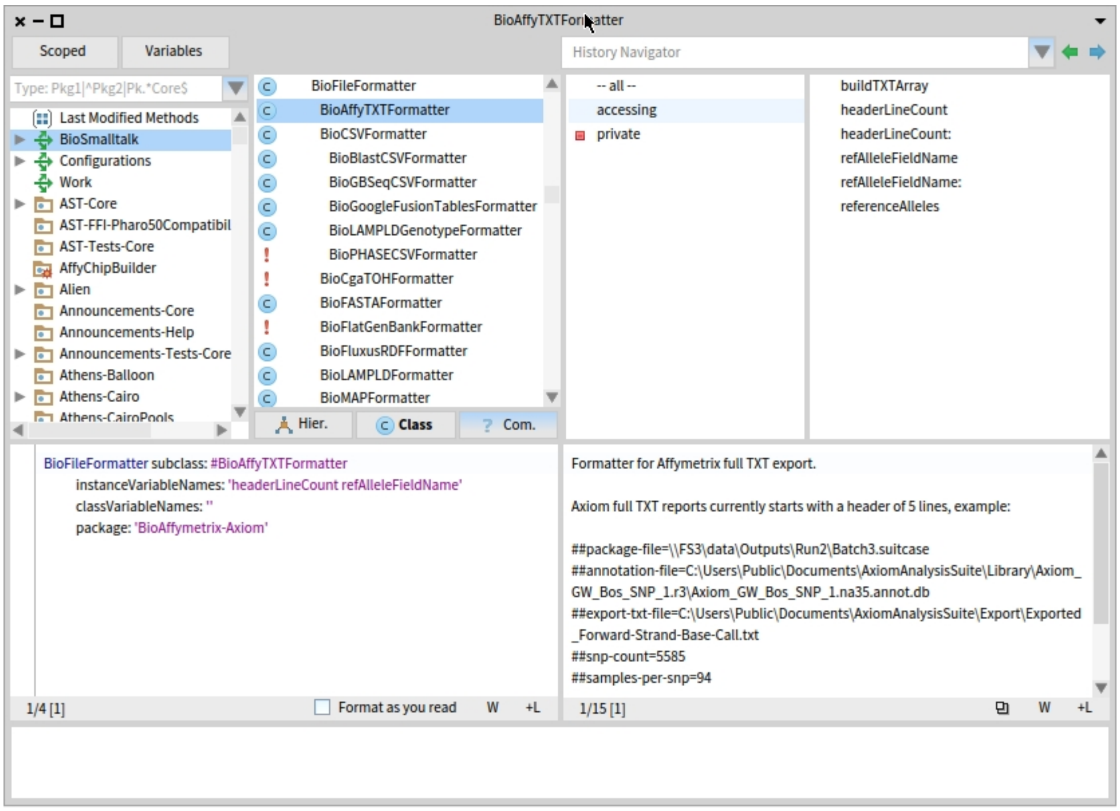
```
| pedFile |

pedFile := 'Exported.PED' asFileRef.
```

Figure 5: Using the Script Manager

The Nautilus Bio Browser

To facilitate working with Bio API, all the library classes are grouped in the System Browser:



```
BioFileFormatter subclass: #BioAffyTXTFormatter
	instanceVariableNames: 'headerLineCount refAlleleFieldName'
	classVariableNames: ''
	package: 'BioAffymetrix-Axiom'
```

```
Formatter for Affymetrix full TXT export.

Axiom full TXT reports currently starts with a header of 5 lines, example:

##package-file=\\FS3\data\Outputs\Run2\Batch3.suitcase
##annotation-file=C:\Users\Public\Documents\AxiomAnalysisSuite\Library\Axiom_
GW_Bos_SNP_1.r3\Axiom_GW_Bos_SNP_1.na35.annot.db
##export-txt-file=C:\Users\Public\Documents\AxiomAnalysisSuite\Export\Exported
_Forward-Strand-Base-Call.txt
##snp-count=5585
##samples-per-snp=94
```

Figure 6: Browsing the BioSmalltalk Classes

Developer Code Organization

Packages are organized in the following way:

Table 2: BioSmalltalk Code Organization

Package	**Description**
BioBlast	Desktop and web wrappers, filter and reader classes for Blast results and software
BioClassifier	A basic rule engine system and feature classifier
BioEntrez	Access to the Entrez Utilities API
BioFormatters	Basically file format writers. Naming strategies and other utilities also included here
BioTools	Samples could be methods returning String's for other subsstems or Scripts writing output to files
BioToolsSamples	Not necessary, the snapshot system will persist your source code
BioSupport	Classes and methods working in all Smalltalk platforms
BioPharo	Code specific for the Pharo platform
BioSqueak	Code specific for the Squeak platform
BioNCBI	Code to access NCBI's QBLAST service and convenience utilities.
BioParsers	Contains all the parsers. Patches for specific platforms are in their respective packages.
BioPopulation	Classes and methods for working with population genetics.
BioProject	A basic application framework for organizing work in projects.
BioWrappers	Core for wrapping desktop and web programs + specific wrappers for common bioinformatics software.

Building from Source

To build an image from source you will need at least wget or curl to download a Pharo clean image. In all cases, the following instructions creates a BioSmalltalk directory and download a clean Pharo image. Packages are fetched from SmalltalkHub, although there are plans to migrate source code to GitHub. You can build the BioSmalltalk image in two ways: Interactive or non-interactive.

Non-interactive building in Windows

It is highly recommendable to download and install MSYS Shell from the MinGW project. Additionally, you should install the msys-wget, msys-zip and msys-unzip packages from the MinGW Package Manager.

```
mkdir /c/BioSmalltalk
cd /c/BioSmalltalk
wget -O- get.pharo.org/50+vm | bash -
./pharo Pharo.image config ”http://smalltalkhub.com/mc/hernan/
   BioSmalltalk” ”ConfigurationOfBioSmalltalk” --printVersion --
   install=development
```

Non-interactive building in GNU/Linux

```
mkdir /usr/local/src/BioSmalltalk
cd /usr/local/src/BioSmalltalk
wget -O- get.pharo.org/50+vm | bash -
./pharo Pharo.image config ”http://smalltalkhub.com/mc/hernan/
   BioSmalltalk” ”ConfigurationOfBioSmalltalk” --printVersion --
   install=development
```

Interactive Building

Interactive building enables you to view the installation progress and to live debug installation problems. You should start the image from command line, and execute a Smalltalk installer script. To start the Pharo image evaluate:

```
./pharo Pharo.image &
```

One inside Smalltalk, you can evaluate the installer using Gofer or Metacello. Gofer is an API on top of Monticello, a package system of Smalltalk. Both Gofer and Monticello are preinstalled in current Pharo images. The following script uses Gofer:

```
Gofer it
  smalltalkhubUser: 'hernan' project: 'BioSmalltalk';
  configurationOf: 'BioSmalltalk';
  loadDevelopment
```

Metacello is a newer cross-smalltalk package management system. Supports named versioning, dependencies, metadata, conditional package loading and other features.

```
Metacello new
    smalltalkhubUser: 'hernan' project: 'PhyloclassTalk';
    configuration: 'PhyloclassTalk';
    version: #development;
    load.
```

Testing

Testing is the evaluation of your developed software product. Being Smalltalk the technology where the original testing framework was invented (SUnit), writing comprehensive tests is a traditionally important part of the package validation process.

A rapid and excellent introduction to Unit Testing in Smalltalk can be found in the Stéphane Ducasse article: SUnit Explained Revisited http://www.cs.pdx.edu/~black/OOP/papers/SUnitEnglish2.pdf. A tutorial about testing is beyond the scope of this guide, however I would like to introduce some issues about the nature of test writing before starting to write tests:

- Assume the presence of errors, always.
- It's impossible, generally, to find all errors.
- Test to find errors, not to show that it works.
- Not finding errors doesn't mean that your software doesn't contain errors.
- A successful test run discovers errors!
- Write for both valid and invalid conditions.

More useful advices are found in the Pharo By Example book: http://pharobyexample.org/

Executing Tests

You can execute tests by selecting the method you want to test in the Browser:

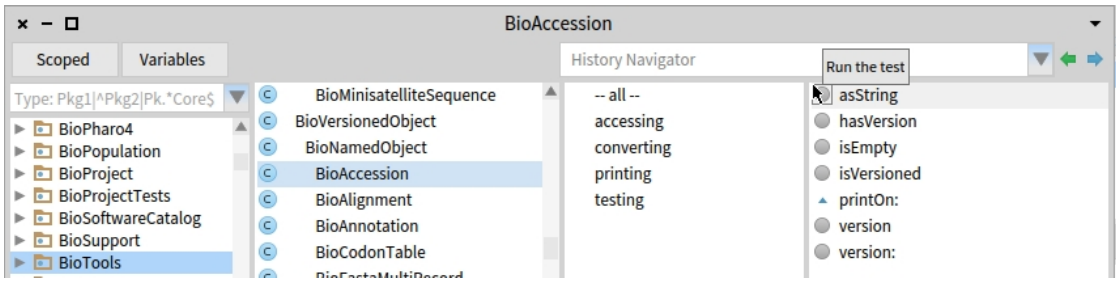


Figure 7: Selecting method to be tested

Once clicked, if the test pass the icon will be displayed in green color.

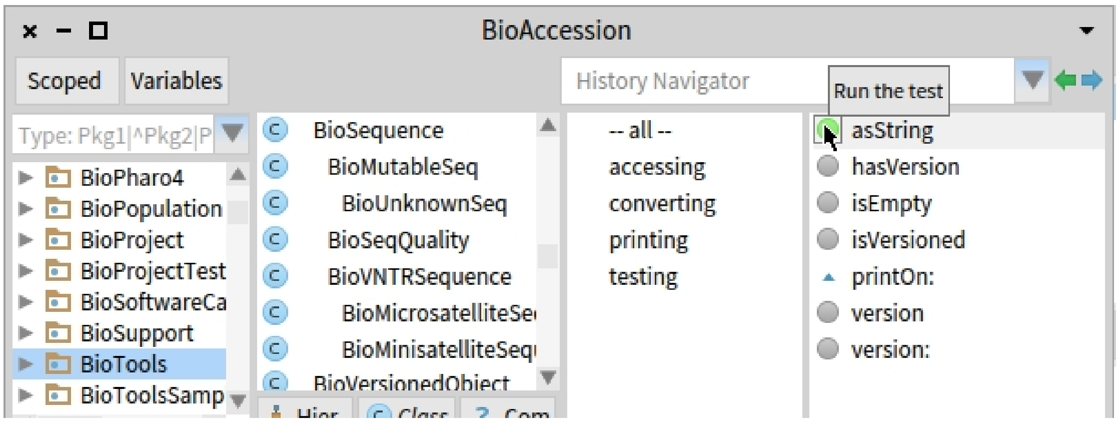


Figure 8: Executing a test on a method

Writing Tests

These are guidelines for writing tests in BioSmalltalk. Eventually they will be mandatory.

- All test classes should inherit from **BioAbstractTest** class
- Tests should be in the "testing" method category
- Tests selectors should start with the prefix "test".

Test Data

Including your own test data set:

Each BioSmalltalk distribution must include a "BioSmalltalkTestFiles" subdirectory (from now the "test files subdirectory") to place all the files used for the test suites.

- If you want to include data sets in your tests, then place them as files in this subdirectory.
- If your data is bigger than 8 Kb, then compress in one or several files and send appropriate messages to uncompress them.
- Do not override the contents or edit the file index.html in the test files subdirectory, it is needed for serving a "document root" when testing the local web servers.

- Testing of databases results is NOT encouraged.

Using already existing data sets in the image:

- In your BioSamples subclass, instance side, add a category named "samples-data" and add appropriate sample data. Beware that each Smalltalk flavor has their own limits in the method size and supporting huge data sets in the image is not encouraged.

```
BioYourTestCase>>testFilesDirectoryName

  ^ super testFilesDirectoryName , BioObject fileDirectoryClass slash
    , 'GenBankTestFiles'
```

SubSystems

The Engines SubSystem

BioSmalltalk supports interchangeable strategies by enabling the selection of the most appropriate underlying software package (called engine in our context) or to discover the best option for specific tasks. This is because in Smalltalk flavors very often you have several packages to perform the same task.

The developer should never hard-code a specific external replaceable package class name in a method, instead, he should write the corresponding adapter subclass and link it to the particular package (see below). Then the BioSmalltalk user could use a generic name for the particular tasks, instead of referring to implementations, which could fail, become unsupported or too slow. For example, to serialize your objects, use

```
BioSerializationEngine serialize: anObject.
```

(In Pharo: to materialize your serialized file just open the File Browser, select the file and Materialize from the right click menu)

To get the contents at an URL

```
(BioWebEngine for: anURL) httpGet
```

Adding an Engine

BioAbstractAdapter class defines the interface and abstract behavior for Adapters.

- What we call “Providers” refers to the external classes which are adaptee (**OSPlatformExecutor**, **KomServer**, etc). They could be missing in your platform.
- What we call “Adapters” refers to the final subclasses acting as adapters (prefixed by “Bio” for example: **BioOSPlatformExecutor**, **BioKomServer**, etc)

The information of which is the current engine in a hierarchy of engines is maintained by the AdapterClass class variable in the corresponding hierarchy. To access the current adapter for an engine evaluate:

```
BioSerializationEngine adapter
```

The information of which is the current provider in the current engine is maintained by the “provider” instance variable.

If you want to add your engine to BioSmalltalk, these are the required steps:

1) Check for the kind of engine you want to support at subclasses of BioAbstractAdapter, currently there are engines for:

- Web clients (The historical HTTP client from Squeak, WebClient from Andreas Raab, ZnHttpClient from Sven Van Caekenberghe)
- Serialization clients (Fuel, SRP, SmartReferenceStream, ReferenceStream, Ma object serialization, BOSS, SIXX/SIF)
- OS Execution wrappers.

If there is no engine type listed, you will have to check by yourself specific users of your methods and refactor code to support multiple strategies, this can be as easy as changing one method, adding a superclass and implementing some template methods, or as complex as reorganizing a whole set of BioSmalltalk hierarchies.

2) Create your subclass and implement these CLASS methods:

```
#clientClass
    " Answer a Symbol, the main external class name for this adapter "
```

Remember to name the class as a Symbol, do not assume the client has installed your adapter. To answer the actual class use for example: ˆ self classFor: #WebUtils

```
#isPreferredAdapter
    " Private – Answer <true> if the receiver is the best option for
   the current image and platform "
```

```
    ^ true
```

Don’t assume the client will have ANY client actually installed, so you might want to check if #providerIsAvailable

The Parsing SubSystem

This intends to be a general guide for writing a parser in BioSmalltalk. Parsing is a whole area in its own, and writing a parser generally is very difficult and takes time, so be careful when choosing your learning sources. To build a framework to start learning about writing parsers, the following concepts could be useful:

- Determine that you actually need a parser. Recall: A parser is an object that scans an input String, decomposes it into its constituent parts (which you may define), and then optionally processes the parts in some suitable manner.
- How: Most tutorials calls writing a parser "by hand" when you write a parser without any support programming library, so developers use a "parser generator" and write rules for this generator. Why: Because keeping track of contingencies caused by different inputs and input paths is extremelly difficult, and a generated parser code can be proved mathematically.
- **BioParser** is the root class for parsers. Parsers use several parsing libraries
- **XMLSupport** for parsing in DOM or SAX mode (see below to understand XML parsing modes)
- **XMLPullParser** for StAX parsing (see below to understand XML parsing modes)
- **PetitParser** for parsing formatted or structured text. If you don't know anything about PetitParser a good guide to start is http://scg.unibe.ch/research/helvetia/petitparser

Adding a new parser

Create the access point in the BioParser class

- Instead of forcing the developers to find the appropriate parser subclass on usage, the BioParser class was implemented as a Facade pattern providing class methods which hides the complexity of the hierarchy, this is, put every access point (method) in the **BioParser class** side.
- Comment the implemented method, this is mandatory.

Create the parser class

- If you want to parse formatted/structured text, subclass from BioAbstract-TextParser or specific subclasses (read class comments to find out which superclass is appropriate for your parser)
- If you want to parse XML, regardless of the mode, subclass from BioXML-Parser

Create the main access method(s) for your parser:

- If you need to return a **Collection** object (**Array**, **OrderedCollection**, etc.), then prefix the selector with #tokenize, example: #tokenizeLocus:, #tokenizeFastaHeader:, #tokenizeSwissProtEntryName:
- If you need to return a BioObject, then prefix the selector with #parse, example: #parseFasta: will return a BioFastaRecord, #parseAmbiguousWith-Separators: will answer a BioSequence with a **BioIUPACAmbiguousDNA** as alphabet.

The BLAST XML parser

Our motivation for building the initial BLAST XML parser is the chance of having customized DOM sub-trees in memory that could be subsequently filtered through messages, not by specifying node names which could change, be sensitive to typos, and are not suitable for performing computational reflection on them. For example, given a blastXmlTree object, instead of filtering by

```
blastXmlTree filter: [: hit |
    hit name = ‘’Hit_def and: [ hit value = ‘Homo ’sapiens ] ]
```

the idea is to add semantics to our tree:

```
blastXmlTree hitDefinitionsEqualTo: 'Homo sapiens'.
```

which enables to modify completely the implementation or algorithms in the method #hitDefinitionsEqualTo: preserving the same functionality for the programmer. The key in parsing BLAST XML contents are the #select... methods to narrow the parsing to specific element groups. This is, instead of building the whole XML tree into memory (which would be the DOM technique) the idea is that users will only be interested in specific DOM sub-trees.

The Entrez SubSystem

The main class of this subsystem is **BioEntrezClient**, it is the only class that should be visible for users. The database classes are located in the **BioEntrez-Database** hierarchy, below in this document there is a step guide to add a new database. The database hierarchy does not validate currently the support rettype's and retmodes as described in http://www.ncbi.nlm.nih.gov/books/NBK25499/table/chapter4.chapter4_table1/?report=objectonly

This means the developer is responsible to build correctly the request with valid parameters.

The entrez commands are supported through the **BioEntrezCommand** hierarchy, which contains code to validate command parameters and the base url of the requests. Currently the mandatory documentation for implementation is located at: http://www.ncbi.nlm.nih.gov/books/NBK25499/

The code samples are located at **BioESamples**, when providing new features, a new method should be created at the appropriate method category.

Steps for adding an Entrez database

- If the database supports retmode and rettype parameters, subclass from **BioERetDatabase**
- If the database supports any of the following parameters, subclass from **BioESeqDatabase**: strand, complexity, seq_start, seq_stop
- If the database is any of the Literature Databases (PubMed, PubMed Central (PMC), Journals, OMIM), subclass from BioELitDatabase
- Add a new method named #dbName in class side of the new recently created class, for example

```
dbName
    " See superimplementor's comment "
    ^ 'sra'
```

- Add a new method in instance side of class **BioEntrezClient**, category "accessing public – databases" for example:

```
biosample
    " Set and answer the receiver's working database "

    ^ self database: (BioEntrezSample new client: self)
```

- Add a simple test case for the database name in the #testDbClassOf method, example:

```
    self assert: (BioEntrezDatabase dbClassOf: 'biosample') =
   BioEntrezSample.
```

The Files SubSystem

- A **BioFile** is used to represent a file, it's not intended to be used for parsing a file. This is useful when having a **BioCSVFile** which we need to set specific parsing paramenters, for example, numbers of columns to read or ignore header lines.
- A **BioFormatter** is what to use when a developer has an INPUT file and possible an OUTPUT file/stream. It is not necessarily a writer.

File Formats

File formats information is available through the class **BioFileFormat** which contains a registry of file formats information. Included information doesn't mean that the format is supported for reading or writing tough, it just acts as an internal small database of formats available. You may query if a format is registered by evaluating:

```
BioFileFormat isRegistered: 'CSV'.
BioFileFormat fileExtensionFor: 'FASTA'.
```

For example to register a new format for CSV files, evaluate;

```
BioFileFormat
    registerFormat: 'CSV'
    extensions: #('csv')
    description: 'General ''Comma'' Separated Values format'
    rootClass: BioCSVFile
```

The root class is not necessary only for registration purposes, so for now you can pass nil as parameter. In the future a GUI wizard will be provided to generate code automatically for registering and unregistering file formats.

Community

Mailing lists

A mailing list for users is available as biosmalltalk-users.

A mailing list for developers is available as biosmalltalk-developers.

Beware that posts are (or will be) publicly available trhough popular search engines and mailing lists archives. If you want to create another list for a particular package or subsystem, contact the project manager through the project page.

To ask questions please use the ettiquete suggested at: http://www.catb.org/~esr/faqs/smart-questions.html