Bioperl course

by Catherine Letondal and Katja Schuerer

Bioperl course [http://www.pasteur.fr/recherche/unites/sis/formation/biope

by Catherine Letondal and Katja Schuerer

Published March, 14 2005

Introduction to Bioperl (www.bioperl.org [http://www.bioperl.org/]). This course introduces to the bioperl modules with examples and exercises. The course content has been upgraded to bioperl 1.0 (differences with bioperl version 0.7 are displayed in yellow color in the diagrams). Apart from this upgrade, the presentation has been reorganized by topics, rather than by daily schedule, without any major changes in the content. A PDF [support.pdf] version is now available.

Contact: help@pasteur.fr [mailto:help@pasteur.fr]

Comments are welcome.

Table of Contents

List of Figures

List of Examples

List of Exercises

Chapter 1. General introduction

1.1. Bioperl documentation

1. Bioperl tutorial [http://www.bioperl.org/Core/Latest/bptutorial.html]
2. Mastering Perl for Bioinformatics: Introduction to bioperl [http://www.oreilly.com/catalog/mperlbio/chapter/ch09.pdf]
3. Bioperl documentation [http://www.bioperl.org/Core/Latest/modules.html]
4. Modules listing. [http://doc.bioperl.org/releases/bioperl-1.0.1/]
5. http://doc.bioperl.org/bioperl-live/,¯ [http://doc.bioperl.org/bioperl-live/]
6. Unix help: man and perldoc.

1.2. General bioperl classes

General bioperl classes [http://www.bioperl.org/images/bioperl.pdf] diagram (cf Figure 6.1 for UML symbols).

Chapter 2. Sequences

2.1. The `Bio::SeqIO` class

2.1.1. Format Converter

Example 2.1. SwissProt -> Fasta

pseudo-code:

- Construct a *SwissProt* formatted sequences input stream
- Construct a *fasta* formatted sequences output stream
- for all sequences:
  - read the sequence from input stream
  - write it to output stream

in bioperl:

```
#! /local/bin/perl -w

use strict;

use Bio::SeqIO;


my $in  = Bio::SeqIO->newFh ( -file   => '<seqs.sp',
          -format => 'swiss' );

my $out = Bio::SeqIO->newFh ( -file   => '>seqs.fasta',
         -format => 'fasta' );


print $out $_ while <$in>;
```

data: seqs.sp [data/seqs.sp]

Exercise 2.1. Bio::SeqIO

Find all available formats via the `Bio::SeqIO` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/SeqIO.html] class (Solution A.1).

Look at the general description in `Bio::SeqIO` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/SeqIO.html] to understand `-file` and `-fh` parameters better,

Exercise 2.2. An universal converter

Modify Example 2.1 to program an universal converter (Solution A.2).

Exercise 2.3. An universal converter using new

Modify Exercise 2.2 to use this form of IO (Solution A.3):

```
use Bio::SeqIO;

my $in  = Bio::SeqIO->new(-file => "inputfilename" , '-format' => 'Fasta');
my $out = Bio::SeqIO->new(-file => ">outputfilename" , '-format' => 'EMBL');

my $seq = $in->next_seq() ;
$out->write_seq($seq);
```

2.2. Sequence classes

2.2.1. Introduction

Example 2.2. Loading a sequence from a remote server

pseudocode:

- Create a sequence object for the `MALK_ECOLI` *SwissProt* entry
- Print its Accession number and description

in bioperl:

```
use Bio::DB::SwissProt;

my $database = new Bio::DB::SwissProt;
my $seq = $database->get_Seq_by_id('MALK_ECOLI');

print "Seq: ", $seq->accession_number(), " -- ", $seq->desc(), "\n\n";
```

Exercise 2.4. Display a sequence in fasta format

Display the `MALK_ECOLI` entry in *fasta* format (Solution A.4).

2.2.2. Building mechanisms summary

de novo:

```
use Bio::Seq;

my $seq1 = Bio::Seq->new ( -seq  => 'ATGAGTAGTAGTAAAGGTA',
                           -id   => 'my seq',
                           -desc => 'this is a new Seq');
```

from a file:

```
use Bio::SeqIO;

my $seqin = Bio::SeqIO->new ( -file   => 'seq.fasta',
                              -format => 'fasta');
my $seq3 = $seqin->next_seq();

my $seqin2 = Bio::SeqIO->newFh ( -file   => 'seq.fasta',
                                 -format => 'fasta');
my $seq4 = <$seqin2>;
my $seqin3 = Bio::SeqIO->newFh ( -file   => 'golden sp:TAUD_ECOLI |',
                                 -format => 'swiss');
my $seq5 = <$seqin3>;
```

by fetching an entry from a remote database:

```
use Bio::DB::SwissProt;

my $banque = new Bio::DB::SwissProt;
my $seq6 = $banque->get_Seq_by_id('KPY1_ECOLI');
```

by fetching an entry from a bioperl indexed local database:

```
use Bio::Index::Swissprot;
my $inx = Bio::Index::Swissprot->new( -filename => 'small_swiss.inx');
my $seq7 = $inx->fetch('MALK_ECOLI');
```

data: small_swiss.inx [data/small_swiss.inx]

2.2.3. A deeper insight into the Bio::Seq class

Figure 2.1 shows the class structure of the `Bio::Seq` class. The `Bio::Annotation` package and `Bio::SeqFeature` package are more detailed in Figure 2.3 and Figure 2.5. Figure 2.2 shows the relation between a SwissProt entry and the corresponding `Bio::Seq` components.

Figure 2.1. Bio::Seq class structure

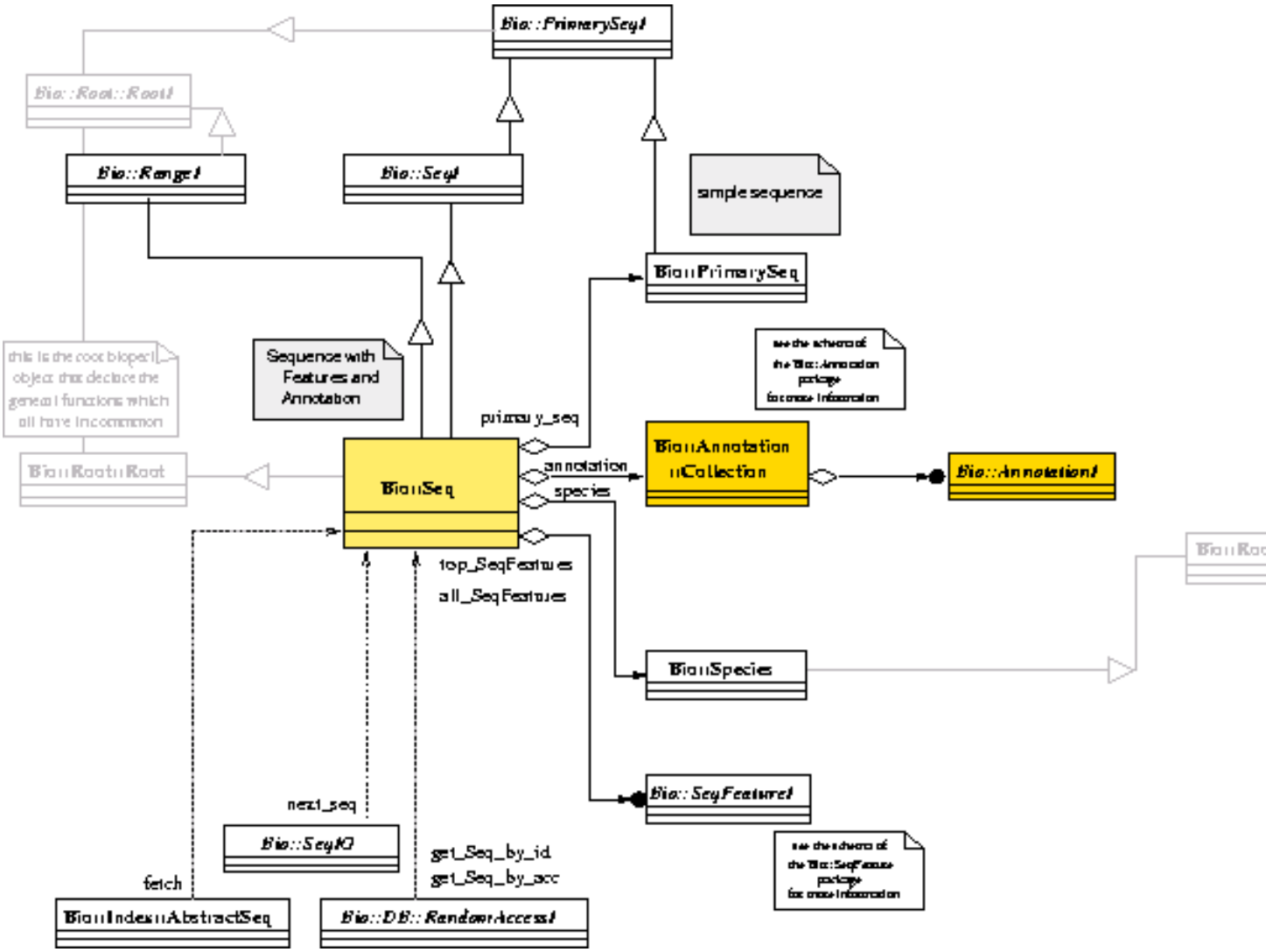


Figure 2.2. Relation of a SwissProt entry and the corresponding Bio::Seq object components

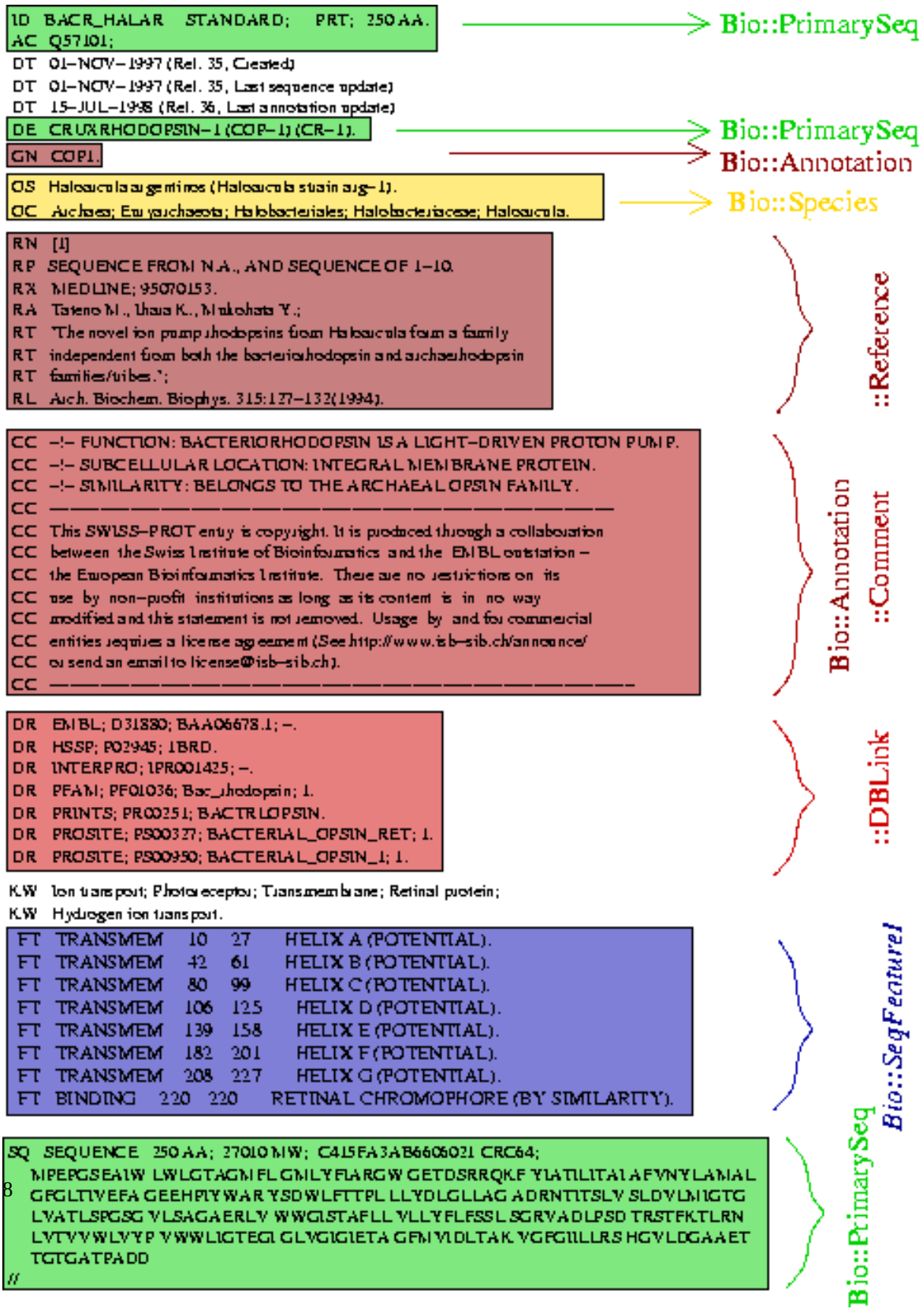
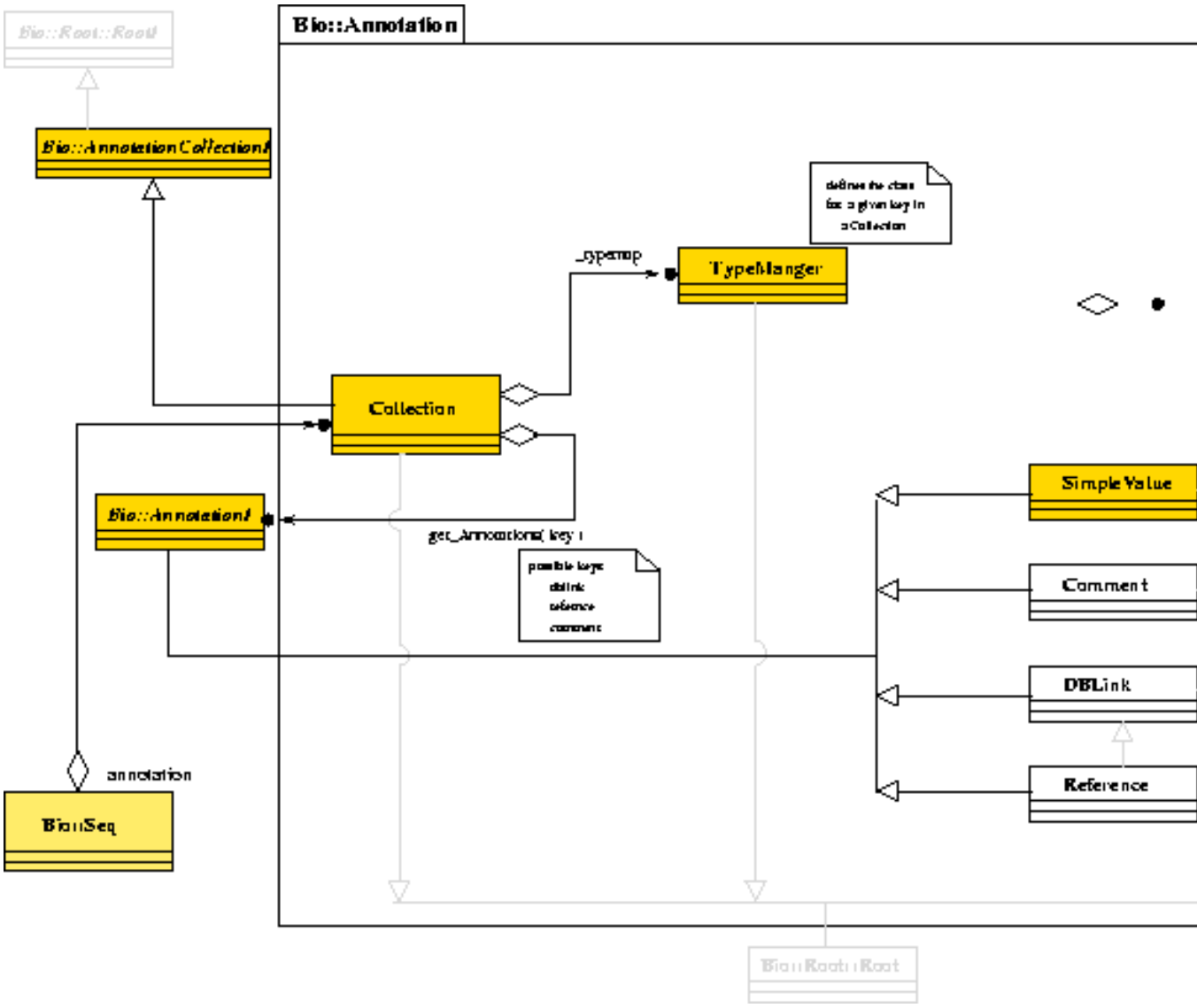


Figure 2.3. Bio::Annotation package structure



Example 2.3. Find the references to the PDB database entries

Does the protein have a known structure, and, if so, what are its cristallographic structures?

Where can you find this information within a SwissProt entry?

Where can you find this information in the sequence object?

in bioperl:

```
# PDB structures entries

$annotation = $seq->annotation();
my @structures = ();
foreach my $link ( $annotation->get_Annotations('dblink') ) {
    if ($link->database() eq 'PDB') {
        push (@structures, $link->primary_id());
    }
}
print "\nPDB Structures: ", join (" ", @structures), "\n";
```

Try this on the SwissProt 'TAUD_ECOLI' entry.

Exercise 2.5. More on annotations

Find the function of the protein within comments (if available) (Solution A.5) (for the MALK_ECOLI entry for instance). The comment field looks like the following:

```
-!- FUNCTION: Catalyzes the conversion of taurine and alpha
    ketoglutarate to sulfite, aminoacetaldehyde and succinate.
    Required for the utilization of taurine (2-aminoethanesulfonic
    acid) as an alternative sulfur source. Pentane-sulfonic acid, 3-
    (N-morpholino)propanesulfonic acid and 1,3-dioxo-2-
    isoindolineethanesulfonic acid are also substrates for this
    enzyme.
-!- CATALYTIC ACTIVITY: Taurine + 2-oxoglutarate + O(2) = sulfite +
    aminoacetaldehyde + succinate + CO(2).
-!- COFACTOR: Binds 1 iron(II) ion per subunit.
-!- ENZYME REGULATION: Activated by ascorbate and inhibited by
    divalent metal ions such as zinc, copper and cobalt.
-!- PATHWAY: Taurine catabolism.
-!- SUBUNIT: Homodimer.
-!- INDUCTION: Repressed by sulfate or cysteine.
-!- MISCELLANEOUS: Optimum pH for activity is 6.9.
-!- SIMILARITY: Belongs to the tfdA dioxygenase family.
-!- CAUTION: Ref.4 sequence differs from that shown due to
    frameshifts.
---------------------------------------------------------------------------
This SWISS-PROT entry is copyright. It is produced through a collaboration
between  the Swiss Institute of Bioinformatics  and the  EMBL outstation -
the European Bioinformatics Institute.  There are no  restrictions on  its
use  by  non-profit  institutions as long  as its content  is  in  no  way
modified and this statement is not removed.  Usage  by  and for commercial
entities requires a license agreement (See http://www.isb-sib.ch/announce/
or send an email to license@isb-sib.ch).
```

Tip

Split the comment on sentences (lines ended by a ’,’) and keep the first sentence containing the ’-!-FUNCTION:’ string.

Go to

Work on Exercise 2.7 before you continue.

Exercise 2.6. Transmembran helices

Find transmembrane helices in the entry (Solution A.6).

Tip

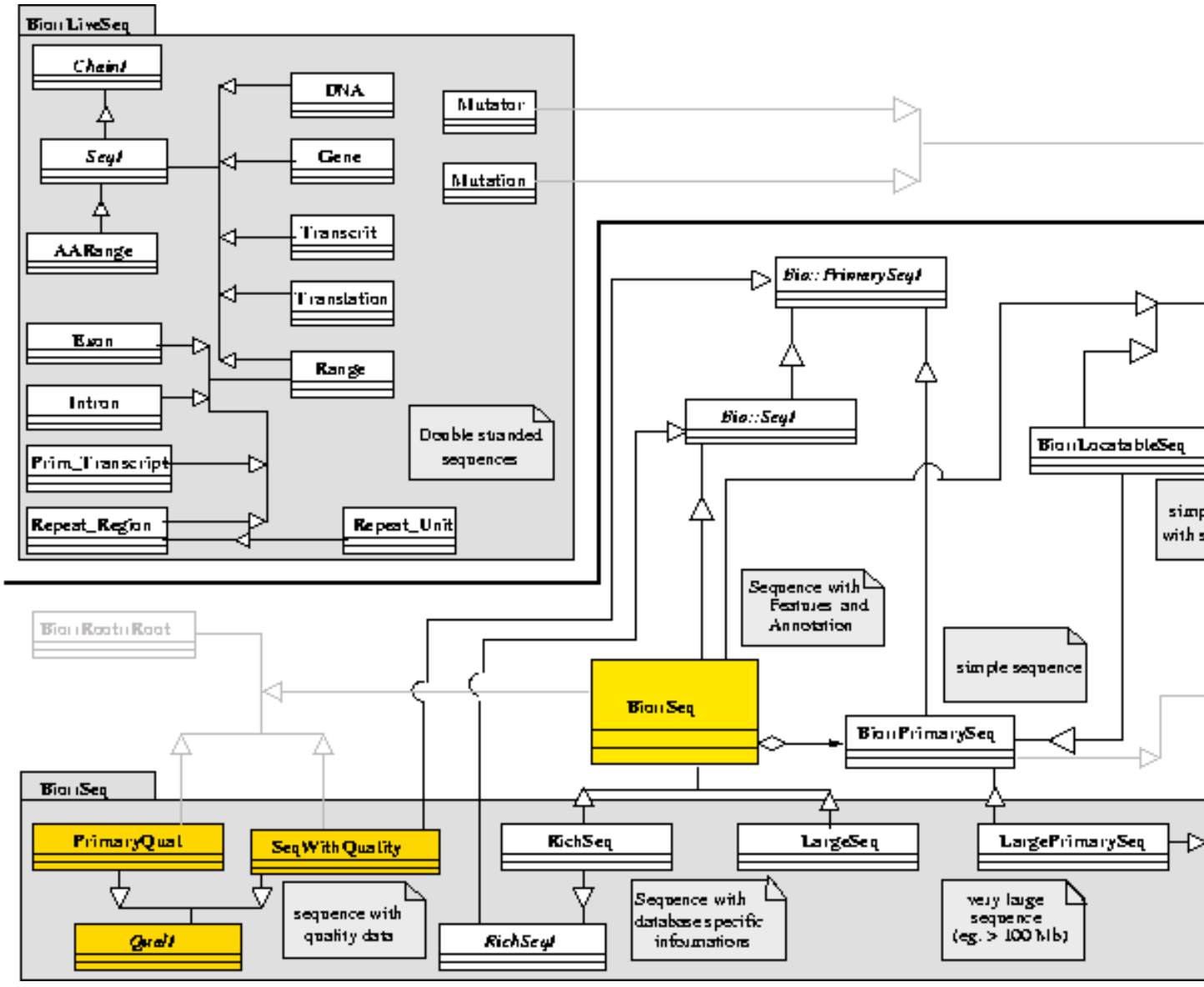
Use Figure 2.2 again.

Use the appropriate object’s documentation

Here is a summary [codes/useSeq1.0.pl] solution of the exercises in this section.

2.2.4. Bioperl ’Sequence’ classes structure

Figure 2.4. Bioperl 'Sequence' classes structure



2.3. Features and Location classes

2.3.1. Feature

They are created either by parsing database entries (see Bio::SeqIO::FTHelper [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SeqIO/FTHelper.html] or Figure A.1) or by parsing programs output. For instance (see Exercise 4.16), a Blast HSP is actually an instance of class Bio::SeqFeature::SimilarityPair [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SeqFeature/SimilarityPair.html], which is a sub-class of Bio::SeqFeature::FeaturePair [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SeqFeature/FeaturePair.html], which again is a feature. The same holds for Genscan predictions (Exercise 5.1).

Have a look at the `Bio::SeqFeatureI` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/SeqFeatureI.html] documentation, and try the example provided in the synopsis (in bioperl, examples in the synopsis section are generally supposed to work by just copy-and-paste :-)) (data: AB042770.gb [data/AB042770.gb]). Add some code to also display the sub-features, if any. Figure 2.5 shows the architecture of the `Bio::SeqFeature` class.

You can also create a feature manually, as shown in example Example 2.4.

Example 2.4. Adding a feature to a Genbank entry

```
use Bio::SeqFeature::Generic;
use Bio::SeqIO;

$in  = Bio::SeqIO->newFh(-file => $ARGV[0]);
$out = Bio::SeqIO->newFh();

$seq = <$in>;

$feat = new Bio::SeqFeature::Generic ( -start => 10, -end => 100,
           -strand => -1,
           -primary => 'repeat',
           -source => 'repeatmasker',
           -score  => 1000,
           -tag    => {
        new => 1,
        author => 'someone',
        sillytag => 'this is silly!' } );

$seq->add_SeqFeature($feat);


print $out $seq;
```

Try this code on the following GenBank entry: (AB042770.gb [data/AB042770.gb]). The file extension '.gb' should be left unchanged. Do you know why?

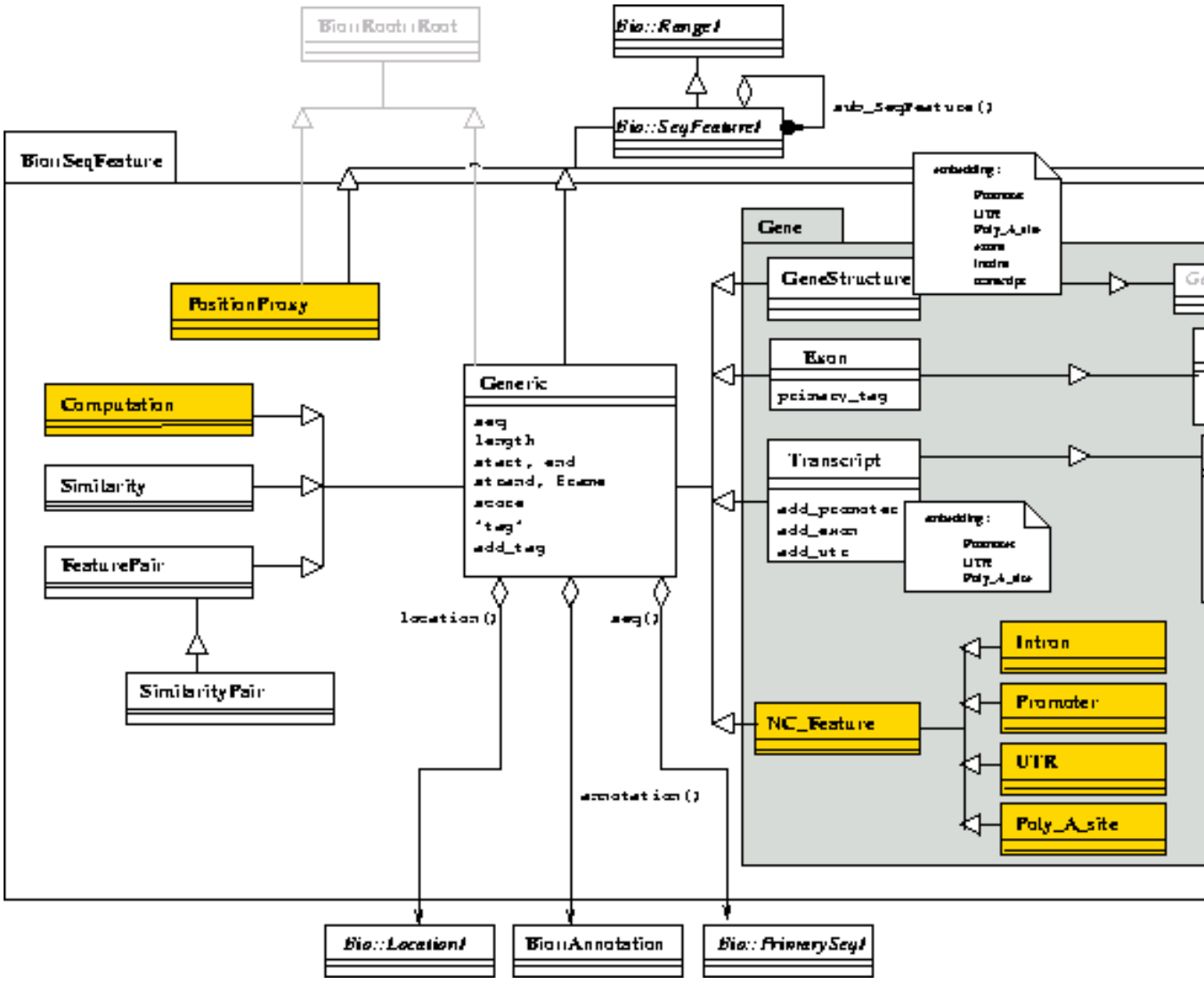
Documentation on features.

- Features table official documentation [http://www.ebi.ac.uk/embl/Documentation/FT_definitions/feature_table.html].
- bioperl tutorial [http://www.bioperl.org/Core/Latest/bptutorial.html#iii_7_1_representing_sequence_annotations__seqfeatu

Features Classes.

- Bio::SeqFeatureI
- Bio::SeqFeature::Generic
- Bio::SeqFeature::FeaturePair
- Bio::SeqFeature::SimilarityPair
- Bio::SeqFeature::Similarity
- Bio::SeqFeature::Gene::
- Bio::SeqFeature::Gene::ExonI
- Bio::SeqFeature::Gene::Exon
- Bio::SeqFeature::Gene::Transcript
- Bio::SeqFeature::Gene::TranscriptI
- Bio::SeqFeature::Gene::GeneStructureI
- Bio::SeqFeature::Gene::GeneStructure

Figure 2.5. Features Classes structure



2.3.2. Code reading: extracting CDS

Exercise 2.7. extractcds

Fetch an EMBL or Genbank entry and try this code [lecture_code/extractcds2] with, for instance, `gb:AB042770` (seq2.gb [data/seq2.gb]).

Web form [http://bioweb.pasteur.fr/seqanal/interfaces/extractcds.html]

Tip

Figure 2.6 shows the relation of features in an Embl entry and the `Bio::SeqFeature` class.

2.3.3. Tag system

This system enables you to refer to feature qualifiers (see documentation [http://www.ebi.ac.uk/embl/Documentation/FT_definitions/featu

In the following example, `CDS` (feature key) is a primary tag, `/codon` a secondary tag.

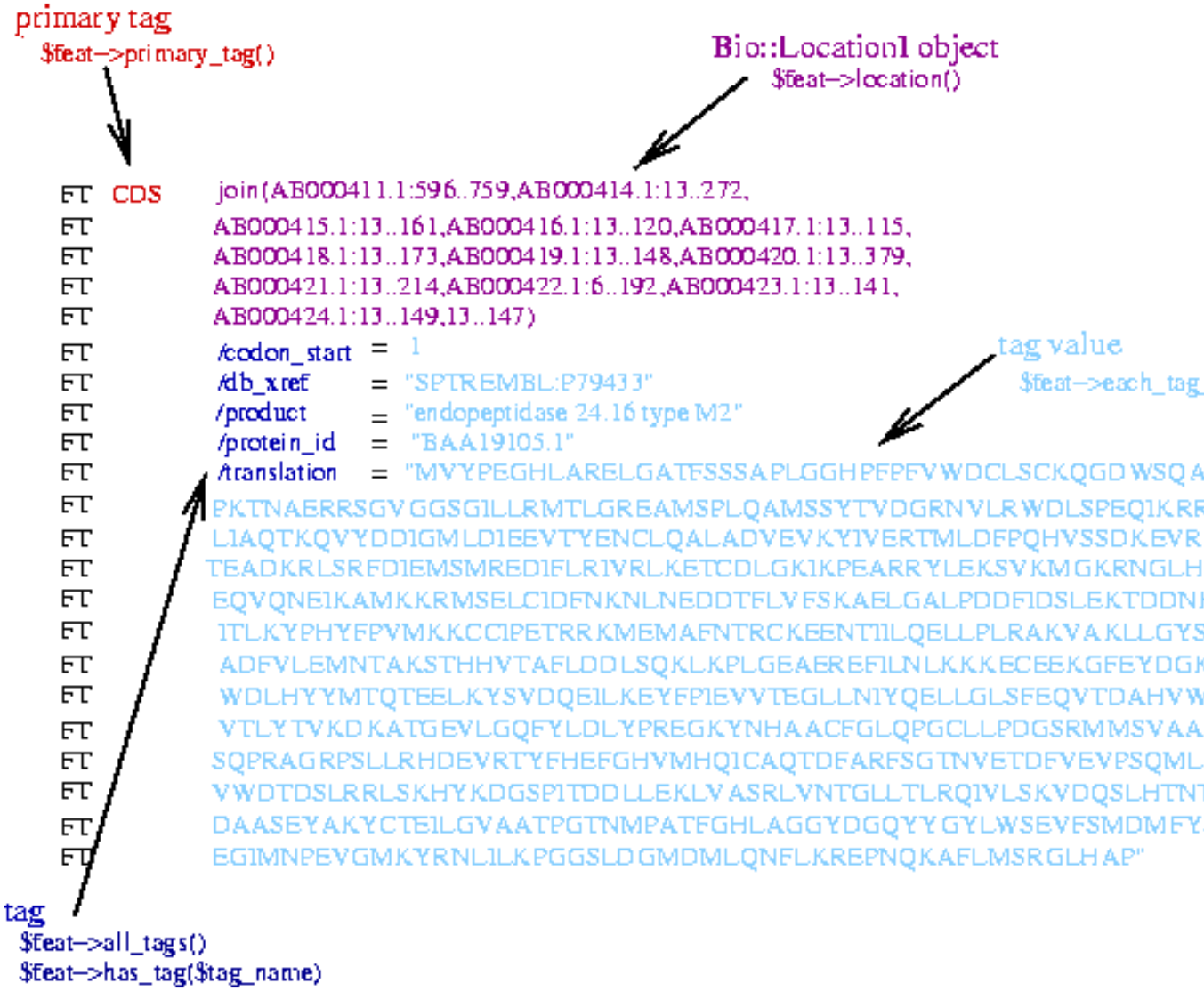
```
CDS             1..1000
                /codon=(seq:"cug",aa:Ser)
                /codon=(seq:"tga",aa:Trp)
```

The tag system also enables to create new qualifier types. Examples in this tutorial:

- Blast hits as features (see Figure 4.1 and Exercise 4.16).
- Creation of a database from a Genscan parsing (Exercise 5.1).

See methods related to tags in Bio::SeqFeature::Generic [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SeqFeature/Generic.html] documentation.

Figure 2.6. Correspondance between an EMBL entry and bioperl tags



2.3.4. Location

Locations are useful for everything that looks like a range within a sequence (cf `Bio::RangeI`): sequences associated to features, sequences belonging to an alignment...

Coordinates types. There are several coordinates types and representations: you can use fuzzy locations, ranges, lists, ... to specify the begin or the end (see the documentation

[http://www.ebi.ac.uk/embl/Documentation/FT_definitions/feature_table.html#3.5]); example taken from Bio::Location::Fuzzy [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Location/Fuzzy.html]:

```
my $fuzzylocation = new Bio::Location::Fuzzy(-start => '<30',
                                             -end   => 90,
                                             -loc_type => '.');
```

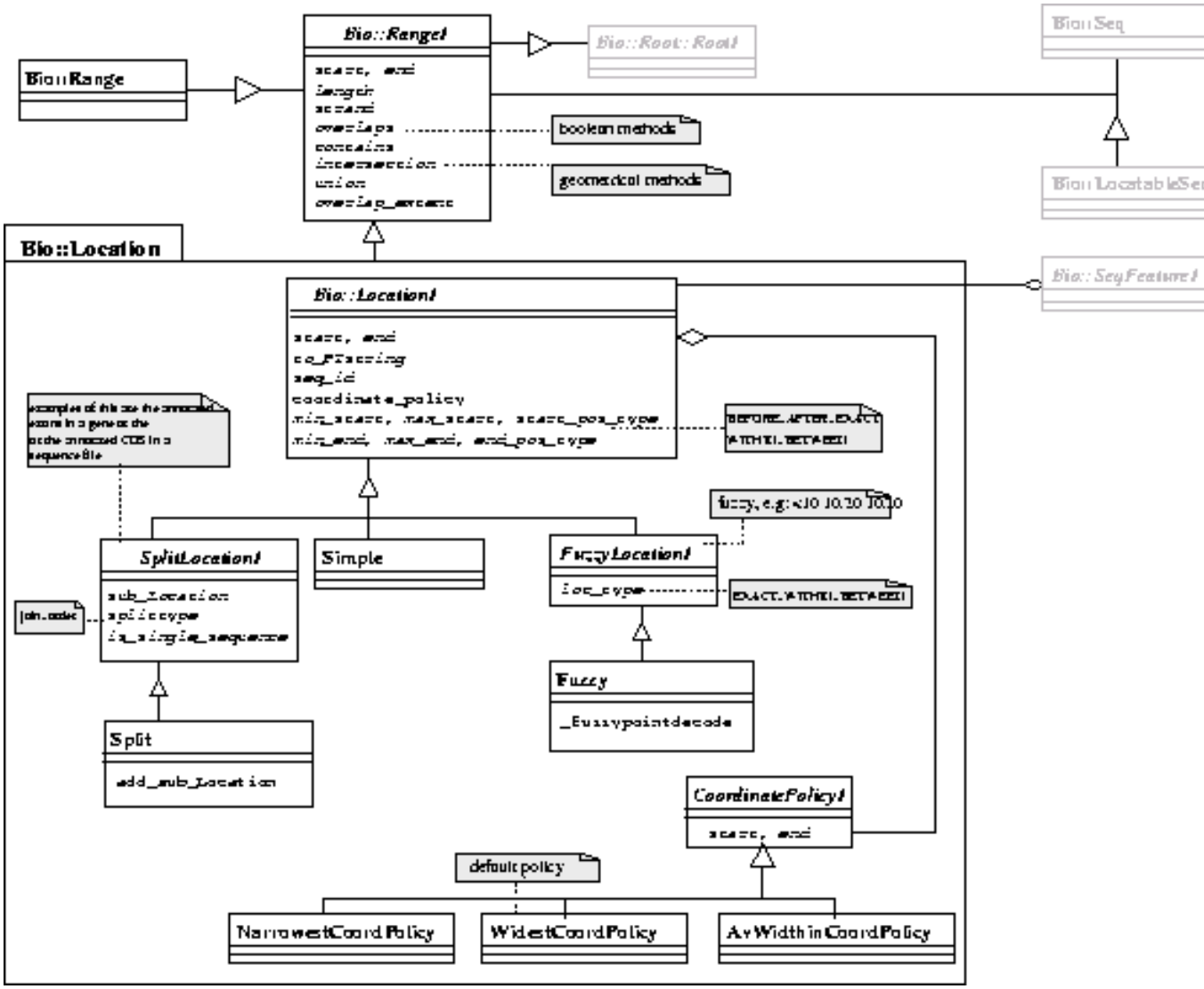
which specifies a location whose begin is before 30 and whose end is at 90. Coding:

- `'..'` : EXACT,
- `'^'` : BETWEEN (a site between 2 bases), exemple : `123^124` a site between bases 123 and 124. `145^177` a site between 2 adjacent bases, somewhere between bases 145 and 177.
- `'.'` : WITHIN (a base within a range), example : `(102.110)` is a site wihtin the 102 .. 110 range, inclusive.
- `'<'` : BEFORE
- `'>'` : AFTER

Location Classes.

- Bio::LocationI
- Bio::Location::Simple
- Bio::Location::SplitLocationIet Bio::Location::Split
- Bio::Location::FuzzyLocationIet Bio::Location::Fuzzy
- Coordinate policies, for fuzzy location start and end determination:
  - interface: Bio::Location::CoordinatePolicyI,
  - default: Bio::Location::WidestCoordPolicy
  - Bio::Location::NarrowestCoordPolicy, Bio::Location::AvWithinCoordPolicy

Figure 2.7. Location Classes Structure

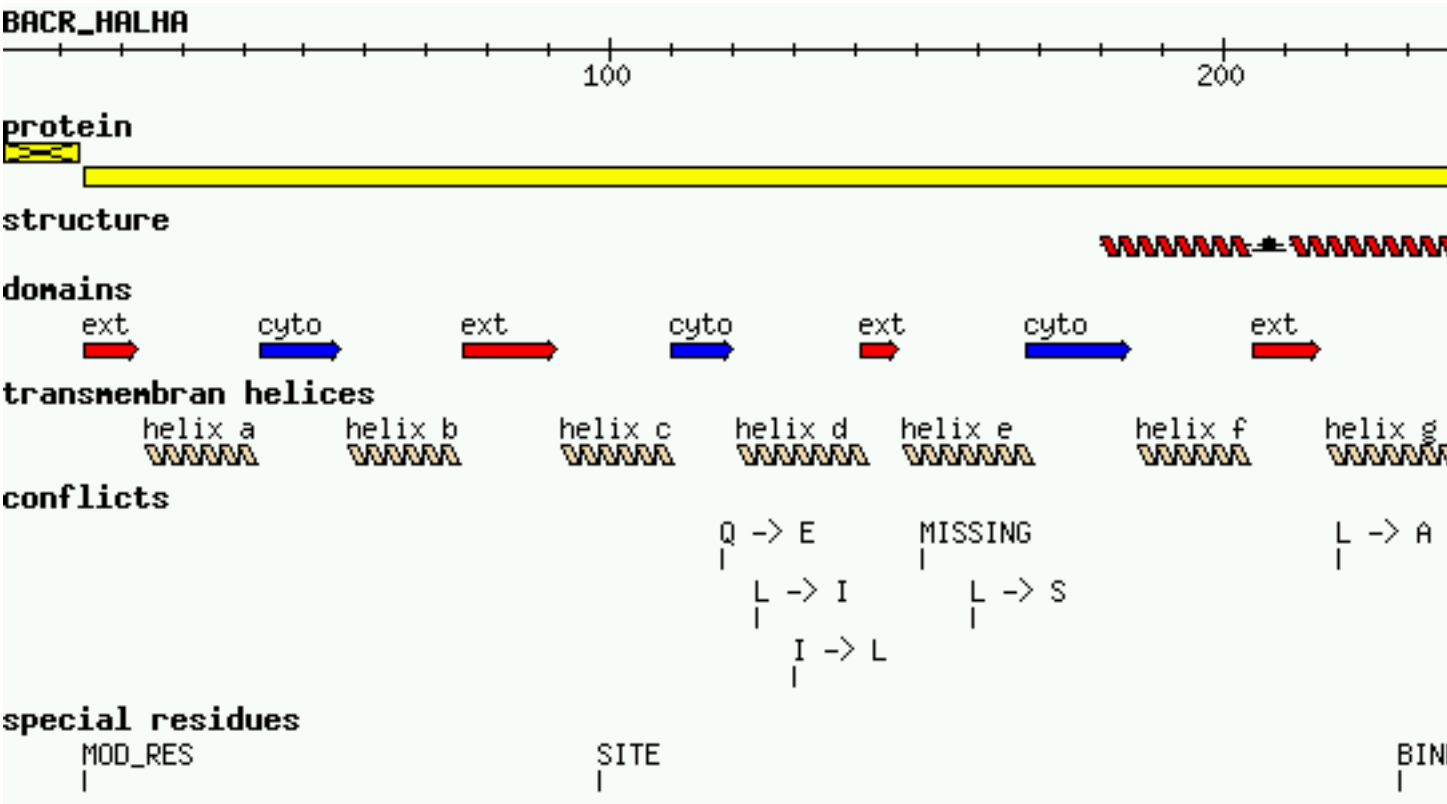


2.3.5. Graphical view of features

The version 1.0 of bioperl has a package to generate annotation images of sequences (see HOWTO [http://bioperl.org/HOWTOs/html/Graphics-HOWTO.html]). The general procedure is to create a `Bio::Graphics::Panel` object that holds the image and the sequence to annotate. Then you can add annotations line per line by using the `add_track` method of the panel object. The `Bio::Graphics::Glyph` package contains several prdefinded styles to show annotations but you also have the possibility to write your own glyphs.

Figure 2.8 shows some features of the SwissProt entry BACR_HALHA. Here [codes/feature_graph.pl] is the code that generates the image. The helix image and the arrow of domains are added glyph types that can be found here [codes/helix.pm].

Figure 2.8. Graphical view of some features of the SwissProt entry BACR_HALHA



2.4. Sequence analysis tools

`Bio::Seq` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/Seq.html] (tutorial [http://www.bioperl.org/Core/Latest/bptutoria
:.

```
$seqobj->seq;

# sequence as string
$seqobj->subseq(10,40);
# sub-sequence as string
$seqobj->trunc(10,100);
# sub-sequence as fresh Bio::PrimarySeq
```

```
$seqobj->translate;
```

`Bio::Tools::SeqStats` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/Tools/SeqStats.html] (tutorial [http://www.bioperl.org/Core/Latest/bptutorial.html#iii_3_2_obtaining_basic_sequence_statistics__seqstats_seqwo
: .

```
$seq_stats  =  Bio::Tools::SeqStats->new(-seq=>$seqobj);
$seq_stats->count_monomers();
$seq_stats->count_codons();
$weight = $seq_stats->get_mol_wt($seqobj);
```

et Bio::Tools::SeqWords [http:doc.bioperl.org/releases/bioperl-1.0.1/Bio/Tools/SeqWords.html//] :

```
$seq_word = Bio::Tools::SeqWords->new(-seq => $seqobj);
$seq_stats->count_words($word_length);
```

`Bio::Tools::SeqPattern` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/Tools/SeqPattern.html] (tutorial [http://www.bioperl.org/Core/Latest/bptutorial.html#iii_3_5_miscellaneous_sequence_utilities__oddcodes__seq
: .

```
$pat     = 'T[GA]AA...TAAT';
$pattern = new Bio::Tools::SeqPattern(-SEQ =>$pat, -TYPE =>'Dna');
$pattern->expand;
$pattern->revcom;
$pattern->alphabet_ok;
```

`Bio::SeqUtils` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/Tools/SeqUtils.html] : .

```
$util = new Bio::SeqUtils;
$polypeptide_3char = $util->seq3($seqobj);
```

or:

```
$polypeptide_3char = Bio::SeqUtils->seq3($seqobj);
```

`Bio::Tools::RestrictionEnzyme` [http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/Tools/RestrictionEnzyme.ht
(tutorial [http://www.bioperl.org/Core/Latest/bptutorial.html#iii_3_3_identifying_restriction_enzyme_sites__bio__rest
: .

```
$re1 = new Bio::Tools::RestrictionEnzyme(-NAME =>'EcoRI');
```

```
@fragments =  $re1->cut_seq($seqobj);
$re2 = new Bio::Tools::RestrictionEnzyme( -NAME =>'EcoRV--GAT^ATC', # creates a new one

                                          -MAKE =>'custom');
```

`Bio::Tools::Sigcleave` **[http://doc.bioperl.org/releases/bioperl-1.0.1/Bio/Tools/Sigcleave.html] (tutorial [http://www.bioperl.org/Core/Latest/bptutorial.html#iii_3_4_identifying_amino_acid_cleavage_sites__sigcleave_]) :.**

```
$sigcleave_object = new Bio::Tools::Sigcleave ('-file'=>'sigtest.aa',
          # not a Bio::Seq object!             '-threshold'=>'3.5',
                                              '-desc'=>'test sigcleave protein seq',
                                               '-type'=>'AMINO      ');
%raw_results = $sigcleave_object->signals;
$formatted_output = $sigcleave_object->pretty_print;
```

Chapter 3. Alignments

3.1. AlignIO

The Bio::AlignIO [http://doc.bioperl.org/releases/bioperl-1.0/Bio/AlignIO.html] class is designed the same as the `SeqIO` class. Therefore format conversions can be done as following:

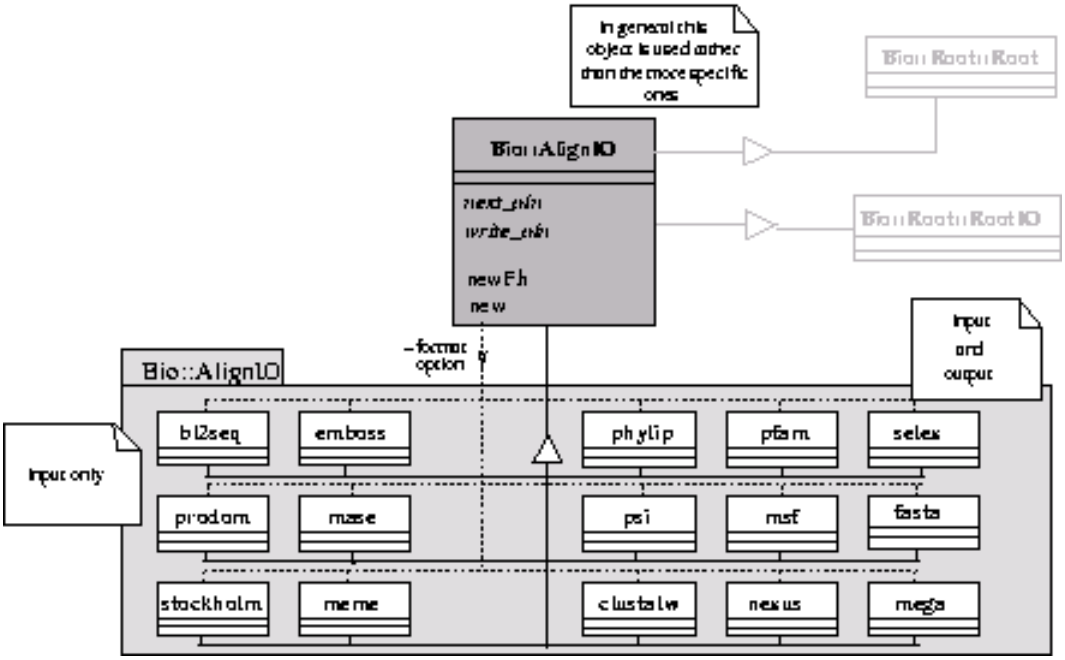
Example 3.1. Format conversions with AlignIO

(data [data/infile.aln])

```
#! /local/bin/perl -w

use strict;
use Bio::AlignIO;

my $inform = shift @ARGV || 'clustalw';
my $outform = shift @ARGV || 'fasta';

my $in  = Bio::AlignIO->newFh ( -fh => \*STDIN, -format => $inform );
my $out = Bio::AlignIO->newFh ( -fh => \*STDOUT, -format => $outform );

print $out $_ while <$in>;
```

`AlignIO` class structure. *Warning:* some alignment formats are available only as input formats.

Figure 3.1. AlignIO Classes diagram



3.2. SimpleAlign

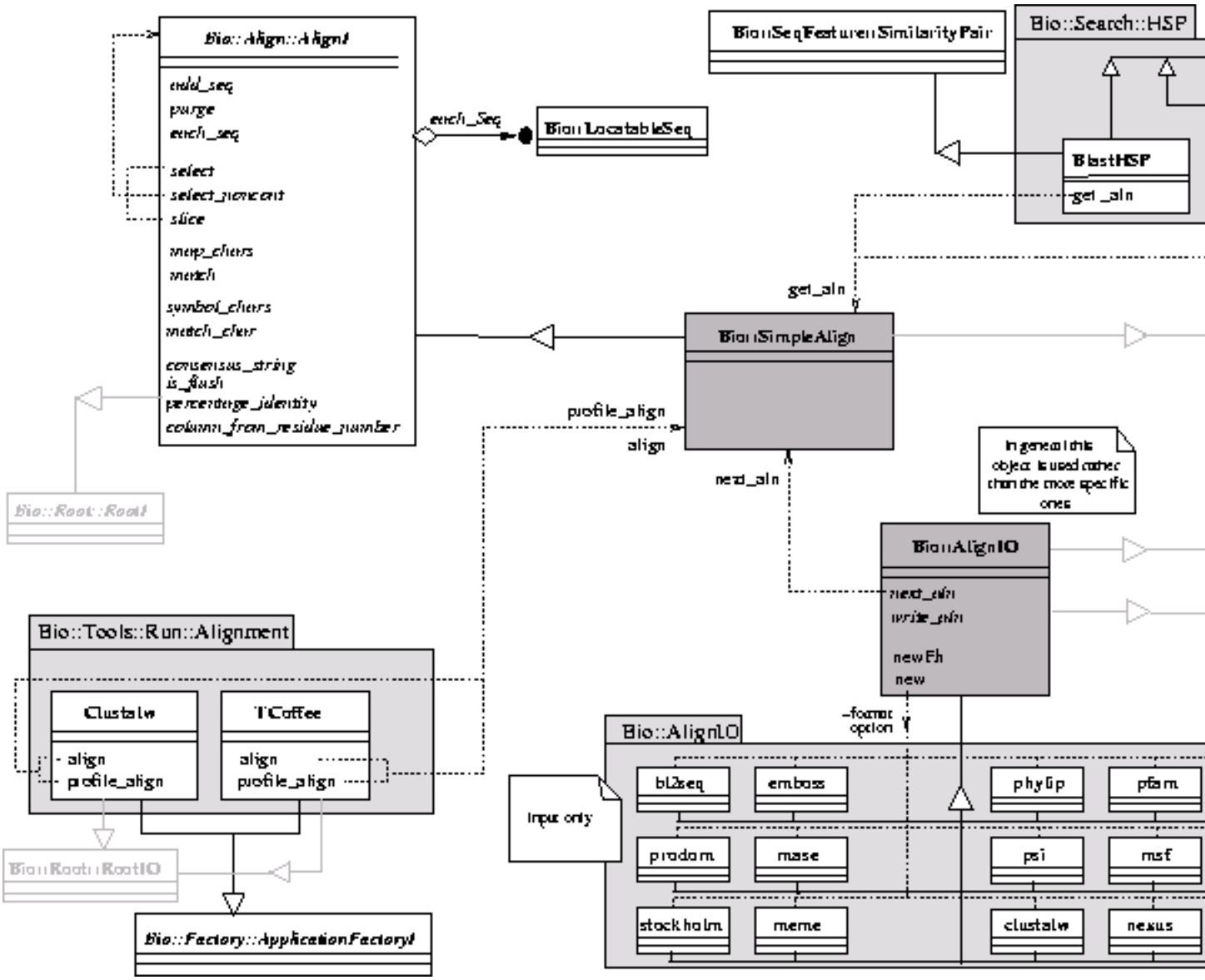
At first, let's see some basic methods of the `SimpleAlign` class:

Example 3.2. Basic methods of SimpleAlign

```
#! /local/bin/perl -w

use strict;
use Bio::AlignIO;

my $in = new Bio::AlignIO ( -file =>, $ARGV[0], -format => 'clustalw' );
my $aln = $in->next_aln();

print "same length of all sequences: ",
      ($aln->is_flush()) ? "yes" : "no", "\n";
print "alignment length: ", $aln->length(), "\n";
printf "identity: %.2f %%\n", $aln->percentage_identity();
printf "identity of conserved columns: %.2f %%\n", $aln->overall_percentage_identity();
```

SimpleAlign class diagram. It contains the `AlignI` interface that declares all methods to be implemented by Alignment classes. The diagram integrates also some classes that can create `SimpleAlign` objects:

Figure 3.2. Align Classes diagram



The `SimpleAlign` class contains methods to select sequences or columns, but it can not filter alignments by functions as could be done by the `UnivAln` class of the old bioperl release. In order to filter columns by

properties, you have to extract the columns by yourself, filter them and reconstruct the new sequences. The following example filters gap columns.

Example 3.3. Filter gap columns

```
#! /local/bin/perl -w

use strict;
use Bio::AlignIO;

my $in = new Bio::AlignIO ( -file => $ARGV[0], -format => 'clustalw' );
my $out = newFh Bio::AlignIO ( -fh => \*STDOUT, -format => 'clustalw' );

my $aln = $in->next_aln();

# if you need to work on the columns, create a list containing all columns as
# strings

my @aln_cols = ();

foreach my $seq ( $aln->each_alphabetically() ) {
  my $colnr = 0;
  foreach my $chr ( split("", $seq->seq()) ) {
    $aln_cols[$colnr] .= $chr;
    $colnr++;
  }
}

# then do the work:
# we want to eliminate all the columns containing gaps
# 1/ we create a list containing all the columns without any gap

my $gapchar = $aln->gap_char();
my @no_gap_cols = ();
foreach my $col ( @aln_cols ) {
 next if $col =~ /\Q$gapchar\E/;
 push @no_gap_cols, $col;
}

# 2/  we modify the sequences in the alignment
# 2a/ we reconstruct the sequence strings

my @seq_strs = ();
foreach my $col ( @no_gap_cols ) {
   my $colnr = 0;
   foreach my $chr ( split"", $col ) {
     $seq_strs[$colnr] .= $chr;
     $colnr++;    }}

# 2b/ we replace the old sequences strings with the new ones
```

```
foreach my $seq ( $aln->each_alphabetically() ) {
    $seq->seq(shift @seq_strs);}

print $out $aln;
```

Exercise 3.1. Create an alignment without gaps

Create a new alignment without gaps at the begining and the end of the alignment.

- sample alignment file [data/infile.aln]
- Hint: you don't have to extract all the columns as in the second example

Solution A.7

3.3. Code reading: protal2dna.

This script aligns DNA sequences, given their protein alignment.

- code [lecture_code/protal2dna.html] (download)
- bioperl classes:
  - Bio::AlignIO [http://doc.bioperl.org/releases/bioperl-1.0/Bio/AlignIO.html]
  - Bio::SimpleAlign [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SimpleAlign.html]
  - Bio::Tools::CodonTable [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/CodonTable.html]
- Web form [http://bioweb/seqanal/interfaces/protal2dna.html]

Chapter 4. Analysis

4.1. Blast

4.1.1. Running Blast

4.1.1.1. Running local Blast with Bio::Tools::Run::StandAloneBlast

Example 4.1. StandAloneBlast run

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0], -format => 'fasta');
my $query = $Seq_in->next_seq();
my $factory = Bio::Tools::Run::StandAloneBlast->new('program'  => 'blastp',
        'database' => 'swissprot',
       _READMETHOD => "Blast"
       );
my $blast_report = $factory->blastall($query);
my $result = $blast_report->next_result;

while( my $hit = $result->next_hit()) {
   print "\thit name: ", $hit->name(), " significance: ", $hit->significance(), "\n";}
```

You can try it with this file [data/1prot.fasta].

 **Exercise 4.1. Running Blast on a Swissprot entry**

Run the above example with the TAUD_ECOLI entry from Swissprot (see Section 2.2.2)).
Solution A.8

 **Exercise 4.2. Running Blast: Setting parameters**

Change the Expectation value (E) Blast parameter to 1.0 (instead of default value, 10.0) (query [data/1prot.fasta]).
Solution A.9

Exercise 4.3. Running Blast: Saving output

Save the blast report in a local file (e.g `blast.out`).
Solution A.10

Exercise 4.4. Running a Remote Blast

Find out from the bioperl Web site documentation [http://doc.bioperl.org/bioperl-live/] how to run a Blast at NCBI.
Solution A.11

4.1.2. Parsing Blast

4.1.2.1. Parsing from a Blast run result.

pseudocode:

- Run Blast.
- Create a report.
- For all hits (class Bio::Search::Hit::GenericHit [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Search/Hit/GenericHit.html])
  - print its name and signifiance
  - for all HSP (class Bio::Search::HSP::GenericHSP [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Search/HSP/GenericHSP.html])
    - print some hit statistics

Example 4.2. StandAloneBlast parsing

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;
my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
                              -format => 'fasta');
my $query = $Seq_in->next_seq();
my $factory = Bio::Tools::Run::StandAloneBlast->new( 'program'  => 'blastp',
    'database' => 'swissprot',
    _READMETHOD => "Blast" );
my $blast_report = $factory->blastall($query);
my $result = $blast_report->next_result;

while( my $hit = $result->next_hit()) {
  print "\thit name: ", $hit->name(), " significance: ", $hit->significance(), "\n";

  while( my $hsp = $hit->next_hsp()) {
    print "E: ", $hsp->evalue(), "frac_identical: ", $hsp->frac_identical(), "\n";    }}
```

`$result` belongs to the Bio::Search::Result::GenericResult class, where you can find the documentation about available methods on results.

Exercise 4.5. Display Blast hits

Display hit accession and hsp length.
Solution A.12

Exercise 4.6. Class of a Blast report

What is the class of `$blast_report`? (hint: use the `ref()` perl function).
Solution A.13

4.1.2.2. Parsing from a file.

The `$blast_report` that is created by the Bio::Tools::Run::StandAloneBlast new method actually belongs to the `Bio::SearchIO` class (see HOWTO [http://bioperl.org/HOWTOs/SearchIO/index.html]). So, you can directly create such an object from a file.

Example 4.3. Parsing from a Blast file

```
use Bio::SearchIO;
```

```
my $blast_report = new Bio::SearchIO (’-format’ => ’blast’,
                 ’-file’   => $ARGV[0]);
my $result = $blast_report->next_result;

while( my $hit = $result->next_hit()) {
  print "\thit name: ", $hit->name(), "\n";
  while( my $hsp = $hit->next_hsp()) {
    print "E: ", $hsp->evalue(), "frac_identical: ", $hsp->frac_identical(), "\n";    }}
```

Exercise 4.7. Parse a Blast output file

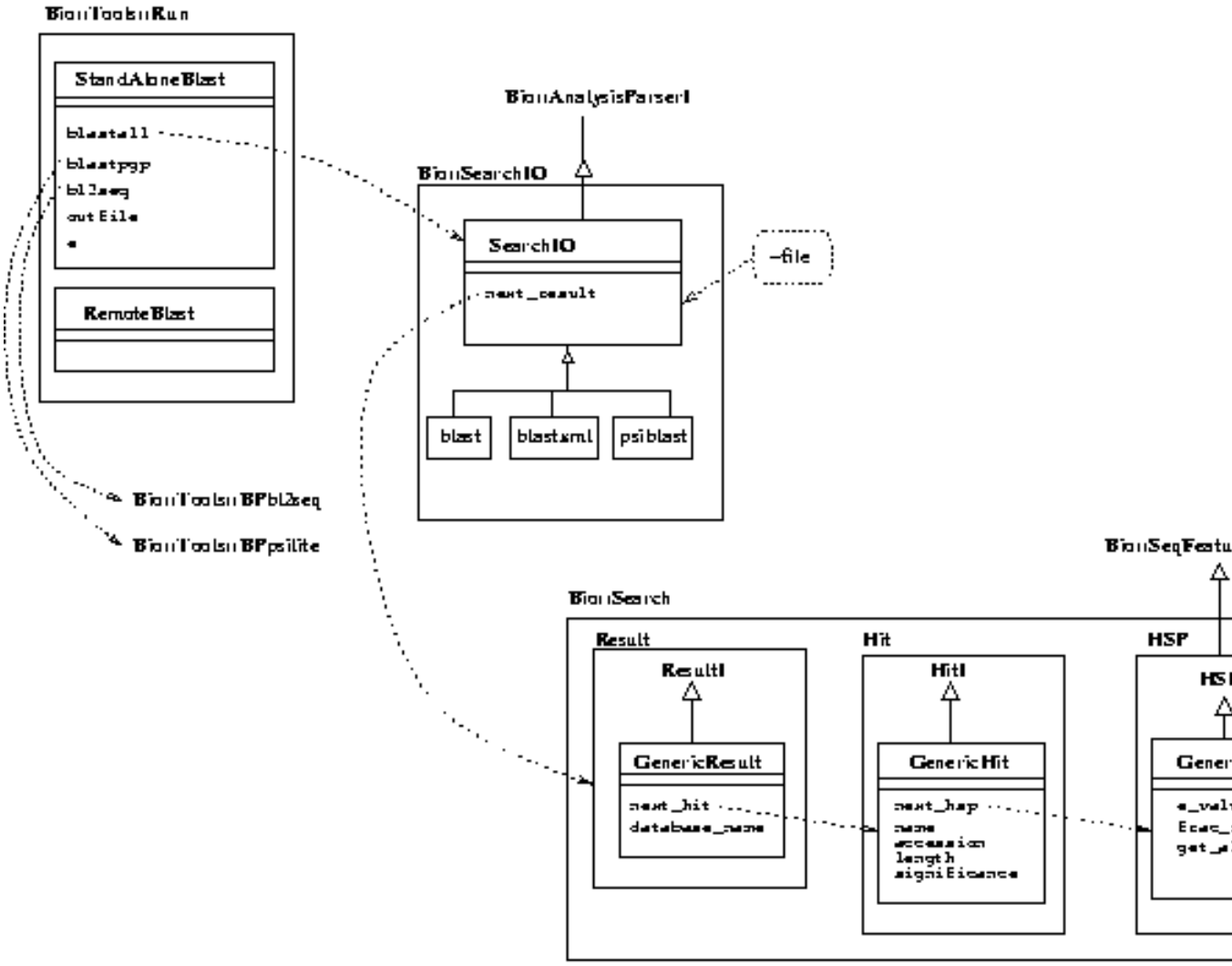
Run Example 4.3 with the file you have just created in Exercise 4.3.

Exercise 4.8. Parse Blast results on standard input

Run Example 4.3 with the blast report given on standard input.
Solution A.14

4.1.2.3. Blast Classes diagram

Figure 4.1. Blast Classes diagram



4.1.2.4. Filtering Blast output

Exercise 4.9. Filtering hits by length

Only display hits of length > to a given value (create a blast report from this file [data/blast.out]).
Solution A.15

Exercise 4.10. Filtering hits by position

Only display hits between two positions.
Solution A.16

Exercise 4.11. Display the best hit by databank

Display the best hit for each database in a Blast NCBI report (example [data/ncbi-blast.out]).

- If necessary, you can find here a `dbname(hit->name)` [solution/dbname.pl] function to extract database name from the hit name.
- You may use a perl hash to mark a database as already done:

```
$done{$database} = 1;
```

Solution A.17

Exercise 4.12. Multiple queries

Submit two sequences and display the best hit for each. (2nd query [data/P42704.fasta])
Solution A.18

4.1.2.5. Extracting data from a Blast report

Exercise 4.13. Extracting the subject sequence

Extract as a string the subject sequence from HSP with identity above a given level.
Solution A.19

Exercise 4.14. Extracting alignments

Extract HSP alignment from hits which name contains a given pattern, and print this alignment in Clustalw format.
Solution A.20

Exercise 4.15. Locate EST in a genome

Locate EST in a genome:

- displays the occurrences of this EST [data/est] in this contig [data/contig] of a genome;
- You may first display this as a Clustalw alignment (on sequence by HSP);
- Blast report [data/blast-est.out] ;

Solution A.21

Exercise 4.16. Record Blast hits as sequence features

Iterate a local Blast on databases given on the command line and record each corresponding best hit as a feature for the query sequence.
Solution A.22

4.1.3. Bio::Tools::BPlite family parsers

The old Bio::Tools::BPlite [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPlite.html] family of parsers (BPlite [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPlite.html], Bio::Tools::BPpsilite [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPpsilite.html], Bio::Tools::BPbl2seq [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPbl2seq.html]) is still maintained in the 1.0 release, although is will be deprecated one day. It is actually still necessary for blasting two sequences and to perform a psiblast (Position Specific Iterative Blast). It is also still the default parser returned by Bio::Tools::Run::StandAloneBlast, as shown below (although this might change soon).

Example 4.4. Parsing with BPLite

In the following example (notice that we have removed the `_READMETHOD` parameter), the `$blast_report` does not belong to the Bio::SearchIO [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SearchIO.html] class, but to the Bio::Tools::BPlite [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPlite.html] class. That is why the method to use in order to get informations are different, namely nextSbjct [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPlite.html#POD8] and nextHSP [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPlite/Sbjct.html#POD4] of class Bio::Tools::BPlite::Sbjct [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPlite/Sbjct.html] instead of `next_hit` and `next_hsp`. There is no `next_result` method, since these parsers were not able to deal with multiple reports.

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
         -format => 'fasta');
my $query = $Seq_in->next_seq();;

my $factory = Bio::Tools::Run::StandAloneBlast->new(
          'program'  => 'blastp',
          'database' => 'swissprot'
          );
my $blast_report = $factory->blastall($query);
while (my $subject = $blast_report->nextSbjct()) {
    print $subject->name(), "\n";
    while (my $hsp = $subject->nextHSP()) {
 print join("\t",
     $hsp->P,
     $hsp->percent,
```

```
     $hsp->score), "\n";
    }
}
```

Exercise 4.17. Print informations from a BPLite report

Take Example 4.4 and print hit names, and for all HSP, its P value, percent identity and score (see Bio::Tools::BPlite::HSP).
Solution A.23

Exercise 4.18. Create a Bio::Tools::BPlite from a file

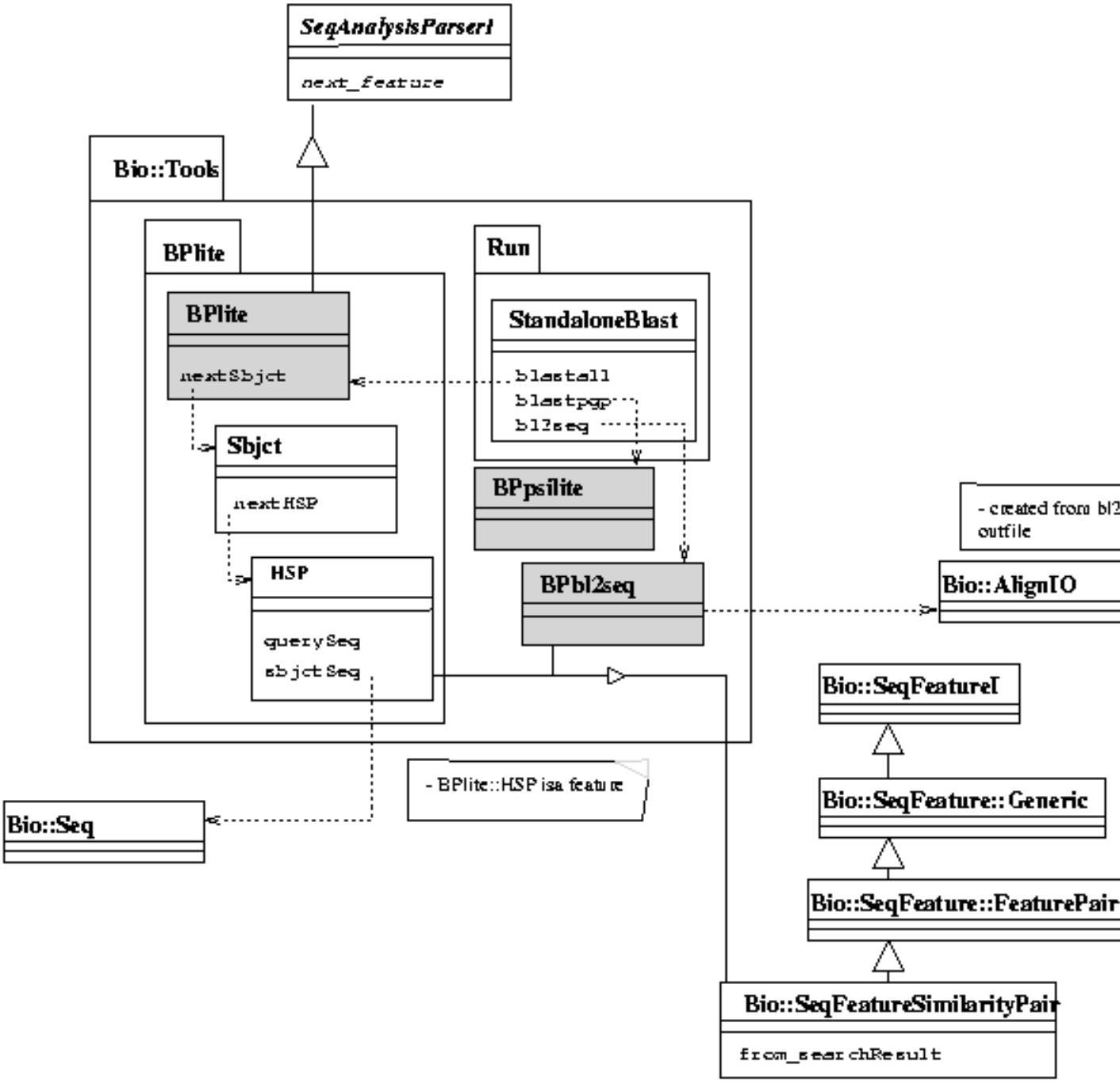
Create a Bio::Tools::BPlite from a file.
Solution A.24

Exercise 4.19. Create a Bio::Tools::BPlite from standard input

Create a Bio::Tools::BPlite from standard input.
Solution A.25

Chapter 4. Analysis

Figure 4.2. BPLite Classes diagram



4.1.4. PSI-BLAST (Position Specific Iterative Blast)

See documentation [http://www.ncbi.nlm.nih.gov/BLAST/tutorial/Altschul-2.html] or tutorial [http://www.ncbi.nlm.nih.gov/Ed
at NCBI.

Example 4.5. Running PSI-blast

The following example run a PSI-blast:

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $query_in  = Bio::SeqIO->newFh ( -file   => $ARGV[0],
        -format => 'fasta' );

my $query = <$query_in>;
$factory = Bio::Tools::Run::StandAloneBlast->new('database' => 'swissprot');
$factory->j(2);
$blast_report = $factory->blastpgp($query);

for my $iteration (1 .. $blast_report->number_of_iterations()) {
  print "iteration: $iteration\n";
  my $result = $blast_report->round($iteration);

  while( my $hit = $result->nextSbjct()) {
    print "\thit name: ", $hit->name(), "\n";
    foreach my $hsp ($hit->nextHSP) {
      print "\tstart: ", $hsp->hit->start(), " end: ",$hsp->hit->end(),"\n";
      print "\tscore: ", $hsp->score(), "\n"; }    }}
```

You can also load a report from a file, or from standard input:

```
my $blast_report = Bio::Tools::BPpsilite->new(-fh=>\*FH);
```

Example 4.6. PSI-Blast SearchIO class

There is also a `SearchIO` class for PSI-blast report, but it does not give access to the specific iterations:

```
use Bio::SearchIO;

my $in = Bio::SearchIO->new( -format  => 'psiblast',
           -file    => $ARGV[0] );

while ( my $result = $in->next_result() ) {
  foreach my $hit ( $result->hits ) {
```

```
    print "Hit: $hit\n"; }}
```

Exercise 4.20. Parse a PSI-blast report

Load a PSI-blast report (example [data/psiblast.out]) and at each iteration, display:

- the hits that were not present at the previous iteration
- the hits that have disappeared;

Solution A.26

4.1.5. bl2seq: Blast 2 sequences

Example 4.7. Running bl2seq

```
use Bio::SeqIO;use Bio::Tools::Run::StandAloneBlast;

my $query1_in  = Bio::SeqIO->newFh ( -file   => $ARGV[0],
        -format => 'fasta' );
my $query1 = <$query1_in>;
my $query2_in  = Bio::SeqIO->newFh ( -file   => $ARGV[1],
        -format => 'fasta' );
my $query2 = <$query2_in>;

$factory = Bio::Tools::Run::StandAloneBlast->new('program'  => 'blastp');
$report = $factory->bl2seq($query1, $query2);
print "ref: ", ref($blast_report), "\n";

while(my $hsp = $report->next_feature) {
  print "homology seq :\n",  $hsp->homologySeq, "\n";
  print "sbjctSeq :\n",  $hsp->sbjctSeq, "\n";}
```

The call to bl2seq will return a Bio::Tools::BPbl2seq [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/BPbl2seq.html] object.

Exercise 4.21. Build a Bio::SimpleAlign object

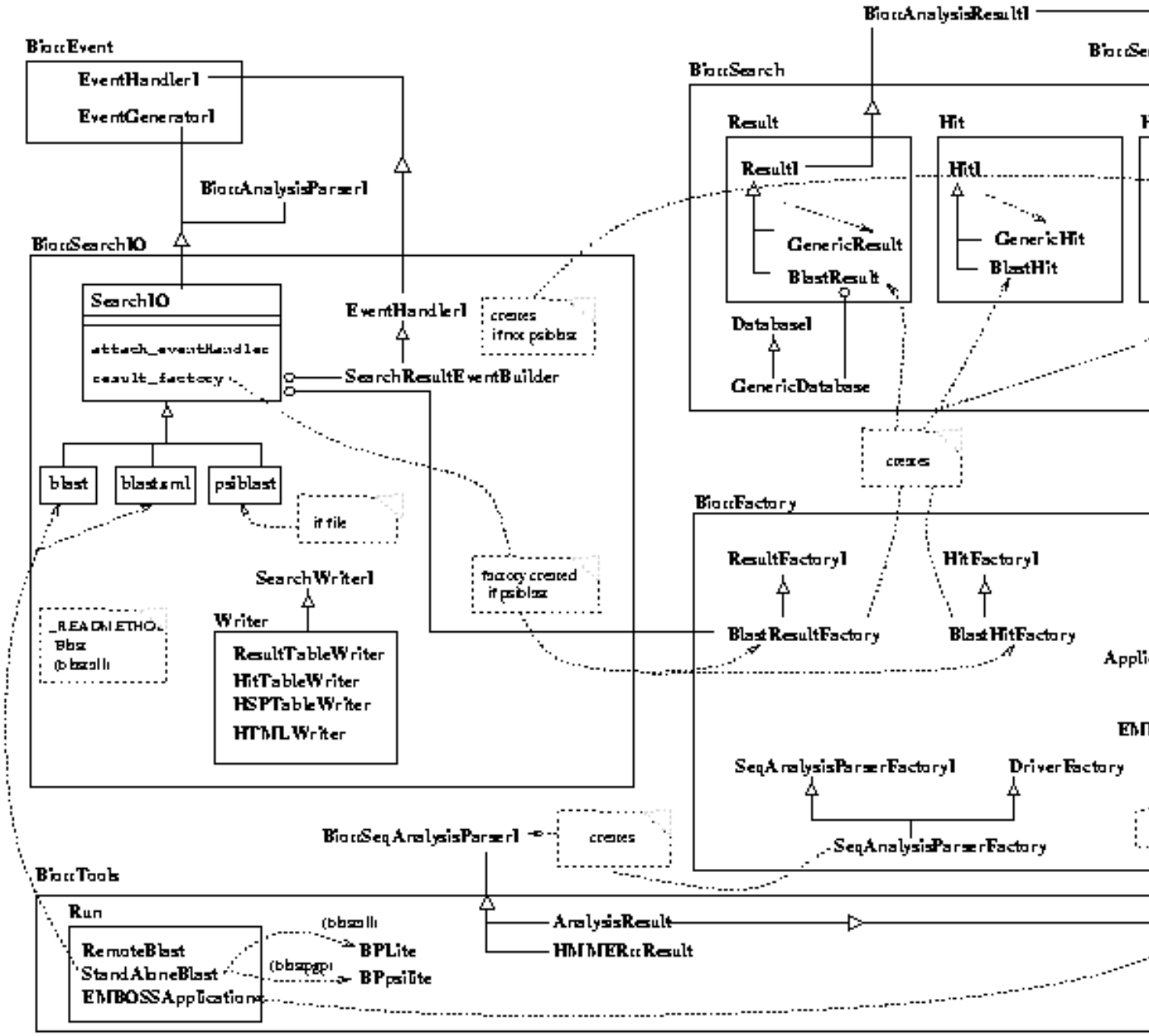
Build a `Bio::SimpleAlign` object from the subject and query sequences, and display this alignment in Clustalw format.

Solution A.27

4.1.6. Blast Internal classes structure

The following diagram shows the internal classes structure. It is provided for information, and it has not to be understood to use bioperl Blast classes.

Figure 4.3. Blast internal classes diagram



4.2. Genscan

Example 4.8. Genscan parsing

The folowing example shows how to use the Bio::Tools::Genscan [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Tools/Genscan.html] parser. (data: Genscan output file [data/genscan.out])

```
use Bio::Tools::Genscan;

my $genscan_file = $ARGV[0];
$genscan = Bio::Tools::Genscan->new(-file => $genscan_file);

while(my $gene = $genscan->next_prediction()) {
  my $prot = $gene->predicted_protein;
  print "protein (", ref($prot), ") :\n", $prot->seq, "\n\n";}
```

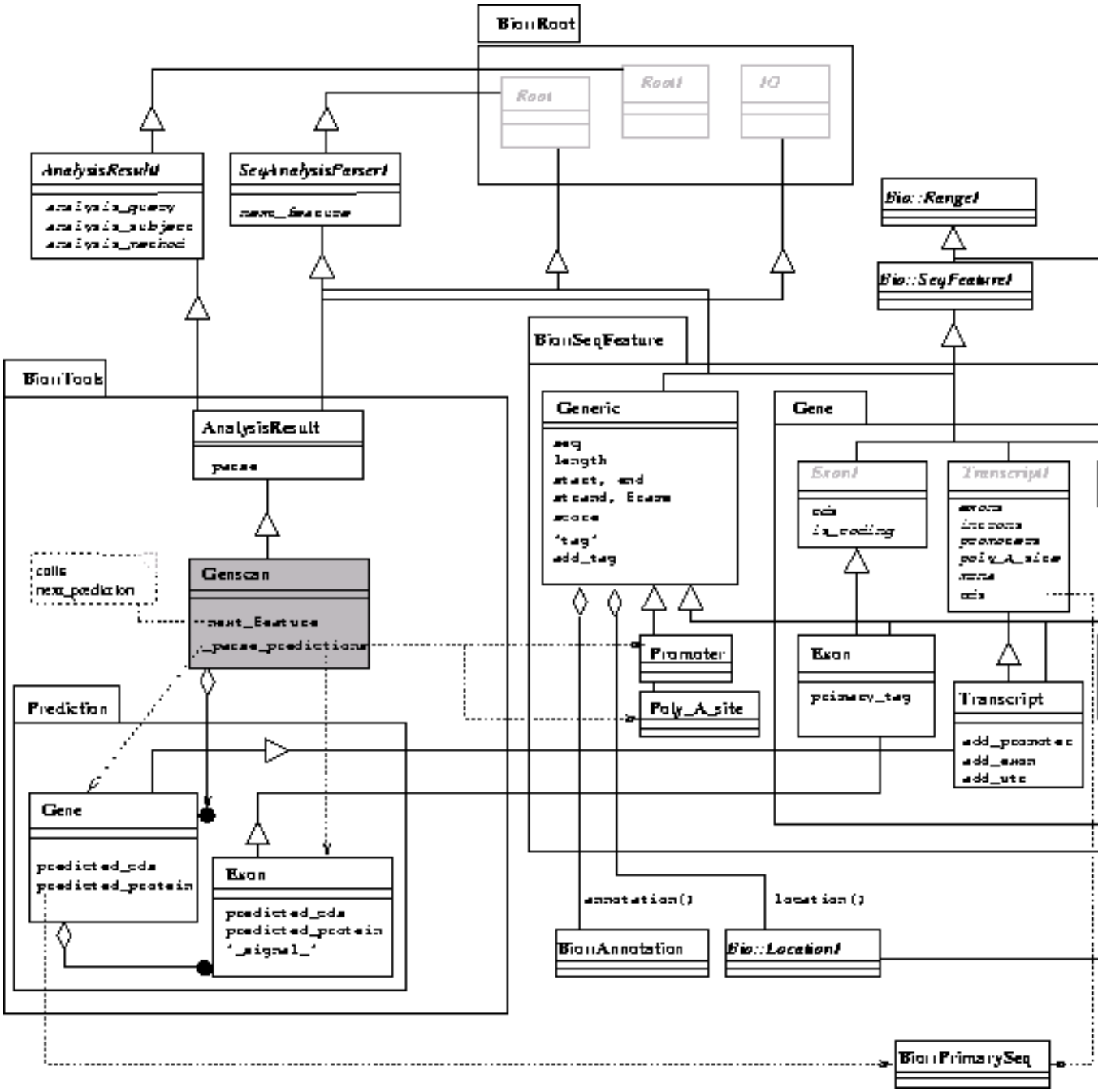
Example 4.9. Genscan parsing, with sub-sequences

(data: Genscan output with CDS [data/genscan-cds.out])

```
# Genscan: example with sub-sequences

use Bio::Tools::Genscan;

my $genscan_file = $ARGV[0];

$genscan = Bio::Tools::Genscan->new(-file => $genscan_file);
while(my $gene = $genscan->next_prediction()) {

    my $prot = $gene->predicted_protein;
    print "protein (", ref($prot), ") :\n", $prot->seq, "\n\n";

    # display genscan predicted cds (if -cds genscan option)
    my $predicted_cds = $gene->predicted_cds;
    print "predicted CDS: \n", $predicted_cds->seq, "\n\n";

    foreach my $exon ($gene->exons()) {
 my $loc = $exon->location;
 print "exon - primary_tag: ",$exon->primary_tag, " coding? ",$exon->is_coding(), " start-end:

    }

    foreach my $intron ($gene->introns()) {
 my $loc = $intron->location;
```

```
 print "intron primary_tag: ",$intron->primary_tag, " start-end: ", $loc->start, "-", $loc->end, "\n"

    }

    foreach my $utr ($gene->utrs()) {
 my $loc = $utr->location;
 print "utr primary_tag: ",$utr->primary_tag, " start-end: ", $loc->start, "-", $loc->end, "\n";

    }
    print "---------------------------------\n";
}
```

Figure 4.4. Genscan Classes Structure



Exercise 4.22. Code reading: Bio::Tools::Genscan module

1. What is the purpose of the `Bio::SeqAnalysisParserI` class?
2. Look at the `_parse_predictions` method in `Bio::Tools::Genscan`.
3. What happens during the creation of a `Bio::Tools::Genscan` instance? which methods are called? at which class hierarchy level?

Chapter 5. Databases

5.1. Database classes

Example 5.1. Database class use

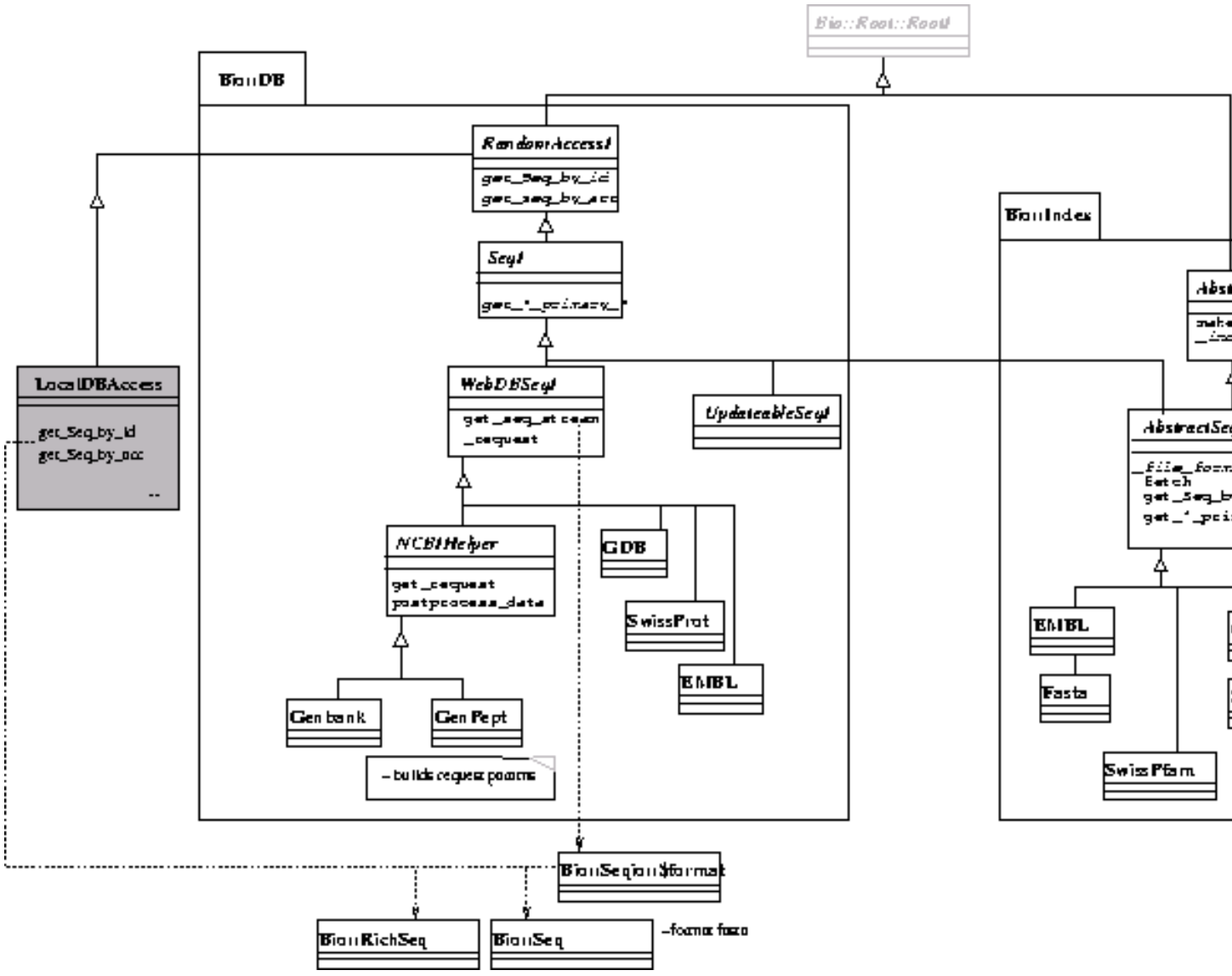
```
use Bio::Index::Swissprot;
use Bio::SeqIO;

my $out = Bio::SeqIO->newFh ( -fh => \*STDOUT, -format => 'fasta');
my $Index_File_Name = shift;
my $inx = Bio::Index::Swissprot->new( -filename => $Index_File_Name);

foreach  my $id (@ARGV) {
  my $seq = $inx->fetch($id);
  print $out $seq;
}
```

Example 5.2. Database Index creation

(data: a SwissProt file to index [data/small_swiss.dat]) :

```
use Bio::Index::Swissprot;

my $Index_File_Name = shift;
my $inx = Bio::Index::Swissprot->new( -filename => $Index_File_Name,
          -write_flag => 1);
$inx->make_index(@ARGV);
```

Figure 5.1. Database Classes structure



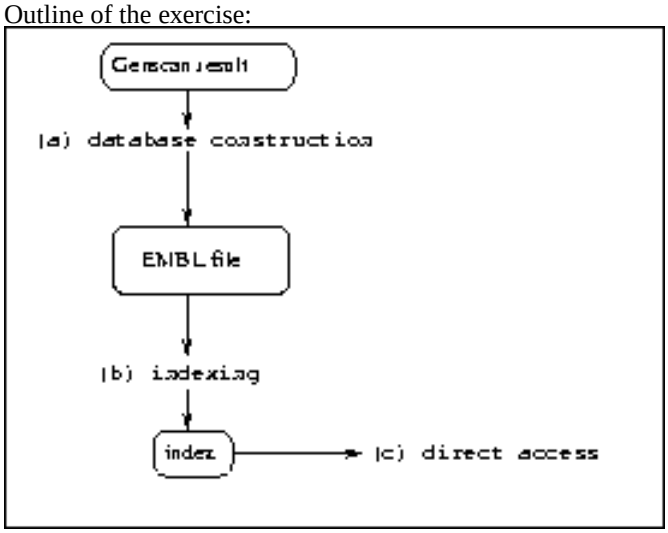
Exercise 5.1. Parse a Genscan report and build a database entry

Parse a Genscan report on a part of Arabidopsis chromosome V (data: Genscan report [data/genscan-arab.out]).

- (a) construct a database in Embl format

-
    - containing an entry for each Genscan predicted CDS
    - put this CDS exons as Features in the current entry
- (b) index this Embl formatted database
- (c) use your indexed database

Outline of the exercise:



Solution A.28

Exercise 5.2. Parse a Genscan report and build a database entry with the genomic sequence

Starting from exercise Exercise 5.1, add the DNA genomic sequence as sequence of the entry (+- delta at the beginning and at the end). Example:

```
SQ   Sequence 2448 BP; 897 A; 487 C; 411 G; 653 T; 0 other;
     ttaagttcca ttgatgtatt tcaaagggtt cagagtttta tcgttttaca aagaaatgat        60
     gagtgttcat gactgtaaaa tccaccttca tcttccactt tcagtttaac ggctccggct       120
```

(data : Arabidopsis sequence - part of the chromosome V [data/arabidop-chr5-1quart.fasta])

- build an additional feature describing all the CDS
- put the translation as a tag for this CDS (see add_tag_value [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SeqFeature/Generic.html#POD14]), which yields, for instance:

```
FT   CDS             join(complement(100..171),complement(284..358),
FT                   complement(457..492),complement(626..673),
FT                   complement(1460..1511),complement(2473..2513),
FT                   complement(2638..2716),complement(2813..2969),
FT                   complement(3046..3178),complement(3263..3390),
FT                   complement(3476..3635))
FT                   /translation="MASTEGLMPITRAFLASYYDKYPFSPLSDDVSRLSSDMASLIKLL
FT                   TVQSPPSQGETSLIDEANRQPPHKIDENMWKNREQMEEILFLLSPSRWPVQLREPSTSE
FT                   DAEFASILRTLKDSFDNAFTAMISFQTKNSERIFSTVMTYMPQDFRGTLIRQQKERSER
FT                   NKQAEVDALVSSGGSIRDTYALLWKQQMERRRQLAQLGSATGVYKTLVKYLVGVPQVLL
FT                   DFIRQINDDDGDNEEYKEIIVQAGRTYEDIGFSVEYINASGEKTLILPYRRYEADQGNF
FT                   STLMAGNYKLVWDNSYSTFFKKTLRYKVDCIAPVVEPDPEPEPLN"
```

- regarding the location of the feature, see Section 2.3.4 in this tutorial, as well as add_sub_Location [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Location/Split.html#POD2].

Solution A.29

5.2. Accessing a local database with golden

Exercise 5.3. Build a small bioperl module (for the golden program)

Idea: we want a bioperl module for local databases access. We could use bioperl indexes, except that this is not efficient for large databases (indexing time, and indexes size). All these databases are already indexed by golden [ftp://ftp.pasteur.fr/pub/GenSoft/unix/db_soft/golden] in a very efficient way. The module could be designed to be used like this:

```
use LocalDBAccess;
use Bio::SeqIO;

my $seq = LocalDBAccess->get_Seq_by_id ( $ARGV[0] );
my $out = Bio::SeqIO->newFh ( -fh     => \*STDOUT,
                              -format => 'fasta' );


print $out $seq;
```

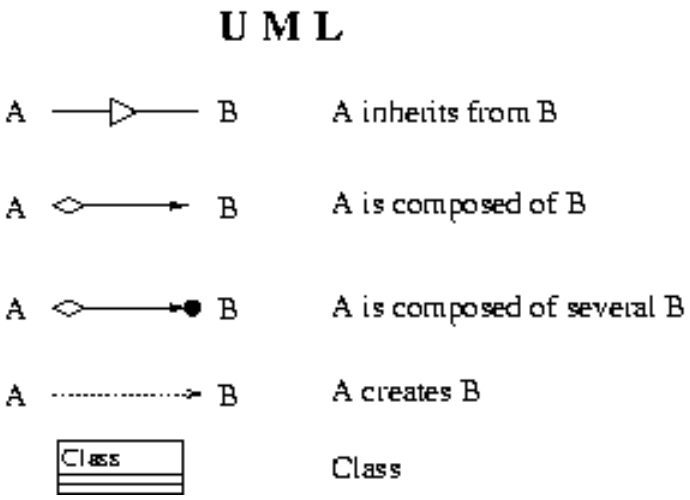
Solution A.30

Chapter 6. Perl Reminders

These reminders may be helpful to use modules [#use], or to a more advanced understanding [#advanced] of them.

6.1. UML

Figure 6.1. UML meanings



6.2. Perl reminders to use bioperl modules

6.2.1. References

- to pass a hash or a function as a parameter to a function:

```
$blastObj = Bio::Tools::Blast->new( -run       => \%runParam,
                                    -parse     => 1,
           -filt_func => \&my_filter);
```

- for an output parameter:

```
%blastParam = (-run        => \%runParam,
               -save_array => \@blast_objs );
```

- to pass a filehandle as a parameter:

```
my $banque = Bio::SeqIO->newFh ( -fh     => \*STDOUT,
          -format => 'embl');
```

- references on an anonymous array:

```
%runParam = ( -method   => 'remote',
              -prog     => 'blastp',
              -database => 'swissprot',
              -seqs     => [ $seq ]);  # Bio::Seq.pm objects.
```

which is "equivalent" to:

```
%runParam = ( -method   => 'remote',
              -prog     => 'blastp',
              -database => 'swissprot',
              -seqs     => \@seqs);  # Bio::Seq.pm objects.
```

with a variable.

- references on anonymous data structures:

```
# (reminder) anonymous hash:
%a = (a => 1, b => 2 );
print "hash ", ref(%a), " ", $a{a}, " ", $a{b}, "\n";

# reference on an anonymous hash:
$ra = {a => 1, b => 2 };
print "ref hash ", ref($ra), " ", $ra->{a}, " ", $ra->{b}, "\n";

# reference on an anonymous array:
$rl = [1, 2];
print "ref array ", ref($rl), " ", $rl->[0], " ", $rl->[1], "\n";
```

See for instance in exercises: 2.4 et 2.7. `ref()` returns the type of the variable (or its class,see below).

6.2.2. Filehandles and streams

- A data stream is for instance an open file. Streams are available through "filehandles" (file descriptors) - in perl: a symbol without a `$`, such as `INFILE` or `OUTFILE`. Standard input and standard output streams are associated by default to the `STDIN` and `STDOUT` descriptors. The filehandle is associated to the stream when opening the file:

```
open (INFILE, $infile) || die "cannot open $infile:$!";
open (OUTFILE, "> $outfile") || die "cannot open $outfile:$!";
$line = <INFILE> ;

# read a sequence objectprint OUTFILE $sequence;
# write a sequence object
```

- In bioperl, there are classes that build filehandles:

```
$in = Bio::SeqIO->newFh ( -file   => 'f') ;  # make a tied filehandle
$sequence = <$in>;   # read a sequence object
$out = Bio::SeqIO->newFh ( -file   => '> f');
print $fh $sequence;  # write a sequence object
```

- Filhandles such as `INFILE` or `STDOUT` belong to a specific namespace and can not be used in an assignation or as a parameter. For this, you have to pass a reference to the typeglob:

```
my $banque = Bio::SeqIO->newFh ( -fh     => \*STDOUT,
                      -format => 'embl');
```

typeglobs are in a general namespace, on top of the other namespaces for scalars, arrays, hashes and functions. You can use it to create aliases (see the "Advanced Perl Programming" book).

6.2.3. Exceptions

- When using a bioperl module, the following can happen:

```
-------------------- EXCEPTION --------------------
MSG: Attempting to set the sequence to [==] which does not look healthy
STACK Bio::PrimarySeq::seq /local/lib/perl5/site_perl/5.6.0/Bio/PrimarySeq.pm:243
STACK Bio::PrimarySeq::new /local/lib/perl5/site_perl/5.6.0/Bio/PrimarySeq.pm:218
STACK Bio::Seq::new /local/lib/perl5/site_perl/5.6.0/Bio/Seq.pm:132
STACK Bio::SeqIO::fasta::next_primary_seq /local/lib/perl5/site_perl/5.6.0/Bio/SeqIO/fasta.

STACK Bio::SeqIO::fasta::next_seq /local/lib/perl5/site_perl/5.6.0/Bio/SeqIO/fasta.pm:85

STACK toplevel solution/blast_features.pl:22
---------------------------------------------
```

bioperl indeed use perl exception mechanisms (`die` et `eval`) to handle use errors. An exception is raised by the perl module an can be caught:

- `throw` : in bioperl, this is the method for raising exceptions. It is provided by the Bio::Root::RootI [http://doc.bioperl.org/releases/bioperl-1.0/Bio//Root/RootI.html] class:

```
sub throw{
  my ($self,$string) = @_;
  my $std = $self->stack_trace_dump();
  my $out = "-------------------- EXCEPTION --------------------\n"."MSG: ".$string."\n".$std."---

  die $out;}
```

- You can catch exceptions with `eval` (don't forget the `;` at the end of the `eval` block:

```
eval {
  # this instruction may throw an exception
  # if parameters are incorrect
  $report = $factory->bl2seq($querySeq, $subjectSeq );
};
if( $@ ) {
  print "Caught exception";
}
else {
  print "no exception";
}
```

- In bioperl, exceptions are also used to implement interfaces classes; the following example in Bio::PrimarySeqI [http://doc.bioperl.org/releases/bioperl-1.0/Bio/PrimarySeqI.html]:

```
sub seq {
  my ($self) = @_;
  if( $self->can('throw') ) {
    $self->throw("Bio::PrimarySeqI definition of seq - implementing class did not provide this met

  }
  else {
    confess("Bio::PrimarySeqI definition of seq - implementing class did not provide this method")

  }
}
```

is a mean to oblige the subclass to implement a `seq` method. It is thus a way to encode abstract methods.

6.2.4. Getopt::Std

- To handle command line options: `man Getopt::Std`.
- Example :

```
use Getopt::Std;
my %opts = ();getopts('i:o:I:O:', \%opts);
my $infile  = $opts{'I'} || 'infile';
my $outfile = $opts{'O'} || 'outfile';
my $inform  = $opts{'i'} || 'swiss';
my $outform = $opts{'o'} || 'fasta';
```

- Another example :

```
use Getopt::Std;getopts('f:plgh');
if ($opt_f) {    $format = $opt_f;}
else {    $format = "Genbank";}
```

6.2.5. Classes

- Inheritance, example in Bio::Seq [http://doc.bioperl.org/releases/bioperl-1.0/Bio/Seq.html]:

```
@ISA = qw(Bio::Root::RootI Bio::SeqI);
```

- In bioperl, you use classes the following ways:
  - By instantiating a class, most often with a `new` method (this is a coding convention - see also `newFh` [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SeqIO.html#newFh], `from_searchResult` [http://doc.bioperl.org/releases/bioperl-1.0/Bio/SeqFeature/SimilarityPair.html#from_searchResult], ...) :

```
$seq_stats  =  Bio::Tools::SeqStats->new ($seqobj);
```

  - As a library component.

- Test whether a class or an object *is-a*:

```
if (Bio::Tools::Blast->isa(Bio::Root::RootI)) {
  ...}
if ($seq->isa(Bio::Seq::RichSeq)) {
  ...}
```

- find the class of an object:

```
print "class of exon: ", ref($exon), "\n";
```

6.2.6. BEGIN block

Statements to be executed when the module is loaded (e.g when issuing a `use Module;` statement) ; see for instance:

`Bio::LocationI` to define the default coordinate policy:

```
BEGIN { $coord_policy = Bio::Location::WidestCoordPolicy->new();}
```

Remark: this is necessary in object-oriented style, since everything is encapsulated in methods.

6.3. Perl reminders for a further advanced understanding of bioperl modules

6.3.1. Modules

- Exporter [http://www.perldoc.com/perl5.6/lib/Exporter.html], handles module variables scope (even though there are no private variables in perl).
  - `@EXPORT= qw(...);` : symbols exported by default;
  - `@EXPORT_OK = qw(...);` : symbols imported on demand;

- `%EXPORT_TAGS = tag => [...];` tags for symbols sets; example :

  ```
  use Bio::Tools::Blast qw(:obj);
  ```

  imports symbols tagged as `obj` - here:`$Blast`, a static `Bio::Tools::Blast` object for a restrcited use of the module methods.

6.3.2. Compiler instructions

- `use strict`: no symbolic references, no access to non local variables (local variable are declared by a `my` statement), no barewords.
- `use vars qw(@ISA %valid_type);` : a "pragma"to pre-declare global variables.

6.3.3. Tie

: Associates a class to a variable. whenever a variable is "tied",read and write statements (access, assignation, ...) trigger a call topredefined subroutines (`FETCH`, `STORE`, `PRINT`, `READLINE`...). You can find an example in (Bio::SeqIO):

```
sub fh {  my $self = shift;  my $class = ref($self) || $self;
  my $s = Symbol::gensym;
  tie $$s,$class,$self;  return $s;
}
```

which returns a filehandle tied to the Bio::SeqIO::Fhclass.As explained in Tying-FileHandles, Bio::SeqIO class has thus to redefine these methods:

```
sub READLINE {
  my $self = shift;
  return $self->{'seqio'}->next_seq() unless wantarray;
  my (@list, $obj);
  push @list, $obj while $obj = $self->{'seqio'}->next_seq();
  return @list;
}

sub PRINT {
  my $self = shift;  $self->{'seqio'}->write_seq(@_);
}
```

These methods enable the programmer to read and print through the variable:

```
$fh = $obj->fh;
# make a tied filehandle$sequence = <$fh>;
# read a sequence object (calls READLINE)
print $fh $sequence;
# write a sequence object (calls PRINT)
```

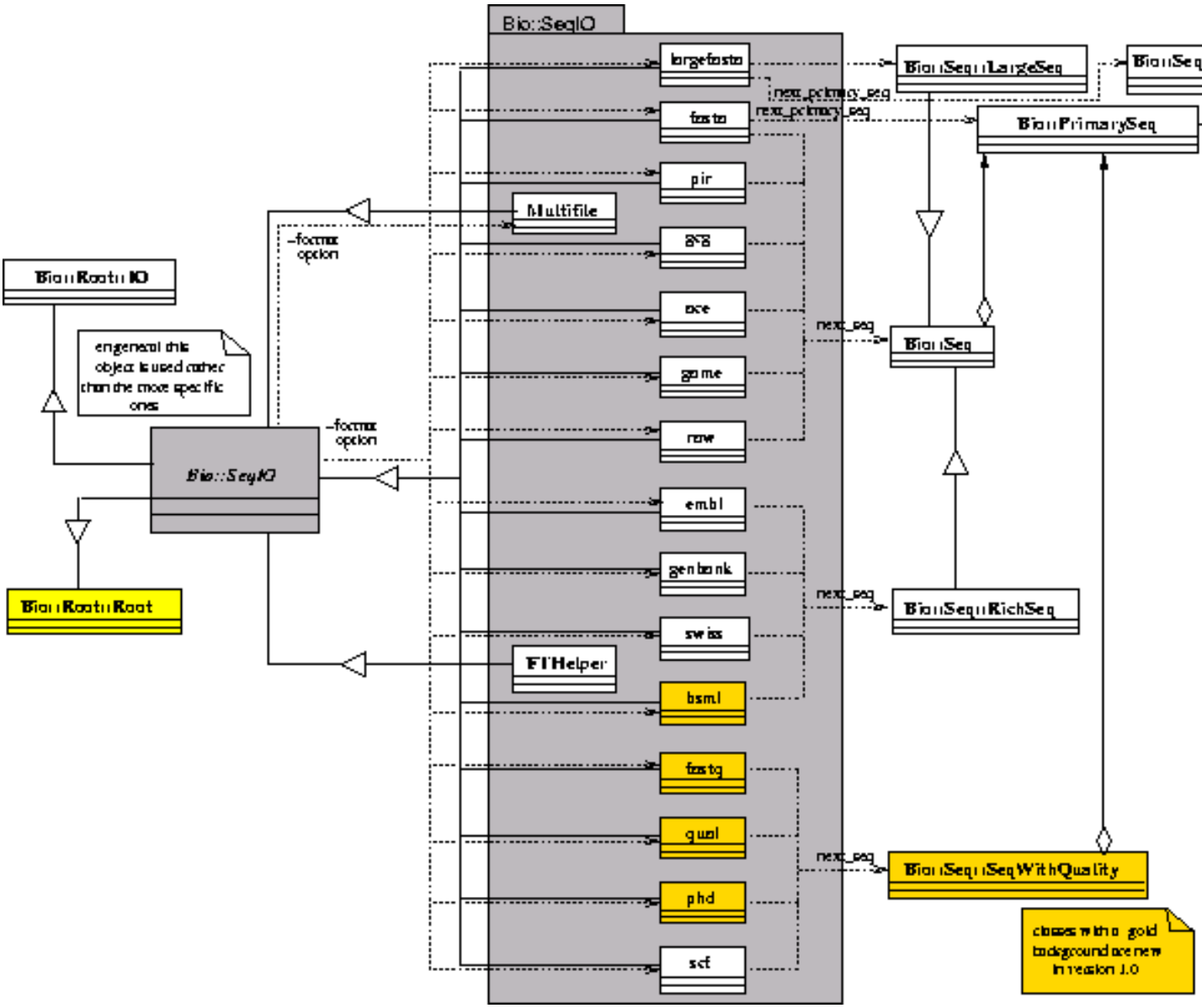
Appendix A. Solutions

A.1. Sequences

Solution A.1. Bio::SeqIO structure

Exercise 2.1

Figure A.1. Bio::SeqIO structure



Solution A.2. An universal converter

Exercise 2.2

UnivConvert via a file.

```
#! /local/bin/perl -w

# exercise 1.1 : UnivConvert via file

use strict;

use Bio::SeqIO;

my $inform = shift @ARGV;
my $outform = shift @ARGV;

my $infile = shift @ARGV;
my $outfile = shift @ARGV;
if ( ! defined $outfile ) {
    $outfile = $infile;
    $outfile =~ s/\..*$// if $outfile =~ /\./;
    $outfile .= "." . $outform;
}

my $in  = Bio::SeqIO->newFh ( -file   => $infile,
          -format => $inform );

my $out = Bio::SeqIO->newFh ( -file   => ">$outfile",
         -format => $outform );

print $out $_ while <$in>;
```

You can use it like this:

```
perl UnivConvert2.pl swiss gcg seqs.sp
```

UnivConvert via stream.

```
#! /local/bin/perl -w

# exercise 1.1 : UnivConvert via flux

use strict;

use Bio::SeqIO;

my $inform = shift @ARGV || 'swiss';
my $outform = shift @ARGV || 'fasta';

my $in  = Bio::SeqIO->newFh ( -fh     => \*STDIN,
          -format => $inform );

my $out = Bio::SeqIO->newFh ( -fh     => \*STDOUT,
```

```
         -format => $outform );

print $out $_ while <$in>;
```

You can use it like this:

```
cat seqs.sp | perl ../solution/UnivConvertStdmin.pl swiss gcg
```

UnivConvert with options.

```
#! /local/bin/perl -w

# exercice 1.1 : UnivConvert avec options

use strict;
use Getopt::Std;

use Bio::SeqIO;

my %opts = ();
getopts('i:o:I:O:', \%opts);

my $infile  = $opts{'I'} || 'infile';
my $outfile = $opts{'O'} || 'outfile';
my $inform  = $opts{'i'} || 'swiss';
my $outform = $opts{'o'} || 'fasta';


my $in  = Bio::SeqIO->newFh ( -file   => $infile,
          -format => $inform );

my $out = Bio::SeqIO->newFh ( -file   => ">$outfile",
         -format => $outform );


print $out $_ while <$in>;
```

You can use it like this:

```
perl ../solution/UnivConvert.pl -I seqs.sp -O seqs.fasta
```

UnivConvert with options and stream.

```
#! /local/bin/perl -w
```

```
# exercise 1.1 : UnivConvert with option and stream

use strict;
use Getopt::Std;

use Bio::SeqIO;

my %opts = ();
getopts('hi:o:', \%opts);

usage() if (exists $opts{'h'});

my $inform  = lc($opts{'i'}) || 'swiss';
my $outform = lc($opts{'o'}) || 'fasta';

(format_ok($inform) && format_ok($outform)) || exit 1;

my $in  = Bio::SeqIO->newFh ( -fh     => \*STDIN,
          -format => $inform );

my $out = Bio::SeqIO->newFh ( -fh     => \*STDOUT,
         -format => $outform );


print $out $_ while <$in>;

sub format_ok {

    my $form = shift @_;

    my %formats = qw(fasta 1 swiss 1 embl 1 genbank 1
       scf 1 pir 1 gcg 1 raw 1 ace 1 bsml 1 fastq 1
       phd 1 qual 1);

    if (! exists $formats{$form}) {
 print STDERR $form, " -- unknown format\n";
 usage ();
 return 0; # false
    }

    return 1; # true
}


sub usage {

    my $prog ='basename $0'; chomp($prog);

    print "\n";
    print "$prog convert files from one format into another format\n";
    print "\n";
```

```
    print "$prog [-i informat] [-o outformat] < infile\n";
    print "\n";
    print "-i informat    -- infile format (default 'swiss')\n";
    print "-o outformat   -- outfile format (default 'fasta')\n";
    print "\n";
    print "-h             -- print this message\n";
    print "\n";
    print "Available formats: fasta, swiss, EMBL, GenBank, SCF, Pir, GCG, raw et ACE\n";

    print "\n";

    exit 1;

}
```

You can use it like this:

```
cat seqs.sp | perl ../solution/UnivConvertStd.pl -o gcg > seqs.gcg
```

Solution A.3. An universal converter using the new method

Exercise 2.3

```
use Getopt::Std;

use Bio::SeqIO;

my %opts = ();
getopts('i:o:I:O:', \%opts);

my $infile  = $opts{'I'} || 'infile';
my $outfile = $opts{'O'} || 'outfile';
my $inform  = $opts{'i'} || 'swiss';
my $outform = $opts{'o'} || 'fasta';

my $in  = Bio::SeqIO->new ( -file   => $infile,
        -format => $inform );

my $out = Bio::SeqIO->new ( -file   => ">$outfile",
       -format => $outform );

while ( my $seq = $in->next_seq() ) {
       $out->write_seq($seq);
}
```

Solution A.4. Display a sequence in fasta format

Exercise 2.4

```
use Bio::DB::SwissProt;

$database = new Bio::DB::SwissProt;

$seq = $database->get_Seq_by_id('MALK_ECOLI');

my $out = Bio::SeqIO->newFh ( -fh => \*STDOUT,
         -format => 'fasta');
print $out $seq;
```

Solution A.5. More on annotations

Exercise 2.5

```
# get the protein function -- exo
my $function = "";
foreach my $comment ( $annotation->get_Annotations('comment') ) {
    my @lines = split (/\.\n/, $comment->text());
    foreach my $line (@lines) {
 if ( $line =~ s/^-!- FUNCTION:// ) {
     $line =~ s/\n/ /g;
     $function = $line;
     last;
 }
    }
    last if $function ne "";
}
print "\nFunction:", $function, "\n";
```

Solution A.6. Transmembran helices

Exercise 2.6

```
# almost for GenBank/EMBL entries
my @features = $seq->top_SeqFeatures();

foreach my $feat (@features) {
    if ($feat->primary_tag() eq 'TRANSMEM') {
 print "TM Segment: from ", $feat->start(),
                    " to ", $feat->end();
 print "(", $feat->seq()->seq, ")\n";
    }
```

```
}
```

A.2. Alignments

Solution A.7. Create an alignment without gaps

Exercise 3.1

```
#! /local/bin/perl -w

###  Create a new alignment without gaps at the begining and the end
###  of an alignment

use strict;
use Bio::AlignIO;

# get alignment

my $in;
if ( $#ARGV > -1 ) {
    $in = new Bio::AlignIO ( -file =>, $ARGV[0], -format => 'clustalw' );
}
else {
    $in = new Bio::AlignIO ( -fh => \*STDIN, -format => 'clustalw' );
}
my $out  = newFh Bio::AlignIO ( -fh => \*STDOUT, -format => 'clustalw' );

my $aln = $in->next_aln();

# find max column index of starting gaps and
# min column index of ending gaps

my $startgaps = 0;
my $endgaps = 0;
my $gap_char = $aln->gap_char();
foreach my $seq ($aln->each_seq) {
    my $str = $seq->seq();
    $str =~/^(\Q$gap_char\E*)/;
    my $len = length($1);
    $startgaps = ($startgaps > $len) ? $startgaps : $len;
    $str =~/(-*)$/;
    $len = length($1);
    $endgaps = ($endgaps > $len) ? $endgaps : $len;
}

# cut the starting and ending block containing gaps

print $out $aln->slice($startgaps+1, $aln->length()-$endgaps);
```

A.3. Analysis

Solution A.8. Running Blast on a Swissprot entry

Exercise 4.1

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

# a) solution if you have a local program to fetch
# database entries
# (ftp://ftp.pasteur.fr/pub/GenSoft/unix/db_soft/golden/)

#my $Seq_in = Bio::SeqIO->new (-file => 'golden sp:TAUD_ECOLI |',
#         -format => 'swiss');
#my $query = $Seq_in->next_seq();;

# b) else:

use Bio::DB::SwissProt;
my $database = new Bio::DB::SwissProt;
my $query = $database->get_Seq_by_id('TAUD_ECOLI');

my $factory = Bio::Tools::Run::StandAloneBlast->new(
    'program'  => 'blastp',
    'database' => 'swissprot',
     _READMETHOD => "Blast"
    );
my $blast_report = $factory->blastall($query);
my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    print "\thit name: ", $hit->name(),
    " significance: ", $hit->significance(), "\n";
}
```

Solution A.9. Running Blast: Setting parameters

Exercise 4.2

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
         -format => 'fasta');
```

```
my $query = $Seq_in->next_seq();;

my $factory = Bio::Tools::Run::StandAloneBlast->new(
    'program'  => 'blastp',
    'database' => 'swissprot',
     _READMETHOD => "Blast"
    );
$factory->e(1.0);
my $blast_report = $factory->blastall($query);
my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    print "\thit name: ", $hit->name(),
    " significance: ", $hit->significance(), "\n";
}
```

Solution A.10. Running Blast: Saving output

Exercise 4.3

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
         -format => 'fasta');
my $query = $Seq_in->next_seq();;

my $factory = Bio::Tools::Run::StandAloneBlast->new(
    'program'  => 'blastp',
    'database' => 'swissprot',
     _READMETHOD => "Blast"
    );
$factory->outfile('blast.out');
my $blast_report = $factory->blastall($query);
my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    print "\thit name: ", $hit->name(),
    " significance: ", $hit->significance(), "\n";
}
```

Solution A.11. Running a remote Blast

Exercise 4.4

```
use Bio::SeqIO;
use Bio::Tools::Run::RemoteBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
```

```
         -format => 'fasta');
my $query = $Seq_in->next_seq();

my $factory = Bio::Tools::Run::RemoteBlast->new(
    '-prog'  => 'blastp',
    '-data' => 'swissprot',
     _READMETHOD => "Blast"
    );
my $blast_report = $factory->submit_blast($query);
my $max_number = 100;
my $trial = 0;


while ( my @rids = $factory->each_rid ) {

    print STDERR "\nSorry, maximum number of retries $max_number exceeded\n" if $trial >= $max

    last if $trial >= $max_number;
    $trial++;

    print STDERR "waiting... ".(5*$trial)." units of time\n" ;

    # RID = Remote Blast ID (e.g: 1017772174-16400-6638)
    foreach my $rid ( @rids ) {
 my $rc = $factory->retrieve_blast($rid);
 if( !ref($rc) ) {
     if( $rc < 0 ) {
  # retrieve_blast returns -1 on error
  $factory->remove_rid($rid);
     }
     # retrieve_blast returns 0 on 'job not finished'
            sleep 5*$trial;
 } else {

     #---- Blast done ----
     $factory->remove_rid($rid);
     my $result = $rc->next_result;
     print "database: ", $result->database_name(), "\n";
     while( my $hit = $result->next_hit ) {
  print "hit name is: ", $hit->name, "\n";
  while( my $hsp = $hit->next_hsp ) {
      print "score is: ", $hsp->score, "\n";
  }
     }
 }
    }
}
```

Solution A.12. Display Blast hits

Exercise 4.5

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
                              -format => 'fasta');
my $query = $Seq_in->next_seq();;

my $factory = Bio::Tools::Run::StandAloneBlast->new(
       'program'  => 'blastp',
       'database' => 'swissprot',
                           _READMETHOD => "Blast"
                                       );
my $blast_report = $factory->blastall($query);
my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    print "\thit accession: ", $hit->accession(), "\n";
    while( my $hsp = $hit->next_hsp()) {
 print "length: ", $hsp->length(), "\n";
    }
}
```

Solution A.13. Class of a Blast report

Exercise 4.6

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
                              -format => 'fasta');
my $query = $Seq_in->next_seq();;

my $factory = Bio::Tools::Run::StandAloneBlast->new(
       'program'  => 'blastp',
       'database' => 'swissprot',
                           _READMETHOD => "Blast"
                                       );
my $blast_report = $factory->blastall($query);
print "Class of blast_report is: ", ref($blast_report), "\n";
```

Solution A.14. Parse Blast results on standard input

Exercise 4.8

```
use Bio::SearchIO;
my $blast_report = new Bio::SearchIO (’-format’ => ’blast’,
          ’-fh’   => \*STDIN
          );
my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    print "\thit name: ", $hit->name(), "\n";
    while( my $hsp = $hit->next_hsp()) {
 print "E: ", $hsp->evalue(), "frac_identical: ",
 $hsp->frac_identical(), "\n";
    }
}
```

One way to use this code is by feeding a **blastall** output directly to the script through a Unix pipe:

```
blastall -p blastp -d swissprot -i data/1prot.fasta | perl solution/blast_parse_stdin.pl
```

Solution A.15. Filtering hits by length

Exercise 4.9

```
use Bio::SearchIO;
my $max_length = ($ARGV[1])? $ARGV[1] : 200;
my $blast_report = new Bio::SearchIO (’-format’ => ’blast’,
          ’-file’   => $ARGV[0]);
my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    if ($hit->length() >= $max_length) {
 print "\thit name: ", $hit->name(), " hit length: ", $hit->length(), "\n";
 while( my $hsp = $hit->next_hsp()) {
     print "E: ", $hsp->evalue(), "frac_identical: ",
     $hsp->frac_identical(), "\n";
 }
    }
}
```

Solution A.16. Filtering hits by position

Exercise 4.10

```
use Bio::SearchIO;

my $min_start = ($ARGV[1])? $ARGV[1] : 1;
my $max_end = ($ARGV[2])? $ARGV[2] : -1;

my $blast_report = new Bio::SearchIO ('-format' => 'blast',
          '-file'   => $ARGV[0]);
my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    print "\thit name: ", $hit->name(), "\n";
    while( my $hsp = $hit->next_hsp()) {
 if ($hsp->start() >= $min_start &&
     ($max_end == -1 || ($hsp->end() <= $max_end))
     ) {
     print "\t\tstart: ", $hsp->start(), " end: ", $hsp->end(), "\n";
     print "\t\tE: ", $hsp->evalue(), "frac_identical: ",
     $hsp->frac_identical(), "\n";
 }
    }
}
```

Solution A.17. Display the best hit by databank

Exercise 4.11

```
use Bio::SearchIO;

sub dbname {
    my ($name) = @_;
    $db= $name;
    $db =~ /(\w+)|.*/;
    $db = $1;
    return $db;
}

my $blast_report = new Bio::SearchIO ('-format' => 'blast',
          '-file'   => $ARGV[0]);
my $result = $blast_report->next_result;

while( my $hit = $result->next_hit()) {
    my $dbname = dbname($hit->name());
    if ($done{$dbname}) {
        next;
    } else {
        $done{$dbname} = 1;
    }
    print "\thit name: ", $hit->name(), "\n";
```

```
    while( my $hsp = $hit->next_hsp()) {
     print "\t\tE: ", $hsp->evalue(), "frac_identical: ",
     $hsp->frac_identical(), "\n";
    }
}
```

Solution A.18. Multiple queries

Exercise 4.12

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my @query_seqs = ();
my $Seq1_in = Bio::SeqIO->new (-file => $ARGV[0],
         -format => 'fasta');
push (@query_seqs, $Seq1_in->next_seq());

my $Seq2_in = Bio::SeqIO->new (-file => $ARGV[1],
         -format => 'fasta');
push (@query_seqs, $Seq2_in->next_seq());

my $factory = Bio::Tools::Run::StandAloneBlast->new(
          'program'  => 'blastp',
          'database' => 'swissprot',
          _READMETHOD => "Blast"
          );
my $blast_report = $factory->blastall(\@query_seqs);
while (my $result = $blast_report->next_result()) {
    while( my $hit = $result->next_hit()) {
 print "\thit name: ", $hit->name(),
 " significance: ", $hit->significance(), "\n";
 last;
    }
}
```

Solution A.19. Extracting the subject sequence

Exercise 4.13

```
use Bio::SearchIO;

my $blast_report = new Bio::SearchIO ('-format' => 'blast',
          '-file'   => $ARGV[0]);
my $result = $blast_report->next_result;
my $level = $ARGV[1];
```

```
while( my $hit = $result->next_hit()) {
    while( my $hsp = $hit->next_hsp()) {
 if ($hsp->frac_identical() >= $level) {
     print $hsp->hit_string, "\n";
 }
    }
}
```

Solution A.20. Extracting alignments

Exercise 4.14

```
use Bio::SearchIO;
use Bio::AlignIO;

my $blast_report = new Bio::SearchIO ('-format' => 'blast',
          '-file'   => $ARGV[0]);
my $result = $blast_report->next_result;
my $pattern = $ARGV[1];
my $out = Bio::AlignIO->newFh(-format => 'clustalw' );

while( my $hit = $result->next_hit()) {
    if ($hit->name() =~ /$pattern/i ) {
 while( my $hsp = $hit->next_hsp()) {
     my $aln = $hsp->get_aln();
     print $out $aln;
 }
    }
}
```

Solution A.21. Locate EST in a genome

Exercise 4.15

```
use Bio::SeqIO;
use Bio::SearchIO;
use Bio::AlignIO;
use Bio::SimpleAlign;
use Bio::LocatableSeq;

my $seq_in = Bio::SeqIO->new (-file => $ARGV[0],
         -format => 'fasta');
my $seq = $seq_in->next_seq();
my $contig_seq = Bio::LocatableSeq->new(
     -seq => $seq->seq,
     -id => $seq->display_id,
     -start => 1,
     -end => $seq->length
```

```
     );
my $aln = Bio::SimpleAlign->new();
$aln->add_seq($contig_seq);

my $blast_report = new Bio::SearchIO ('-format' => 'blast',
                                      '-file'   => $ARGV[1]);

my $result = $blast_report->next_result;
while( my $hit = $result->next_hit()) {
    my $i = 0;
    while( my $hsp = $hit->next_hsp()) {
 $i++;
 my $start = $hsp->hit->start();
 my $end = $hsp->hit->end();
 my $name = $result->query_name . "_HSP_$i";
 my $seq = fill_gaps($hsp->query_string, $start, $aln->length);
 my $query_seq = Bio::LocatableSeq->new(
          -seq => $seq,
          -id => $name,
          -start => $start,
          -end => $end
          );
 $aln->add_seq($query_seq);
    }
}

my $out = Bio::AlignIO->newFh(-format => 'clustalw' );
print $out $aln;

sub fill_gaps {
    # add gaps from 0 to start and from start + seq length to end
    my ($seq, $start, $length) = @_;
    my $gapped_seq = "";
    for (my $i = 0; $i < $start - 1; $i++) {
 $gapped_seq .= "-";
    }
    $gapped_seq .= $seq;
    for (my $i = $start + length($seq); $i <= $length; $i++) {
 $gapped_seq .= "-";
    }
    return $gapped_seq;
}
```

Example of use:

```
perl solution/blast_locate.pl data/contig data/blast-est.out
```

Solution A.22. Record Blast hits as sequence features

Exercise 4.16

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::Seq;
use Bio::SeqFeature::SimilarityPair;

my $seqfile = shift @ARGV;
my $in = Bio::SeqIO->new (-file => $seqfile, -format => 'fasta');
my $query = $in->next_seq();
my @dbs = @ARGV;

foreach my $db (@dbs) {
    my $factory = Bio::Tools::Run::StandAloneBlast->new(database => $db,
       program => 'blastp');

    my $report = $factory->blastall($query);

    while(my $sbjct = $report->nextSbjct) {
 print STDERR "name : ", $sbjct->name, "\n";
 while (my $hsp = $sbjct->nextHSP) {
     print STDERR "GFF:\n", $hsp->query()->gff_string(), "\n";
     $query->add_SeqFeature($hsp);
 }
 last;
    }
}

foreach my $feat ($query->all_SeqFeatures()) {
    print STDERR "Feature from : ", $feat->start, " to : ",
    $feat->end, " Primary tag : ", $feat->primary_tag,
    ", produced by ", $feat->source_tag(), "\n";
    $feat->primary_tag("BLAST");
}

$out = Bio::SeqIO->newFh(-fh => \*STDOUT , '-format' => 'swiss');
print $out $query;
```

Solution A.23. Print informations from a BPLite report

Exercise 4.17

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;

my $Seq_in = Bio::SeqIO->new (-file => $ARGV[0],
```

```
         -format => 'fasta');
my $query = $Seq_in->next_seq();;

my $factory = Bio::Tools::Run::StandAloneBlast->new(
          'program'  => 'blastp',
          'database' => 'swissprot'
          );
my $blast_report = $factory->blastall($query);
while (my $subject = $blast_report->nextSbjct()) {
    print $subject->name(), "\n";
    while (my $hsp = $subject->nextHSP()) {
 print join("\t",
     $hsp->P,
     $hsp->percent,
     $hsp->score), "\n";
    }
}
```

Solution A.24. Create a Bio::Tools::BPlite from a file

Exercise 4.18

```
use Bio::Tools::BPlite;

my $report = new Bio::Tools::BPlite(-file => $ARGV[0]);
while (my $subject = $report->nextSbjct()) {
    print $subject->name(), "\n";
    while (my $hsp = $subject->nextHSP()) {
 print join("\t",
     $hsp->P,
     $hsp->score,
     $hsp->length,
     $hsp->match,
     $hsp->positive), "\n";
    }

}
```

Solution A.25. Create a Bio::Tools::BPlite from standard input

Exercise 4.19

```
use Bio::Tools::BPlite;

my $report = new Bio::Tools::BPlite(-fh => \*STDIN);
while (my $subject = $report->nextSbjct()) {
    print $subject->name(), "\n";
    while (my $hsp = $subject->nextHSP()) {
```

```
 print join("\t",
     $hsp->P,
     $hsp->score,
     $hsp->length,
     $hsp->match,
     $hsp->positive), "\n";
    }

}
```

Solution A.26. Parse a PSI-blast report

Exercise 4.20

```
use Bio::Tools::BPpsilite;

my $blast_report = new Bio::Tools::BPpsilite ('-file'   => $ARGV[0]);

for my $iteration (1 .. $blast_report->number_of_iterations()) {
    print "\n-----------------------------\niteration: $iteration\n";
    my $result = $blast_report->round($iteration);
    my $hitarray_ref = $result->newhits;
    foreach my $hit (@{ $hitarray_ref }) {
 print "NEW hit name: $hit\n";
    }
    my $oldhitarray_ref = $result->oldhits;
    foreach my $hit (@previous_hitarray) {
 if (grep /\Q$hit\E/, @{ $oldhitarray_ref }) {
     next;
 }
 print "DISAPEARED hit name: $hit\n";
    }
    push (@previous_hitarray, @{ $oldhitarray_ref });
    push (@previous_hitarray, @{ $hitarray_ref });

}
```

Solution A.27. Build a Bio::SimpleAlign object

Exercise 4.21

```
use Bio::SeqIO;
use Bio::Tools::Run::StandAloneBlast;
use Bio::AlignIO;
use Bio::SimpleAlign;
use Bio::LocatableSeq;
```

```
my $query1_in  = Bio::SeqIO->newFh ( -file   => $ARGV[0],
        -format => 'fasta' );
my $query1 = <$query1_in>;

my $query2_in  = Bio::SeqIO->newFh ( -file   => $ARGV[1],
        -format => 'fasta' );
my $query2 = <$query2_in>;

my $out = Bio::AlignIO->newFh(-format => 'clustalw' );

$factory = Bio::Tools::Run::StandAloneBlast->new(
       'program'  => 'blastp'
       );
$report = $factory->bl2seq($query1, $query2);

while(my $hsp = $report->next_feature) {
    my $aln = Bio::SimpleAlign->new();
    my $querySeq = Bio::LocatableSeq->new(
           -seq => $hsp->qs,
           -id  => $query1->display_id,
           -start => 1,
           -end => $query1->length
        );
    my $sbjctSeq = Bio::LocatableSeq->new(
       -seq => $hsp->ss,
       -id => $report->sbjctName,
       -start => 1,
       -end => $query2->length
       );
    $aln->add_seq($querySeq);
    $aln->add_seq($sbjctSeq);
    print $out $aln;
}
```

A.4. Databases

Solution A.28.

Exercise 5.1

- (a) database construction:

```
#! /local/bin/perl -w
```

```
# (a): Genscan, build a database containing one entry per CDS

use strict;

use Bio::Tools::Genscan;
use Bio::SeqIO;

my $genscan_file = shift @ARGV;

# genscan report
my $genscan = Bio::Tools::Genscan->new(-file => $genscan_file);

# database to construct
my $banque = Bio::SeqIO->newFh ( -fh     => \*STDOUT,
     -format => 'embl');

# date of construction
my $date = gmtime;

while(my $gene = $genscan->next_prediction()) {

    # create entry object as database sequence
    my $entry = Bio::Seq::RichSeq->new();
    $entry->add_date($date);
    $entry->molecule('mRNA');
    $entry->division('PLN');

    # extract id of genscan prediction
    my $cds = $gene->predicted_cds;
    $cds->id() =~ /\|(.+)\|/;
    my $id = $1 . " "; # bug
    $cds->id($id);
    # add cds as sequence to the entry
    $entry->primary_seq($cds);

    # annotation
    # add all detected exons as SeqFeature to the entry
    foreach my $exon ($gene->exons()) {
 $entry->add_SeqFeature($exon);
    }

    # add all detected UTR's as SeqFeature to the entry
    foreach my $utr ($gene->utrs()) {
 $entry->add_SeqFeature($utr);
    }

    # print the entry to the database flatfile in embl format
    print $banque $entry;
}
```

You can use this script like this:

```
perl solution/build_db.pl data/genscan-arab.out > mydb.embl
```

- (b) make index:

```
#! /usr/bin/perl -w

# (b): Genscan, create the index

use strict;
use Bio::Index::EMBL;

my $Index_File_Name = shift;

my $inx = Bio::Index::EMBL->new( -filename => $Index_File_Name,
     -write_flag => 1);
$inx->make_index(@ARGV);
```

You can use this script like this:

```
perl solution/make_index.pl mydb.idx mydb.embl
```

- (c) use index:

```
#! /usr/bin/perl -w

# (c): Genscan, retrieve some files

use strict;
use Bio::Index::EMBL;
use Bio::SeqIO;

my $out = Bio::SeqIO->newFh ( -fh => \*STDOUT,
         -format => 'fasta');

my $Index_File_Name = shift;
my $inx = Bio::Index::EMBL->new($Index_File_Name);
```

```
foreach  my $id (@ARGV) {
    my $seq = $inx->fetch($id);
    if (defined $seq) { print $out $seq; }
    else { print STDERR "no entry with id ", $id, " in this banque\n."; }

}
```

You can use this script like this:

```
perl solution/use_index.pl mydb.idx GENSCAN_predicted_CDS_3
```

since entries names follows the pattern: GENSCAN_predicted_CDS_xxx.

Solution A.29. Parse a Genscan report and build a database entry with the genomic sequence

Exercise 5.2

```
#! /local/bin/perl -w

# Genscan, build a database
#                - 1 entry per gene with genomic DNA
#                   + 1 feature for the whole CDS
#                   + 1 feature per exon

use strict;

use Bio::Tools::Genscan;
use Bio::SeqIO;

my $genscan_file = shift @ARGV;

my $seqin = Bio::SeqIO->new ( -fh => \*STDIN,
         -format => 'fasta');

# sequence submitted to genscan
my $seq = $seqin->next_seq();

# genscan report
my $genscan = Bio::Tools::Genscan->new(-file => $genscan_file);

# database to construct
my $banque = Bio::SeqIO->newFh ( -fh     => \*STDOUT,
     -format => 'embl');

# +- delta of the subsequence for each entry in database
```

```
my $delta = 100;
# date of construction
my $date = gmtime;

# print $banque $seq;

while(my $gene = $genscan->next_prediction()) {

    # create entry object as database sequence
    my $entry = Bio::Seq::RichSeq->new();
    $entry->add_date($date);
    $entry->molecule('DNA');
    $entry->division('PLN');

    # extract id of genscan prediction
    my $cds = $gene->predicted_cds;
    my @ids = split(/\|/, $cds->id());

    # annotation
    my $start = 0;
    my $end   = 0;
    my $loc;
    my $first = 1;
    my $newloc = new Bio::Location::Split();
    my $strand = 1;
    my $nocds = 0;
    foreach my $exon ($gene->exons()) {
 $loc = $exon->location();

 if ($first) { $first=0; $start=$loc->start-$delta; }
 if ($start < 0) { $start = 0; }

 $loc->start($loc->start-$start);
 $loc->end ($loc->end-$start);

 $exon->add_tag_value('source', 'genscan');
 $exon->primary_tag =~ m/(.*)Exon/;
 $exon->primary_tag('Exon');
 $exon->add_tag_value(lc($1), 1);

 # add all detected exons as SeqFeature to the entry
 $entry->add_SeqFeature($exon);

 # construct location for CDS
 if ($strand == -1 && $loc->strand == 1) {
     $newloc->warn("exons are coded on different strands");
     $nocds = 1;
 }
 if ($loc->strand == -1) {
     $loc->strand(1);
     $newloc->strand(-1);
```

```
 }
 $newloc->add_sub_Location($loc);

    }
    $end = $loc->end+$delta;
    if ($end > $seq->length()) { $end = $seq->length(); }

    if (! $nocds) {
 # contruct a SeqFeature CDS for the entry ( join all exons)
 my $newcds = Bio::SeqFeature::Generic->new ( -primary => 'CDS',
           -location    => $newloc);

 # add the translation to the CDS Feature
 $newcds->add_tag_value ('translation', $gene->predicted_protein()->seq());

 # add the CDS as Feature to the entry
 $entry->add_SeqFeature($newcds);
    }

    # construct the sequence for the entry
    # subsequence of the sequence submitted to genscan +- delta at each side
    my $seqstr;
    if ( $start > $end ) { $seqstr = $seq->subseq($end, $start); }
    else { $seqstr = $seq->subseq($start, $end); }

    my $newseq = Bio::PrimarySeq->new ( -seq     => $seqstr,
     -id      => $ids[1] . " ",
     -moltype => 'dna');
    $entry->primary_seq($newseq);

    # print the entry to the database flatfile in embl format
    print $banque $entry;
}
```

Solution A.30. Build a small bioperl module (for the golden program)

Exercise 5.3

```
# Build a small bioperl module

package LocalDBAccess;
```

```
use strict;
use vars qw(@ISA  %AVAIL_LOCALDB);

use Bio::DB::RandomAccessI;
use Bio::SeqIO;
#use Bio::Seq;

BEGIN {

    %AVAIL_LOCALDB = ( sprot   => 'swiss',
         sptrnrdb    => 'swiss',
         gb          => 'genbank',
         embl        => 'embl' );

}


@ISA = qw(Bio::DB::RandomAccessI);

sub get_Seq_by_id{
    my ($self, @args) = @_;

    return $self->get_Seq( @args, '-i');
}

sub get_Seq_by_acc{
    my ($self, @args) = @_;

    return $self->get_Seq(@args, '-a');
}

sub get_Seq {

    my ($self, $entry, $access) = @_;
    my ($db, $id) = split(':', $entry);

    if ($access !~ /^-[ia]$/) { $access = ""; }

    if (!$id) {
 $self->throw ("no database specified");
    }
    if (!_has_local_DB($db)) {
 $self->throw ("$db -- no such database available");
    }

    my $in = Bio::SeqIO->new( -file   => "golden $access $entry 2> /dev/null |",
         -format => _get_seq_format($db));
    my $seq = $in->next_seq();

#      $in->DESTROY();
```

```
#      if ($? >> 8) {
#   $self->throw ("$id -- no such entry in database $db");
#      }

    $self->throw ("$id -- no such entry in database $db") if (! defined $seq);

    return $seq;
}



sub _has_local_DB {

    my ($db) = @_;

    if (exists $AVAIL_LOCALDB{$db}) { return 1; }
    return 0;

}

sub _get_seq_format {

    my ($db) = @_;

    return $AVAIL_LOCALDB{$db};
}
```