Ensembl Assignment:

Write a perl script to do the following:

1. Connect to the Ensembl mus_musculus_core_26_33b database.
2. Retrieve a region of chrnm. III, from base 3e6 to 4e6.
3. Print out to a file named [yourname]_ensembl.out the following:
   a. all Gene IDs, start and stop positions, and strand (1 or -1),
   b. all Transcript IDs, start and stop positions, and strand,
   c. all Exon IDs, start and stop positions, and strand,
   d. Repeat information found in this region for all ‘AT_rich’ regions,
   e. and single-gapped DNA alignment information along with percent identity.

Ensembl uses a MySQL db to keep track of data in the following schema:

- **Sequence region tables** hold the genome sequences for sequence-level entities (e.g. BAC clones) and assembly information for top-level entities (e.g. chromosomes).
  
  The `seq_region` table provides information about every piece of DNA inside Ensembl that is intended for feature storing. Not every `seq_region` has to be associated with sequence. The name part should only contain the following characters [a-zA-Z.0-9].

  The `dna` table provides sequence data for a sequence region. In order to be retrievable by the API, the `seq_region` has to be connected to the `sequence_level` coordinate system. The length of the sequence has to be the length of the `seq_region`.

  The `dna_c` table can contain 2-bit compressed dna sequence. This is done by the API and should not be tried manually.

  The `coord_system` table describes a coordinate system. Typical examples would be `chromosome`, `contig`, `chunk`, `clone` etc. The version is meant to distinguish coordinate systems that have `seq_region`es of the same name with different content, like chromosome names in different assemblies. The rank indicates the level of context in this coordinate system. Rank 1 is the broadest e.g. `chromosome`, lower ranks have less context (meaning smaller `seq_regions`).

  The `assembly` table states which parts of `seq_region`es are exactly equal. It enables transformation of coordinates between `seq_region`es. Typically this contains how chromosomes are made of contigs, clones out of contigs, and chromosomes out of supercontigs. It allows you to artificially chunk chromosome sequence into smaller parts.

  Every `seq_region` can have arbitrary attributes associated with it. These are stored in the `seq_region_attrib` table. The API currently knows about the `toplevel` attribute. A `seq_region` with that attribute is considered to represent its sequence in the broadest possible context.

- **General Feature tables** store generic features that are annotated on the genome sequence.

  A `feature_id`. The column name should be `tablename_id`. Not all features need to have a `feature_id`. All features that have links in other tables need one; others might have one out of consistency reasons. In the future they may become mandatory.

  All features describe their sequence position with a link to the `seq_region` they are defined on. They have `seq_start` and `seq_end` on this region, which range from 1 to the length of the linked `seq_region`. Most will have strand information. The strand does not change the counting for the sequence position, which always happens from the 5-prime end of the `seq_region`.

  Many features will contain a displayable name (`display_label` or `name`). Future changes might make this more consistent.

  Almost all features link to an analysis entry.

  Special features contain any number of other attributes; `simple_feature` entries for example contain a `score` entry.

- **The Analysis table** provides information on individual analysis steps that were run to annotate features on the genome sequence.

  `created` is a date stamp created automatically when the analysis entry is stored.

  `db` contains the common name for a database, like "UniProt" or "EMBL". If the filename is different from this name it is stored in `db_file`. If this database is versioned this information should be stored in `db_version`.

  `program` contains the name for the binary that produced the feature. If the program cannot be located by name, its file system location should be stored in `program_file`. Use `program_version` if you have version information.
parameters contains the command line parameters for the file or general parameters for the module that was used.

module contains a perl module name that produced the features. Use module_version if you have version information.

gff_source and gff_feature are there to support gff dumps and should be filled according to the format.

- Common Feature tables contain essential feature annotation relevant for most species, including cDNA and protein alignments to the genome sequence and ab initio predicted transcripts.

The dna_align_feature table
  - seq_region_id, seq_region_start, seq_region_end, seq_region_strand: sequence position of the alignment.
  - hit_name the name or primary id of the aligned sequence in the database, described in the associated analysis.
  - hit_start, hit_end, hit_strand: sequence position of the aligned stretch of DNA on the hit_name sequence.
  - score, evalue, perc_ident selected quality numbers for the alignment. The API supports score as a filter criterion.
  - cigar_line a string that describes the alignment. Pieces of matching sequence M, deleted bases D and inserted bases I.

The protein_align_feature table
  - Describes an alignment between an external peptide sequence and a piece of Ensembl DNA sequence. It is equivalent to the dna_align_feature table apart from:
    - there is no hit_strand as protein sequence cannot have a negative strand
    - the interpretation of the hit coordinates, as they are amino acid counts.
    - the cigar_string uses DNA basepair counts for the alignment description.

The repeat_features and repeat_consensus tables
  - Repetitive regions on the genome are stored in this table. All repeats are further classified by the repeat_consensus table.
  - repeat_start, repeat_end and score are retrievable by the API but not used by it (the API cannot filter by this criteria). Repeats are retrievable by their different consensus sequence (type).

The prediction_transcripts table
  - A prediction transcript is a gene prediction of low quality where there is not much information provided (compared to the information for a gene, see below). Prediction transcripts link to one or more prediction exons and an analysis, which generated this prediction (and the prediction exons).

Prediction exons
  - rank The position of the exon, starting from 1, in the transcript from the 5 prime end.
  - start_phase describes whether the prediction exon starts with the first, second or third base in a codon.
    - Phase 0: starts with the first base
    - Phase 1: starts with the second base
    - Phase 2: starts with the third base of the codon
  - The score and pvalue are used to store scores for the prediction algorithm (no filter criteria).

- Gene-related tables contain Ensembl gene, transcript, protein and exon annotation.

  These tables reflect the Ensembl gene model, which states that a gene contains one or more transcripts, which in turn contain one or more exons. Transcripts may have a translation for protein-coding genes or may not if they are pseudogenes or RNA genes. All of the above objects can have a stable_id that Ensembl tries to map between different genebuilds. This is stored in separate tables because during the gene prediction process these IDs are not known and are uploaded later (as it is easier to upload than update statements).

Some table columns need further explanation:

gene.type currently contains either ensembl or pseudogene and is in need of standardisation (as does the analysis link).

gene.display_xref_id links to the xref entry that gives this gene an external name. This also defines the gene as known.

transcript.display_xref_id same as for the gene.

exon_transcript table builds a many-to-many relationship between transcripts and exons. The same exon can be shared by many transcripts (in the same gene). Rank count begins at 1 and goes from the 5 prime end of transcript.

exon.phase (values 0, 1, 2 or -1). The phase of the exon. If the first base in the exon is not coding its phase is -1.

exon.end_phase (values 0, 1, 2 and -1).
  - Phase of -1: the last base of the exon is non coding
  - Phase of 1: the last codon has one base only
- Phase of 2: the last codon has two bases
- Phase of 0: the last codon has three bases

- **External references** link Ensembl gene, transcript and protein predictions to other biological source databases. Ensembl supports the link of external database entries to internal objects. This is currently only done on translation objects, but can be easily extended. An external entry is described in the `xref` table and the database it comes from in `external_db`. Any synonyms for the external entry are stored in `external_synonym`.

  Links to Ensembl are in the `object_xref` table. Some links need additional information (attached to the link and not the external entry). Derived GO identifiers are qualified by the way they were derived (`go_xref.linkage_type`). External links that are derived from aligned peptides are annotated with the alignment details in `identity_xref`.

- **Miscellaneous Feature tables** contain all those features not organised in other tables. In general the Ensembl schema aims to give each feature type its own table. This is somewhat impractical if you have only a small number of new features to introduce or if the new feature does not add any extra functionality.

  A `misc_feature` is a general feature. It can have a location (`misc_feature.seq_region_id/start/end/strand`), a number of self defined attributes (`misc_attribute.value`) and self defined attribute types (`attrib_type.code`). Misc_features can belong to one or more sets of features (`misc_set`).

  The columns `attrib_type.code` and `misc_set.code` are used to identify a certain attribute type or misc set to the API. For display purposes the value in the name column should be used. The description column is optional.

- **Archive tables** contain mappings from expired Ensembl stable gene, transcript and protein IDs to the newer sets.

- **Marker tables** store information for STS markers with physical sequence evidence, as well as markers on genetic maps. An Ensembl marker is uniquely identified by a pair of primers. This is practical for Ensembl even if it deviates from the typical biological definition of a marker. A pair of primers is used to map a marker on the genome via a `marker_feature`. If a marker has more than one marker feature, it will be placed at more than one position on the genome. The `marker_feature.map_weight` column stores the number of times the marker maps to the genome. (This is obviously done after all `marker_features` are created).

  The marker’s name in Ensembl is the `display_marker_synonym_id`. If a marker in Ensembl is known under different names in other databases, these are stored in `marker_synonym`. The same identifier might be associated with different primer pairs and will have different `marker_id` in Ensembl.

  Ensembl markers are usually part of a mapping project that uses experimental methods to place them on the genome. A map represents a collection of experimentally placed markers and depending on the type of experiment the marker map location may have different forms (bands, radiation hybrid, cM) depending. An experimental mapping usually produces a chromosome name. If the markers used for the experiments are not defined by a primer pair, the actual name is provided as link to the synonym table.

### MySQL

Additionally, Ensembl would like to encourage users to directly extract information from our databases via SQL rather than downloading huge flat files. We offer a public MySQL interface at ensembldb.ensembl.org that accepts SQL queries as user 'anonymous'.

```sql
>mysql -u anonymous -h ensembldb.ensembl.org
mysql# show databases;
mysql# use [database];
mysql# show tables;
```

### Introduction of Perl

The Perl API provides a level of abstraction over the Ensembl databases and is used by the Ensembl web interface, pipeline, and genebuild systems. To external users the API may be useful to automate the extraction of particular data, to customize Ensembl to fulfill a particular purpose, or to store additional data in Ensembl. As a brief introduction this tutorial focuses primarily on the retrieval of data from the Ensembl databases.

The Perl API is only one of many ways of accessing the data stored in Ensembl. Additionally there is a Java API, the genome browser web interface, and the EnsMart system. If you are a Java programmer then the Java API is likely to be of more interest to you. Similarly, EnsMart may be a more appropriate tool for certain types of data mining.
Sources of Documentation

The Perl API has a decent set of code documentation in the form of PODs (Plain Old Documentation). This is documentation is mixed in with the actual code, but can be automatically extracted and formatted using some software tools. One version of this documentation is available at: [http://www.ensembl.org/info/software/Pdoc/](http://www.ensembl.org/info/software/Pdoc/)

For additional information you can contact ensembl-dev, the Ensembl development mailing list: (see [http://www.ensembl.org/info/about/contact.html](http://www.ensembl.org/info/about/contact.html)).

Perl

The Ensembl Perl API is compatible with Perl versions 5.6.0 and later. You can tell what version of Perl you are using by typing perl -v. This will give you version information like the following:

```
perl -v
This is perl, v5.8.4 built for i686-linux
```

Database Access

If you don't have, or don't want to install, the Ensembl database locally you can point your scripts at a publicly available database at the Sanger Centre. Use the following connection information in your scripts (where X_Y is the latest version of the database, for example 24_34e):

```
host       ensembldb.ensembl.org
dbname     homo_sapiens_core_X_Y
user       anonymous
```

DBI and DBD::mysql

You will need to install the Perl DBI and DBD::mysql modules from CPAN if they are not already present on your system. See the CPAN site (www.cpan.org) for installation instructions and further information on DBI and DBD::mysql.

Setting up the Environment

You can use the perl pragma use lib at the top of your scripts to point to the location of the perl modules you wish to use.

```
use lib '/usr/local/lib/perl5/site_perl/5.8.4/ensembl/modules';
```

Code Conventions

Several naming conventions are used throughout the API. Learning these conventions will aid in your understanding of the code.

Variable names are underscore separated all-lowercase words.

```
$slice, @exons, %exon_hash, $database_adaptor
```

Class and package names are mixed-case words that begin with capital letters.

```
Bio::EnsEMBL::GeneAdaptor, Bio::EnsEMBL::Exon, Bio::EnsEMBL::Slice,
Bio::EnsEMBL::DBSQL::DBAdaptor
```

Method names are entirely lowercase, underscore separated words. Class names in the method are an exception to this convention; these words begin with an uppercase letter and are not underscore separated. The word dbID is another exception which denotes the unique database identifier of an object. No method names begin with a capital letter, even if they refer to a class.

```
fetch_all_by_Slice, get_all_Genes, traslation, fetch_by_dbID
```

Method names that begin with a an underscore '_' are intended to be private and should not be called externally from the class in which they are defined.

```
ObjectAdaptors are responsible for the creation of various objects. The adaptor should be named after the object it creates, and the methods responsible for the retrieval of these objects should all start with the word fetch. All of the fetch methods should return only objects of the type that the adaptor creates. Therefore the object name is not required in the method name. For example, all fetch methods in the GeneAdaptor return Gene objects. Non-adaptor methods generally avoid the use of the word fetch.
```
Method which begin with get_all or fetch_all return references to lists. Many methods in Ensembl pass lists by reference, rather than by value, for efficiency. This takes some getting used to, but it results in more efficient code, especially when very large lists are passed around (as they often are in Ensembl).

get_all_Transcripts, fetch_all_by_Slice, get_all_Exons

The following examples demonstrate some of Perl’s list reference syntax. You do not need to understand the API concepts in this example. The important thing to note is the language syntax; the concepts will be described later.

```perl
# fetch all clones from the slice adaptor (returns listref)
my $clones_ref = $slice_adaptor->fetch_all('clone');

# if you want a copy of the referenced array, do this:
my @clones = @$clones_ref;

# get the first clone from the list via the reference:
my $first_clone = $clones_ref->[0];

# another way of getting the same thing:
($first_clone) = @$clones_ref;

# iterate through all of the genes on a clone
foreach my $gene (@{$first_clone->get_all_Genes()}) {
  print $contig->stable_id() . "\n";
}

# another way of doing the same thing:
my $genes = $first_clone->get_all_Genes();
foreach my $contig (@$genes) {
  print $contig->name . "\n";
}

# retrieve a single Slice object (not a listref)
$clone = $slice_adaptor->fetch_by_region('clone', 'AL031658.11');
# no dereferencing needed:
print $clone->seq_region_name() . "\n";
```

**Connecting to the Database - The DBAdaptor**

All data used and created by Ensembl is stored in a MySQL relational database. If you want to access this database the first thing you have to do is to connect to it. This is done behind the scenes by Ensembl using the DBI module. You will need to know three things before you start:

- **host** - the hostname where the Ensembl database lives
- **dbname** - the name of the Ensembl database
- **user** - the username to access the database

First, we need to import any Perl modules that we will be using. Since we need a connection to an Ensembl database we first have to import the DBAdaptor modules that we use to establish this connection. Almost every Ensembl script that you will write will contain a use statement like the following:

```perl
use Bio::EnsEMBL::DBSQL::DBAdaptor;
```

Then we set the some variables containing the location of the database:

```perl
my $host = 'ensembldb.ensembl.org';
my $user = 'anonymous';
my $dbname = 'homo_sapiens_core_20_34c';
```

Now we can make a database connection:

```perl
my $db = new Bio::EnsEMBL::DBSQL::DBAdaptor(
  -host => $host,
  -user => $user,
  -dbname => $dbname);
```
We've made a connection to an Ensembl database and passed parameters in using the -attribute => 'somevalue' syntax present in many of the Ensembl object constructors. Formatted correctly, this syntax lets you see exactly what arguments and values you are passing.

In addition to the parameters provided above the optional port, driver and pass parameters can be used specify the TCP port to connect via, the type of database driver to use, and the password to use respectively. These values have sensible defaults and can often be omitted.

### Object Adaptors

Before we launch into the ways the API can be used to retrieve and process data from the Ensembl databases it is best to mention the fundamental relationships the Ensembl objects have with the database.

The Ensembl API allows manipulation of the database data through various objects. For example, some of the more heavily used objects are the Gene, Slice and Exon objects. More details of how to effectively use these objects will be covered later. These objects are retrieved and stored in the database through the use of object adaptors. Object adaptors have internal knowledge of the underlying database schema and use this knowledge to fetch, store and remove objects (and data) from the database. This way you can write code and use the Ensembl API without having to know anything about the underlying databases you are using. The database adaptor that we created in the previous section is a special adaptor which has the responsibility of maintaining the database connection and creating other object adaptors.

Object adaptors are obtained from the main database adaptor via a suite of methods with the naming convention get_ObjectAdaptor. To obtain a SliceAdaptor or a GeneAdaptor (which retrieve Slice and Gene objects) do the following:

```perl
my $gene_adaptor = $db->get_GeneAdaptor();
my $slice_adaptor = $db->get_SliceAdaptor();
```

Don't worry if you don't immediately see how useful this could be. Just remember that you don't need to know anything about how the database is structured, but you can retrieve the necessary data (neatly packaged in objects) by asking for it from the correct adaptor. Throughout the rest of this document we are going to work through the ways the Ensembl objects can be used to derive the information you want.

### Slices

A Slice object represents a single continuous region of a genome. Slices can be used to obtain sequence, features or other information from a particular region of interest. To retrieve a Slice it is first necessary to get a SliceAdaptor:

```perl
my $slice_adaptor = $db->get_SliceAdaptor();
```

The SliceAdaptor provides several ways to obtain Slices, but we will start with the fetch_by_region method which is the most commonly used. This method takes numerous arguments but most of them are optional. In order, the arguments are:

- coord_system_name
- seq_region_name
- start
- end
- strand
- coord_system_version

The following are several examples of how to use the fetch_by_region method:

- # obtain a slice of the entire chromosome X:
  ```perl
  my $slice = $slice_adaptor->fetch_by_region('chromosome', 'X');
  ```

- # obtain a slice of the entire clone AL359765.6
  ```perl
  $slice = $slice_adaptor->fetch_by_region('clone','AL359765.6');
  ```

- # obtain a slice of an entire NT contig
  ```perl
  $slice = $slice_adaptor->fetch_by_region('supercontig',
                     'NT_011333');
  ```

- # obtain a slice of 1-2MB of chromosome 20
  ```perl
  $slice = $slice_adaptor->fetch_by_region('chromosome', '20',
                     1e6, 2e6);
  ```

Another useful way to obtain a Slice is with respect to a gene:

```perl
my $slice = $slice_adaptor->fetch_by_gene_stable_id('ENSG00000099889', 5000);
```

This will return a Slice that contains the sequence of the gene specified by its stable Ensembl id. It also returns 5000bp of flanking sequence at both the 5' and 3' ends, which is useful if you are interested in the environs that a gene inhabits. You needn't have the flanking sequence if you don't want it - in this case set the number of flanking bases to 0 or omit the second argument entirely. Note that for historical reasons the fetch_by_gene_stable_id method always returns a slice on the forward strand even if the gene is on the reverse strand.
To retrieve a set of slices from a particular coordinate system the fetch_all method can be used:

```perl
@slices = @{$slice_adaptor->fetch_all('chromosome')};
```

To retrieve a set of every BAC clone in the database:

```perl
@slices = @{$slice_adaptor->fetch_all('clone')};
```

For certain types of analysis it is necessary to break up regions into smaller manageable pieces. The method split_Slices can be imported from the Bio::EnsEMBL::Utils::Slice module to break up larger slices into smaller component slices.

```perl
use Bio::EnsEMBL::Utils::Slice qw(split_Slices);
```

```perl
my $slices = $slice_adaptor->fetch_all('chromosome');
# basepairs overlap between returned slices
my $overlap = 0;
# maximum size of returned slices
my $max_size = 100000;
# break chromosomal slices into smaller 100k component slices
$slices = split_Slices($slices, $max_length, $overlap);
```

To obtain sequence from a slice the seq or subseq methods can be used:

```perl
my $sequence = $slice->seq();
print "$sequence\n";
$sequence = $slice->subseq(100, 200);
```

We can query the Slice for information about itself:

```perl
# coord_system() returns a Bio::EnsEMBL::CoordSystem object
my $coord_sys = $slice->coord_system()->name();
my $seq_region = $slice->seq_region_name();
my $start = $slice->start();
my $end = $slice->end();
my $strand = $slice->strand();
print "Slice: $coord_sys $seq_region $start-$end ($strand)\n";
```

Many object adaptors can provide a set of features which overlap a slice. The Slice itself also provides a means to obtain features which overlap its region. The following are two ways to obtain a list of genes which overlap a Slice:

```perl
my @genes = @{$gene_adaptor->fetch_all_by_Slice($slice)};
# another way of doing the same thing:
@genes = @{$slice->get_all_Genes()};
```

## Features

Features are objects in the database which have a defined location on the genome. All features in Ensembl inherit from the Bio::EnsEMBL::Feature class and have the following location defining attributes: start, end, strand, slice.

In addition to locational attributes all features have internal database identifiers accessed via the method dbID. All feature objects can be retrieved from their associated object adaptors using a Slice object or the feature's internal identifier (dbID). The following example illustrates how Transcript features and DnaDnaAlignFeature features can be obtained from the database.

```perl
my $tr_adaptor = $db->get_TranscriptAdaptor();
my $daf_adaptor = $db->get_DnaAlignFeatureAdaptor();
```
# fetch all of the transcripts overlapping chr20 10-11MB
my $transcripts = $tr_adaptor->fetch_all_by_Slice($slice);
foreach my $tr (@$transcripts) {
    my $dbID   = $tr->dbID();
    my $start  = $tr->start();
    my $end    = $tr->end();
    my $strand = $tr->strand();
    my $stable_id = $tr->stable_id();
    print "Transcript $stable_id [$dbID] $start-$end($strand)\n";
}

# fetch all of the dna-dna alignments overlapping chr20 10-11MB
my $dafs = $daf_adaptor->fetch_all_by_Slice($slice);
foreach my $daf (@$dafs) {
    my $dbID     = $daf->dbID();
    my $start    = $daf->start();
    my $end      = $daf->end();
    my $strand   = $daf->strand();
    my $hseqname = $daf->hseqname();
    print "DNA Alignment $hseqname [$dbID] $start-$end($strand)\n";
}

# fetch a transcript by its internal identifier
my $transcript = $tr_adaptor->fetch_by_dbID(100);

# fetch a dnaAlignFeature by its internal identifiers
my $dna_align_feat = $daf_adaptor->fetch_by_dbID(100);

All features also have the methods transform, transfer, and project which are described in detail in the Transform, Transfer and Project sections of this tutorial.

---

**Genes, Transcripts, Exons**

Genes, Exons and Transcripts are also features and can be treated in the same way as any other feature within Ensembl. A Transcript in Ensembl is a grouping of Exons. A Gene in Ensembl is a grouping of Transcripts which share any overlapping (or partially overlapping) Exons. Transcripts also have an associated Translation object which defines the UTR and CDS composition of the Transcript. Introns are not defined explicitly in the database but can be obtained by the transcript method get_all_Introns.

Like all Ensembl features the start of an Exon is always less than or equal to the end of the Exon, regardless of the strand it is on. The start of the Transcript is the start of the first Exon of a forward strand Transcript or the start of the last Exon of a reverse strand Transcript. The start and end of a Gene are defined to be the lowest start value of its Transcripts and the highest end value respectively.

Genes, Translations, Transcripts and Exons all have stable identifiers. These are identifiers that are assigned to Ensembl's predictions, and maintained in subsequent releases. For example, if a Transcript (or a sufficiently similar Transcript) is re-predicted in a future release then it will be assigned the same stable identifier as its predecessor.

The following is an example of the retrieval of a set of Genes, Transcripts and Exons:

```
sub feature2string {
    my $f = shift;
    my $stable_id = $f->stable_id();
    my $seq_region = $f->slice->seq_region_name();
    my $start = $f->start();
    my $end = $f->end();
    my $strand = $f->strand();

    return "$stable_id : $seq_region:$start-$end ($strand)"
}
```
$slice_adaptor = $db->get_SliceAdaptor();
$slice = $slice_adaptor->fetch_by_region('chromosome','X',
    1e6,10e6);

foreach my $gene (@{$slice->get_all_Genes()}) {
    my $gstring = feature2string($gene);
    print "$gstring
";
    foreach my $trans (@{$gene->get_all_Transcripts()}) {
        my $tstring = feature2string($trans);
        print " $tstring
";
        foreach my $exon (@{$trans->get_all_Exons()}) {
            my $estring = feature2string($exon);
            print "    $estring
";
        }
    }
}

In addition to the methods which are present on every feature, the transcript class has many other methods which are
commonly used. Several methods can be used to obtain transcript related sequences. For historical reasons some of these
methods return strings while others return Bio::Seq objects. The following example demonstrates the use of some of these
methods:

# spliced_seq returns the concatenation of the exon sequences.
# This is the cDNA of the transcript
print "cDNA: ", $trans->spliced_seq(), "\n";

# translateable_seq returns only the CDS of the transcript
print "CDS: ", $trans->translateable_seq(), "\n";

# UTR sequences are obtained via the five_prime_utr and
# three_prime_utr methods
my $fiv_utr = $trans->five_prime_utr();
my $thr_utr = $trans->three_prime_utr();

print ($fiv_utr) ? $fiv_utr->seq() : 'No 5' UTR', "\n";
print ($thr_utr) ? $thr_utr->seq() : 'No 3' UTR', "\n";

# The peptide sequence is obtained from the translate method
# undef is returned if this transcript is non-coding
my $pep = $trans->translate();
print ($pep) ? $pep->seq() : 'No Translation', "\n";

### Translations and ProteinFeatures

Translation objects and peptide sequence can be extracted from a Transcript object. It is important to remember that some
Ensembl transcripts are non-coding (pseudogenes, ncRNAs, etc.) and have no translation. The primary purpose of a
Translation object is to define the CDS and UTRs of its associated Transcript object. Peptide sequence is obtained directly
from a Transcript objectâ€”not a Translation object as might be expected. The following example obtains the peptide
sequence of a Transcript and the Translation’s stable identifier:

my $stable_id = 'ENST00000044768';
my $transcript_adaptor = $db->get_TranscriptAdaptor();
my $transcript =
    $transcript_adaptor->fetch_by_stable_id($stable_id);
print $transcript->translation()->stable_id(), "\n";
print $transcript->translate()->seq(), "\n";

ProteinFeatures are features which are on an amino acid sequence rather than a nucleotide sequence. The method
get_all_ProteinFeatures can be used to obtain a set of protein features from a Translation object.

$translation = $transcript->translation();
my $protein_feats = $translation->get_all_ProteinFeatures();

foreach my $pf (@$protein_feats) {
    my $logic_name = $pf->analysis()->logic_name();
    print $pf->start(), '-', $pf->end(), ' ', $logic_name, ' ',
               $pf->interpro_ac(), ' ', $pf->idesc(), "\n";
}

If only the protein features created by a particular analysis are desired the name of the analysis can be provided as an argument. To obtain the subset of features which are considered to be 'domain' features the convenience method get_all_DomainFeatures can be used:

my $seg_feats = $translation->get_all_ProteinFeatures('Seg');
my $domain_feats = $translation->get_all_DomainFeatures();

PredictionTranscripts

PredictionTranscripts are the results of ab initio gene finding programs that are stored in Ensembl. Example programs include Genscan and SNAP. Prediction transcripts have the same interface as normal transcripts and thus they can be used in the same way.

my $ptranscripts = $slice->get_all_PredictionTranscripts;

foreach my $ptrans (@$ptranscripts) {
    my $exons = $ptrans->get_all_Exons();
    my $type = $ptrans->analysis->logic_name();
    print "$type prediction has " . scalar(@$exons) . " exons\n";

    foreach my $exon (@$exons) {
        print $exon->start . ' - ' . $exon->end . ' : ' . $exon->strand . ' ' . $exon->phase . "\n";
    }
}

Alignment Features

Two types of alignments are stored in the core Ensembl database: alignments of DNA sequence to the genome and alignments of peptide sequence to the genome. These can be retrieved as DnaDnaAlignFeatures and DnaPepAlignFeatures respectively. A single gapped alignment is represented by a single feature with a CIGAR line. A CIGAR line is a concise representation of a gapped alignment as single string containing letters M (match) D (deletion), and I (insertion) prefixed by integer lengths (the number may be omitted if it is 1). A gapped alignment feature can be broken into its component ungapped alignments by the method ungapped_features which returns a list of FeaturePair objects. The following example shows the retrieval of some alignment features.

# retrieve dna-dna alignment features from the slice region
my $feats = $slice->get_all_DnaAlignFeatures('Vertrna');
print_align_features($feats);

# retrieve protein-dna alignment features from the slice region
$feats = $slice->get_all_ProteinAlignFeatures('Swall');
print_align_features($feats);

sub print_align_features {
    my $feats = shift;

    foreach my $feat (@$feats) {
        print_feature_pairs([$feat]);

        print "Percent identity: ", $feat->percent_id(), "\n";
        print "
"
Repeats

Repetitive regions found by RepeatMasker and TRF (Tandem Repeat Finder) are represented in the Ensembl database as RepeatFeatures. Short non-repetitive regions between repeats are found by the program Dust and are also stored as RepeatFeatures. RepeatFeatures can be retrieved and used in the same way as other Ensembl features.

```perl
my $repeats = $slice->get_all_RepeatFeatures();
foreach my $repeat (@$repeats) {
    print $repeat->display_id(), " ",
    $repeat->start(), "-", $repeat->end(), "\n";
}
```

RepeatFeatures are used to perform repeat masking of the genomic sequence. Hard or softmasked genomic sequence can be retrieved from Slice objects using the get_repeatmasked_seq method. Hardmasking replaces sequence in repeat regions with Ns. Softmasking replaces sequence in repeat regions with lowercase sequence.

```perl
my $unmasked_seq = $slice->seq();
my $hardmasked_seq = $slice->get_repeatmasked_seq();
my $softmasked_seq = $slice->get_repeatmasked_seq(undef, 1);
my $tandem_masked_seq = $slice->get_repeatmasked_seq(['TRF'], 1);
```

Markers

Markers are imported into the Ensembl database from UniSTS and several other sources. A marker in Ensembl consists of a pair of primer sequences, an expected product size and a set of associated identifiers known as synonyms. Markers are placed on the genome electronically using an analysis program such as ePCR and their genomic positions are retrievable as MarkerFeatures. Map locations (genetic, radiation hybrid and in situ hybridization) for markers obtained from actual experimental evidence are also accessible.

Markers can be fetched via their name from the MarkerAdaptor.

```perl
my $marker_adaptor = $db->get_MarkerAdaptor();
my ($marker) = @{$marker_adaptor->fetch_all_by_synonym('D9S1038E')};
foreach my $synonym ($marker->get_all_MarkerSynonyms()) {
    print the various names associated with the same marker
}
```
print $synonym->source(), ':
' if($synonym->source());
print $synonym->name(), ' 
'}

# print the primer info
print "\left primer: ", $marker->left_primer(), "\n";
print "right primer: ", $marker->right_primer(), "\n";
print "product size: ", $marker->min_primer_dist(), ' - ',
$marker->max_primer_dist(), "\n";

# print out genetic/RH/FISH map information
print "Map locations:\n";
foreach my $map_loc (@{$marker->get_all_MapLocations()}) {
  print "  ", $map_loc->map_name(), ' ',
$map_loc->chromosome_name(), ' ',
$map_loc->position(), "\n";
}

MarkerFeatures, which represent genomic positions of markers, can be retrieved and manipulated in the same way as other
Ensembl features.

# obtain the positions for an already retrieved marker
foreach my $marker_feat (@{$marker->get_all_MarkerFeatures()}) {
  print $marker_feat->seq_region_name(),
$marker_feat->start(), '-', $marker_feat->end(), "\n";
}

# retrieve all marker features in a given region
my $marker_feats = $slice->get_all_MarkerFeatures();
foreach my $marker_feat (@$marker_feats) {
  print $marker_feat->display_id(), " ",
$marker_feat->seq_region_name(),
$marker_feat->start(), '-', $marker_feat->end(), "\n";
}

MiscFeatures

MiscFeatures are features with arbitrary attributes which are placed into arbitrary groupings. MiscFeatures can be retrieved as
any other feature and are classified into distinct sets by a set code. Generally it only makes sense to retrieve all features which
have a particular set code because very diverse types of MiscFeatures are stored in the database.

MiscFeature attributes are represented by Attribute objects and can be retrieved via a get_all_Attributes method.

The following example retrieves all MiscFeatures representing ENCODE regions on a given slice and prints out their attributes:

my $enc_regions = $slice->get_all_MiscFeatures('encode_regions');
foreach my $enc_region (@$enc_regions) {
  foreach my $attr (@{$enc_region->get_all_Attributes()}) {
    print $attr->name(), ':', $attr->value(), "\n";
  }
}

This example retrieves all misc features representing a BAC clone via its name and prints out their location and other
information:

my $mfa = $db->get_MiscFeatureAdaptor();
my $clones = $mfa->fetch_all_by_attribute_type_value('Name',
'RP11-62N12');

foreach my $clone (@$clones) {
  my $slice = $clone->slice();
  print $slice->coord_system->name(), ' ',
$slice->seq_region_name(), ' ',
$clone->start(), ' - ', $clone->end(), "\n";
}
foreach my $a (@{$clone->get_all_Attributes()}) {
    print '  ', $a->name, ':', $a->value, "\n";
}
}

**External References**

Ensembl cross references its genes, transcripts and translations with identifiers from other databases. A DBEntry object represents a cross reference and is often referred to as an 'xref'. The following code snippet retrieves and prints DBEntries for a gene, its transcripts and its translations:

```perl
# define a helper subroutine to print DBEntries
sub print_DBEntries {
    my $db_entries = shift;
    foreach my $dbe (@$db_entries) {
        print $dbe->dbname(), " - ", $dbe->display_id(), "\n";
    }
}
print "GENE ", $gene->stable_id(), "\n";
print_DBEntries($gene->get_all_DBEntries());

foreach my $trans(@{$gene->get_all_Transcripts()}){
    print "TRANSCRIPT ", $trans->stable_id(), "\n";
    print_DBEntries($trans->get_all_DBEntries());
    # watch out: pseudogenes have no translation
    if($trans->translation()) {
        my $transl = $trans->translation();
        print "TRANSLATION ", $transl->stable_id(), "\n";
        print_DBEntries($transl->get_all_DBEntries());
    }
}
```

Often it is useful to obtain all of the DBEntries associated with a gene and its associated transcripts and translation as in the above example. As a shortcut to calling get_all_DBEntries on all of the above objects the get_all_DBLinks method can be used instead. The above example could be shortened by using the following:

```perl
print_DBEntries($gene->get_all_DBLinks());
```

**Coordinates**

We have already discussed the fact that Slices and features have coordinates, but we have not defined exactly what these coordinates mean.

Ensembl, and many other bioinformatics applications, use inclusive coordinates which start at 1. The first nucleotide of a DNA sequence is 1 and the first amino acid of a peptide sequence is also 1. The length of a sequence is defined as end - start + 1.

In some rare cases inserts are specified with a start which is one greater than the end. For example a feature with a start of 10 and an end of 9 would be a zero length feature between basepairs 9 and 10.

Slice coordinates are relative to the start of the underlying DNA sequence region. The strand of the Slice represents its orientation relative to the default orientation of the sequence region. By convention the start of the Slice is always less than or equal to the end - 1, and does not vary with its strandedness. Most Slices you will encounter will have a strand of 1, and this is what we will consider in our examples. It is legal to create a Slice which extends past the boundaries of a sequence region. Sequence retrieved from regions where the sequence is not defined will consist of Ns.

All features retrieved from the database have an associated Slice (accessible via the slice method). A feature's coordinates are always relative to this associated Slice, i.e. the start and end attributes define a feature's position relative to the start of the Slice the feature is on (or the end of the Slice if it is a negative strand Slice). The strand attribute of a feature is relative to the strand of the Slice. By convention the start of a feature is always less than or equal to the end of the feature regardless of its strand (except in the case of an insert). It is legal to have features with coordinates which are less than one or greater than the length of the slice. Such cases are common when features that partially overlap a slice are retrieved from the database.

Consider, for example, the following figure of two features associated with a Slice:

```
[-----] (Feature A)
```
The Slice itself has a start of 2, an end of 13, and a length of 12 even though the underlying sequence region only has a length of 11. Retrieving the sequence of such a slice would give the string CTAAATCTTGNN -- the undefined region of sequence is represented by Ns. Feature A has a start of 0, an end of 2, and a strand of 1. Feature B has a start of 3, an end of 6, and a strand of -1.

**Coordinate Systems**

Sequences stored in Ensembl are associated with coordinate systems. What the coordinate systems are varies from species to species. For example, the homo_sapiens database has the following coordinate systems: contig, clone, supercontig, chromosome. Sequence and features may be retrieved from any coordinate system despite the fact they are only stored internally in a single coordinate system. The database stores the relationship between these coordinate systems and the API provides means to convert between them. The API has a CoordSystem object and object adaptor, however, these are most often used internally. The following example fetches a chromosome coordinate system object from the database:

```perl
my $csa = $db->get_CoordSystemAdaptor();
my $cs = $csa->fetch_by_name('chromosome');

print "Coord system: ". $cs->name() . " " . $cs->version . "\n";
```

A coordinate system is uniquely defined by its name and version. Most coordinate systems do not have a version, and the ones that do have a default version, so it is usually sufficient to use only the name when requesting a coordinate system. For example, chromosome coordinate systems have a version which is the assembly that defined the construction of the coordinate system. The version of human chromosome coordinate system might be NCBI33 or NCBI34.

Slice objects have an associated CoordSystem object and a seq_region_name that uniquely defines the sequence that they are positioned on. You may have noticed that the coordinate system of the sequence region was specified when obtaining a Slice in the fetch_by_region method. Similarly the version may also be specified (though it can almost always be omitted):

```perl
$slice = $slice_adaptor->fetch_by_region('chromosome', 'X', 1e6, 10e6, 'NCBI33');
```

Sometimes it is useful to obtain full Slices of every sequence in a given coordinate system; this may be done using the SliceAdaptor method fetch_all:

```perl
my @chromosomes = @{$slice_adaptor->fetch_all('chromosome')};
my @clones = @{$slice_adaptor->fetch_all('clone')};
```

Now suppose that you wish to write code which is independent of the species used. Not all species have the same coordinate systems; the available coordinate systems depends on the style of assembly used for that species (WGS, clone-based, etc.). You can obtain the list of available coordinate systems for a species using the CoordSystemAdaptor and there is also a special pseudo-coordinate system named toplevel. The toplevel coordinate system is not a real coordinate system, but is used to refer to the highest level coordinate system in a given region. The toplevel coordinate system is particulary useful in genomes that are incompletely assembled. For example, the latest zebrafish genome consists of a set of assembled chromosomes, and a set of supercontigs that are not part of any chromosome. In this example, the toplevel coordinate system sometimes refers to the chromosome coordinate system and sometimes to the supercontig coordinate system depending on the region it is used in.

```perl
#list all coordinate systems in this database:
my @coord_systems = @{$csa->fetch_all()};
foreach $cs (@coord_systems) {
    print "Coord system: ". $cs->name() . " " . $cs->version . "\n";
}

#get all slices on the highest coordinate system:
my @slices = @{$slice_adaptor->fetch_all('toplevel')};
```

**Transform**

Features on a Slice in a given coordinate system may be moved to another slice in the same coordinate system or to another coordinate system entirely. This is useful if you are working with a particular coordinate system but you are interested in obtaining the features coordinates in another coordinate system.
The method transform can be used to move a feature to any coordinate system which is in the database. The feature will be placed on a Slice which spans the entire sequence that the feature is on in the requested coordinate system.

```perl
if(my $new_feature = $feature->transform('clone')) {
    print "Feature's clonal position is: ",
    $new_feature->slice->seq_region_name(), ' ',
    $new_feature->start(), '-', $feature->end(), ' (',
    $new_feature->strand(), ")\n";
} else {
    print "Feature is not defined in clonal coordinate system\n";
}
```

The transform method returns a copy of the original feature in the new coordinate system, or undef if the feature is not defined in that coordinate system. A feature is considered to be undefined in a coordinate system if it overlaps an undefined region or if it crosses a coordinate system boundary. Take for example the tiling path relationship between chromosome and contig coordinate systems:

```
|~~~~~~~| (Feature A) |~~~~| (Feature B)
(ctg 1) [=============] (ctg 2) (----==========
(ctg 3) (==============)] (ctg3)
```

Both Feature A and Feature B are defined in the chromosomal coordinate system described by the tiling path of contigs. However, Feature A is not defined in the contig coordinate system because it spans both Contig 1 and Contig 2. Feature B, on the other hand, is still defined in the contig coordinate system.

The special toplevel coordinate system can also be used in this instance to move the feature to the highest possible coordinate system in a given region:

```perl
my $new_feature = $feature->transform('toplevel);
```

```perl
print "Feature's toplevel position is: ",
$new_feature->slice->coord_system->name(), ' ',
$new_feature->slice->seq_region_name(), ' ',
$new_feature->start(), '-', $feature->end(), ' (',
$new_feature->strand(), ")\n";
```

**Transfer**

Another method that is available on all Ensembl features is the transfer method. The transfer method is similar to the previously described transform method, but rather than taking a coordinate system argument it takes a Slice argument. This is useful when you want a feature's coordinates to be relative to a certain region. Calling transform on the feature will return a copy of the which is shifted onto the provided Slice. If the feature would be placed on a gap or across a coordinate system boundary, then undef is returned instead. It is illegal to transfer a feature to a Slice on a sequence region which cannot be placed on. For example, a feature which is on chromosome X cannot be transferred to a Slice on chromosome 20 and attempting to do so will raise an exception. It is legal to transfer a feature to a Slice on which it has coordinates past the slice end or before the slice start. The following example illustrates the use of the transfer method:

```perl
$slice = $slice_adaptor->fetch_by_region('chromosome', '2', 1e6, 2e6);
$new_slice = $slice_adaptor->fetch_by_region('chromosome', '2', 1_500_000, 2_000_000);

foreach $sf (@{$slice->get_all_SimpleFeatures('Eponine')}) {
    print "Before: ", $sf->start, '-', $sf->end, "\n";
    $new_feat = $sf->transfer($new_slice);
    if(!$new_feat) {
        print "Could not transfer feature\n";
    } else {
        print "After: ", $new_feat->start, '-', $new_feat->end, "\n";
    }
}
```

In the above example a Slice from another coordinate system could also have been used, provided you had an idea about what sequence region the features would be mapped to.
**Project**

When moving features between coordinate systems it is usually sufficient to use the transfer or transform methods. Sometimes, however, it is necessary to obtain coordinates in another coordinate system even when a coordinate system boundary is crossed. Even though the feature is considered to be undefined in this case, the feature's coordinates can still be obtained in the requested coordinate system using the project method.

Both Slices and features have their own project methods, which take the same arguments and have the same return values. The project method takes a coordinate system name as an argument and returns a reference to a list of ProjectionSegment objects. A projection segment has three attributes: from_start, from_end, to_Slice. The from_start and from_end methods return integers representing the part of the feature or Slice that is used to form that part of the projection. The to_Slice method returns a Slice object representing the part of the region that the slice or feature was projected to. The following example illustrates the use of the project method on a feature. The project method on a Slice can be used in the same way. As with the Feature transform method the pseudo coordinate system toplevel can be used to indicate you wish to project to the highest possible level.

```perl
$projection = $feature->project('clone');
my $seq_region = $feature->seq_region_name();
my $start = $feature->start();
my $end = $feature->end();
my $strand = $feature->strand();

print "Feature at: $seq_region $start-$end ($strand) projects ".
  "to\n";

foreach my $segment (@$projection) {
    my $to_slice = $segment->to_Slice();
    my $to_seq_region = $to_slice->seq_region_name();
    my $to_start = $to_slice->start();
    my $to_end = $to_slice->end();
    my $to_strand = $to_slice->strand();
    print "$to_seq_region $to_start-$to_end ($to_strand)\n";
}
```

**Feature Convenience Methods**

We have described how a feature's position on the genome is defined by an associated Slice and a start, end, and strand on that slice. Often it is more convenient to retrieve a feature's absolute position on the underlying sequence region rather than its relative position on a Slice. For convenience, a number of methods are provided that can be used to easily obtain a feature's absolute coordinates.

```perl
# shortcuts to doing $feat->slice()->coord_system()->name()
# and $feat->slice()->seq_region_name();
print $feat->coord_system_name(), ' ', $feat->seq_region_name(), ' ';

# get the feature's position on the sequence region
print $feature->seq_region_start(), '-', $feature->seq_region_end(),
  '(', $feature->seq_region_strand(), ')
"\n";

Another useful method is display_id. This will return a string that can be used as the name or identifier for a particular feature. For a gene or transcript, this method would return the stable_id, for an alignment feature, this would return the hit sequence name (hseqname), etc.

```perl
# display_id returns a suitable display value for any feature type
print $feat->display_id(), "\n";
```

The feature_Slice method will return a Slice which is the exact overlap of the feature the method was called on. This slice can then be used to obtain the underlying sequence of the feature or to retrieve other features that overlap the same region, etc.

```perl
$feat_slice = $feat->feature_Slice();
```

```perl
# print the sequence of the feature region
print $feat_slice->seq(), "\n";
```

```perl
# print the sequence of the feature region + 5000bp flanking
```
The Registry

The registry is a convenient storage/retrieval area for all the adaptors and provides an easy way to access them. If you have an Ensembl Web Server setup then you can automatically load all its adaptors with the load_registry_with_web_adaptors method from the Registry module.

```perl
use Bio::EnsEMBL::Registry;
my $reg = "Bio::EnsEMBL::Registry";
$reg->load_registry_with_web_adaptors();
my $ga = $reg->get_adaptor("Homo_sapiens", "estgene", "Gene");
my $gene = $ga->fetch_by_stable_id("ENSESTG0000000015126");
print $gene->seq()."\n";
```

The above gives an example of using the database data held in the Ensembl Web Server to ease the maintainance of code as we do not need to add the host, database name, host etc as this will already be set up. Plus it should now be more readable.

Another example of a general script is given below and takes four arguments the species, chromosome, start and end. This script will print out all the gene names with their start and end points and from which group database they were found for all genes found on the named chromosome between the start and end points specified.

```perl
#test2.pl
use Bio::EnsEMBL::Registry;
my $reg = "Bio::EnsEMBL::Registry";
my ($species, $chrom, $start, $end) = @ARGV;
die("Error species chrom start and end needed\n") unless defined($end);
$reg->load_registry_with_web_adaptors();
$species = $reg->get_alias($species);
my @dbs = $reg->get_all_DBAdaptors();
foreach my $db (@dbs){
    if($db->species eq $species){
        my $slice_adap = $reg->get_adaptor($db->species, $db->group, "Slice");
        if(defined($slice_adap)){
            my $slice = $slice_adap->fetch_by_region('chromosome', $chrom, $start, $end);
            foreach $gene ( @{$slice->get_all_Genes} ) {
                my $gene2= $gene->transform('chromosome');
                my $name =  $gene->stable_id() || $gene->type() . "." . $gene->dbID() ;
                print $db->group . "\t" . $name . "\t" . $gene2->start . "\t" . $gene2->end."
";
            }
        }
    }
}
```

Note the path to the SiteDefs.pm module must first be added to the PERL5LIB environment variable if you want to use the load_registry_with_web_adaptors method. The next example will list all the databases that have been set up for the Ensembl Web Server:-
use Bio::EnsEMBL::Registry;
my $reg = "Bio::EnsEMBL::Registry";

$reg->load_registry_with_web_adaptors();

my @dbs = $reg->get_all_DBAdaptors();
foreach my $db (@dbs) {
    print $db->species() . "\t" .
    $db->group() . "\t" .
    $db->dbc->dbname() . "\t" .
    $db->dbc->host() . "\t" .
    $db->dbc->port() . "\n";
}

To ensure the Registry stores the Adaptors in an organised way two new arguments have been added to the DBAdaptor new method, these are species and group. Default values are used if these are not given. Configuration scripts can be written to enable an easy setup of the Registry for all scripts to use. Below is an example of a configuration script.

use Bio::EnsEMBL::DBSQL::DBAdaptor;
use Bio::EnsEMBL::Utils::ConfigRegistry;
my $reg = "Bio::EnsEMBL::Registry";

my @a = ('H_Sapiens', 'homo sapiens', 'human',
    'Homo_Sapiens','Homo sapiens');

Bio::EnsEMBL::Utils::ConfigRegistry->
    add_alias( -species => "Homo_sapiens",
                        -alias  => \@a);

new Bio::EnsEMBL::DBSQL::DBAdaptor(
    -species => "Homo_sapiens",
    -group => "core",
    -host => 'host1',
    -user => 'anonymous',
    -dbname => 'homo_sapiens_core_24_34e',
    -port => '3306' );

my $db = new Bio::EnsEMBL::DBSQL::DBAdaptor(
    -species => "Homo_sapiens",
    -group => "estgene",
    -host => 'host1',
    -user => 'anonymous',
    -dbname => 'homo_sapiens_estgene_24_34e',
    -port => '3306' );

$reg->add_DNAAdaptor($db->species, $db->group, "core");

The script is ran by calling the method load_all and passing it the file name. Alternatively if there is no file name the Environment Variable ENSEMBL_REGISTRY is checked for a valid file. If that fails the file .ensembl_initrc is checked. So a central configuration script can be setup and occasional API programmers will no longer have to remember what databases are where and on what port etc. So to use the above configuration to get the sequence from a estgene stable_id would be :-
This presumes I have set up ENSEMBL_REGISTRY.

use Bio::EnsEMBL::Registry;
my $reg = "Bio::EnsEMBL::Registry";

$reg->load_all();

my $gadap = $reg->get_adaptor("human", "estgene", "Gene");

my $gene = $gadap->fetch_by_stable_id("ENSESTG0000015126");

print $gene->seq."\n";