OpenMS Tutorial

Version: 1.1

Contents

1 OpenMS Terms

1.1 Mass spectrometry terms

The following terms for MS-related data are used in this tutorial and the OpenMS class documentation:

- **raw data point**

  An unprocessed data point as measured by the instrument.

- **peak**

  Data point that is the result of some kind of peak detection algorithm. Peaks are often referred to as *sticks* or *centroided data* as well.

- **spectrum / scan**

  A mass spectrum containing raw data points (*raw spectrum*) or peaks (*peak spectrum*).

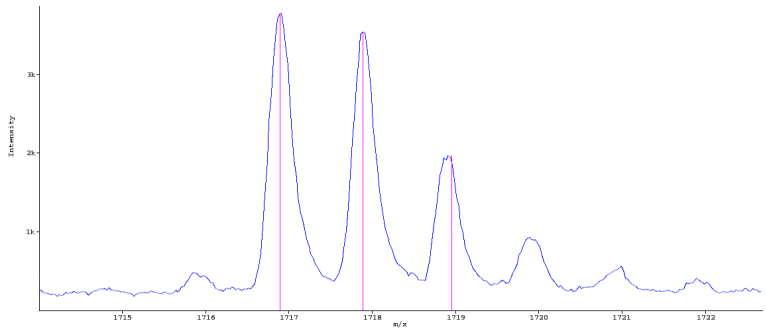

Figure 1: Part of a raw spectrum (blue) with three peaks (red)

- **map**

  A collection of spectra generated by a HPLC-MS experiment. Depending on what kinds of spectra are contained, we use the terms *raw map* or *peak map*. Often a map is also referred to as an *experiment*.

- **feature**

  The signal caused by a chemical entity detected in an HPLC-MS experiment, typically a peptide.

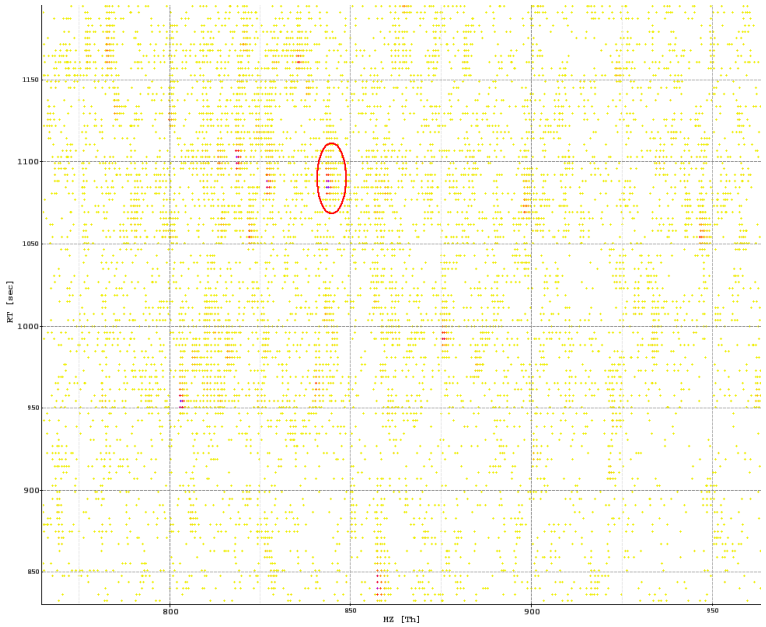

Figure 2: Peak map with a marked feature (red)

2 OpenMS Tutorial

This tutorial gives an introduction to the OpenMS core datastructures and algorithms. It is intended to allow for a quick start in writing your own applications based on the OpenMS framework.

The structure of this tutorial is similar to the modules of the class documentation. First, the basic concepts and datastructures of OpenMS are explained. The next chapter is about the kernel datastructures. These datastructures represent the actual mass spectronomy data: raw data, peaks, spectra and maps. In the following chapters, the more sophisticated datastructures and algorithms, e.g. those used for peak picking, feature finding and protein/peptide identification are presented.

All the example programs used in this tutorial, can be found in *OpenMS/source/EXAMPLES/*.

If you are looking for C++ literature, we recommend the following books:

- **C++:**
  C++ Primer
  Effective C++
- **STL:**
  Generic Programming and the STL
  Effective STL
- **Qt:**
  C++ GUI Programming with Qt 4

2.1 OpenMS concepts

This chapter covers some very basic concepts needed to understand OpenMS code. It describes OpenMS primitive types, namespaces, exceptions and important preprocessor macros. The classes described in this section can be found in the *CONCEPT* folder.

2.1.1 Basic data types

OpenMS has its own names for the C++ primitive types. The integer types of OpenMS are *Int* (int) and *UInt* (unsigned int). For floating point numbers, *Real* (float) and *DoubleReal* (double) are used.

These and more types are defined in *OpenMS/CONCEPT/Types.h*.

2.1.2 The OpenMS namespace

The main classes of OpenMS are implemented in the namespace *OpenMS*. There are several sub-namespaces to the *OpenMS* namespace. The most important ones are:

- *OpenMS::Constants* contains nature constants.
- *OpenMS::Math* contains math functions and classes.
- *OpenMS::Exception* contains the OpenMS exceptions.
- *OpenMS::Internal* contains certain auxiliary classes that are typically used by only one class of the *OpenMS* namespace and not by the user directly.

There are several more namespaces. For a detailed description have a look at the class documentation.

2.1.3 Exception handling in OpenMS

All exceptions are defined in the namespace *OpenMS::Exception*. The Base class for all OpenMS exceptions is *Base*. This base class provides three members for storing the source file, the line number and the function name where the exception occurred. All derived exceptions provide a constructor that takes at least these arguments. The following code snippet shows the handling of an index overflow:

```
void someMethod(UInt index) throw (Exception::IndexOverflow)
{
  if (index >= size())
  {
    throw Exception::IndexOverflow(__FILE__, __LINE__, __PRETTY_FUNCTION__, index, size()-1);
  }
  // do something
};
```

Note the first three arguments given to the constructor: *__FILE__* and *__LINE__* are built-in preprocessor macros that hold the file name and the line number. *__PRETTY_FUNCTION__* is replaced by the GNU g++ compiler with the demangled name of the current function (including the class name and argument types). For other compilers it is defined as "<unknown>". For an index overflow exception, there are two further arguments: the invalid index and the maximum allowed index.

The file name, line number and function name are very useful in debugging. However, OpenMS also implements its own exception handler which allows to turn each uncaught exception into a segmentation fault. This mechanism allows developers to trace the source of an exception with a debugger. To use this feature, set the environment variable *OPENMS_DUMP_CORE*.

2.1.4 Condition macros

In order to enforce algorithmic invariants, the two preprocessor macros *OPENMS_PRECONDITION* and *OPENMS_POSTCONDITION* are provided. These macros are enabled only if debug info is enabled and optimization is disabled in *configure*. Otherwise they are removed by the preprocessor, so they won't cost any performance.

The macros throw Exception::Precondition or Exception::Postcondition respectively if the condition fails. The example from section Exception handling in OpenMS could have been implemented like that:

```
void someMethod(UInt index)
{
  OPENMS_PRECONDITION(index < size(),"Precondition not met!");
  //do something
};
```

2.2 Auxiliary datastructures

This section contains a short introduction to three datastructures you will definitely need when programming with OpenMS. The datastructures module of the class documentation contains many more classes, which are not mentioned here in detail. The classes described in this section can be found in the *DATASTRUCTURES* folder.

2.2.1 The OpenMS string implementation

The OpenMS string implementation *String* is based on the STL *std::string*. In order to make the OpenMS string class more convenient, a lot of methods have been implemented in addition to the methods provided by the base class. A selection of the added functionaliy is given here:

- Checking for a substring (suffix, prefix, substring, char)
- Extracting a substring (suffix, prefix, substring)
- Trimming (left, right, both sides)
- Concatenation of string and other primitive types with *operator+*
- Construction from QString and conversion to QString

2.2.2 D-dimensional coordinates

Many OpenMS classes, especially the kernel classes, need to store some kind of d-dimensional coordinates. The template class *DPosition* is used for that purpose. The interface of DPosition is pretty straightforward. The operator[] is used to access the coordinate of the different dimensions. The dimensionality is stored in the enum value *DIMENSION*. The following example (Tutorial_DPosition.C) shows how to print a DPosition to the standard output stream.

First we need to include the header file for *DPosition* and *iostream*. Then we import all the OpenMS symbols to the scope with the *using* directive.

```
#include <OpenMS/DATASTRUCTURES/DPosition.h>
#include <iostream>

using namespace OpenMS;
```

The first commands in the main method initialize a 2-dimensional *DPosition* :

```
Int main()
{
  DPosition<2> pos;
  pos[0] = 8.15;
  pos[1] = 47.11;
```

Finally we print the content of the DPosition to the standard output stream:

```
  for (UInt i = 0; i < DPosition<2>::DIMENSION; ++i)
  {
    std::cout << "Dimension " << i << ": "<< pos[i] << std::endl;
  }

  return 0;
} //end of main
```

The output of our first little OpenMS program is the following:

```
Dimension 0: 8.15
Dimension 1: 47.11
```

2.2.3 D-dimensional ranges

Another important datastructure we need to look at in detail is *DRange*. It defines a d-dimensional, half-open interval through its two *DPosition* members. These members are accessed by the *min* and *max* methods and can be set by the *setMin* and *setMax* methods.

DRange maintains the invariant that *min* is geometrically less or equal to *max*, i.e. $\min()[x] \leq \max()[x]$ for each dimension *x*. The following example (Tutorial_DRange.C) demonstrates this behavior.

This time, we skip everything before the main method. In the main method, we create a range and assign values to *min* and *max*. Note that the the minimum value of the first dimension is larger than the maximum value.

```
Int main()
{
  DRange<2> range;
  range.setMin( DPosition<2>(2.0, 3.0) );
  range.setMax( DPosition<2>(1.0, 5.0) );
```

Then we print the content of *range* :

```
  for (UInt i = 0; i < DRange<2>::DIMENSION; ++i)
  {
    std::cout << "min " << i << ": "<< range.min()[i] << std::endl;
    std::cout << "max " << i << ": "<< range.max()[i] << std::endl;
  }

  return 0;
} //end of main
```

The output is:

```
min 0: 1
max 0: 1
min 1: 3
max 1: 5
```

As you can see, the minimum value of dimension one was adjusted in order to make the maximum of *1* conform with the invariant.

DIntervalBase is the closed interval counterpart (and base class) of *DRange*. Another class derived from *DIntervalBase* is *DBoundingBox*. It also represents a closed interval, but differs in the methods. Please have a look at the class documentation for details.

2.3 The kernel classes

The OpenMS kernel contains the datastructures that store the actual MS data, i.e. raw data points, peaks, features, spectra, maps. The classes described in this section can be found in the *KERNEL* folder.

2.3.1 Raw data point, Peak, Feature, ...

In general, there are three types of data points: raw data points, peaks and picked peaks. Raw data points provide members to store position (mass-to-charge ratio, retention time, ...) and intensity. Peaks are derived from raw data points and add an interface to store meta information. Picked peaks are derived from peaks and have additional members for peak shape information: charge, width, signal-to-noise ratio and many more.

The kernel data points exist in three versions: one-dimensional, two-dimensional and d-dimensional.

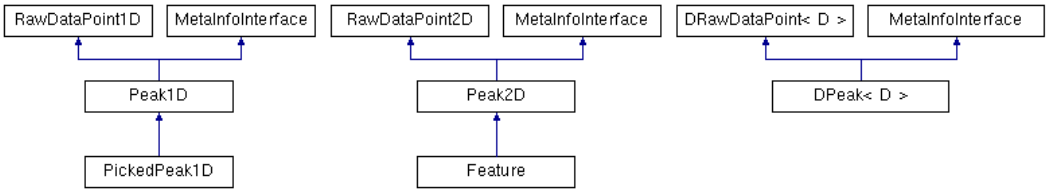

Figure 3: Data structures for MS data points

one-dimensional data points

The one-dimensional data points are most important, the two-dimensional and d-dimensional data points are needed rarely. The base class of the one-dimensional data points is *RawDataPoint1D*. It provides members to store the mass-to-charge ratio (*getMZ* and *setMZ*) and the intensity (*getIntensity* and *setIntensity*).
Peak1D is derived from *RawDataPoint1D* and adds an interface for metadata (see MetaInfo).
PickedPeak1D is derived from *Peak1D* and adds information about the peak shape.

two-dimensional data points

The two-dimensional data points are needed when geometry algorithms are applied to the data points. A special case is the *Feature* class, which needs a two-dimensional position (m/z and RT).
The base class of the two-dimensional data points is *RawDataPoint2D*. It provides the same interface as *RawDataPoint1D* and additional members for the retention time (*getRT* and *setRT*).
Peak2D is derived from *RawDataPoint2D* and adds an interface for metadata.
Feature is derived from *Peak2D* and adds information about the convex hull of the feature, fitting quality and so on.

d-dimensional data points

The d-dimensional data points are needed only in special cases, e.g. in template classes that must operate on any number of dimensions.
The base class of the d-dimensional data points is *DRawDataPoint*. The methods to access the position are *getPosition* and *setPosition*.
Note that the one-dimensional and two-dimensional data points also have the methods *getPosition* and *setPosition*. They are needed in order to be able to write algorithms that can operate on all data point types. It is, however, recommended not to use these members unless you really write such a generic algorithm.

2.3.2 Spectra

one-dimensional spectrum

The most important container for raw data and peaks is *MSSpectrum*. It is a template class that takes the peak type as template argument. The default peak type is *Peak1D*. Possible other peak types are classes derived from *RawDataPoint1D* or classes providing the same interface.
MSSpectrum is derived from two base classes: *DSpectrum*, a generic container for d-dimensional peak data, and from *SpectrumSettings*, a container for the meta data of a spectrum. Here, only MS data handling is explained, *SpectrumSettings* is described in section Meta data of a spectrum.

d-dimensional spectrum

The base class of *MSSpectrum* is *DSpectrum*. This class provides a generic container for d-dimensional data. One of the template arguments of *DSpectrum* is the Container the data is stored in. As container `DPeakArray` or a container with the same interface is used. *DPeakArray* is a vector of data points with a more convenient interface for sorting the data.
The peak container can be accessed through the *getContainer()* method. For convenience, part of the container interface is also provided by *DSpectrum*.

In the following example (Tutorial_MSSpectrum.C) program, a *MSSpectrum* is filled with peaks, sorted according to mass-to-charge ratio and a selection of peak positions is displayed.

First we create a spectrum and insert peaks with descending mass-to-charge ratios:

```
Int main()
{
  MSSpectrum<> spectrum;
  Peak1D peak;

  for (Real mz=1500.0; mz>=500; mz-=100.0)
  {
    peak.setMZ(mz);
    spectrum.push_back(peak);
  }
```

Then we sort the peaks according to ascending mass-to-charge ratio. As the method used for sorting is not wrapped by *DSpectrum*, we need to access the container to sort it.

```
  spectrum.getContainer().sortByPosition();
```

Finally we print the peak positions of those peaks between 800 and 1000 Thomson. For printing all the peaks in the spectrum, we simply would have used the STL-conform methods *begin()* and *end()*.

```
  MSSpectrum<>::Iterator it;
  for(it=spectrum.MZBegin(800.0); it!=spectrum.MZEnd(1000.0); ++it)
  {
    cout << it->getMZ() << endl;
  }

  return 0;
} //end of main
```

Typedefs

For convenience, the following type definitions are defined in *OpenMS/KERNEL/StandardTypes.h*.

```
typedef MSSpectrum<Peak1D> PeakSpectrum;
typedef MSSpectrum<RawDataPoint1D> RawSpectrum;
```

2.3.3 Maps

Although raw data maps, peak maps and feature maps are conceptually very similar. They are stored in different data types. For raw data and peak maps, the default container is *MSExperiment*, which is an array of *MSSpectrum* instances. Just as *MSSpectrum* it is a template class with the peak type as template parameter.

In contrast to raw data and peak maps, feature maps are no collection of one-dimensional spectra, but an array of two-dimensional *Feature* instances. The main datastructure for feature maps is called *FeatureMap*.

Although *MSExperiment* and *FeatureMap* differ in the data they store, they also have things in common. Both store meta data that is valid for the whole map, i.e. sample description and instrument description. This data is stored in the common base class *ExperimentalSettings*.

MSExperiment

The following figure shows the big picture of the kernel datastructures. *MSExperiment* is derived from *ExperimentalSettings* (meta data of the experiment) and from *vector<MSSpectrum>*. The one-dimensional spectrum *MSSpectrum* is derived from SpectrumSettings (meta data of a spectrum) and from *DSpectrum<1>*, which stores the actual peak data in a *DPeakArray*.
Since DPeakArray can store all types of peaks derived from *RawDataPoint*, all the data containers are template classes that take the peak type as template argument. This is omitted in the diagram for simplicity.

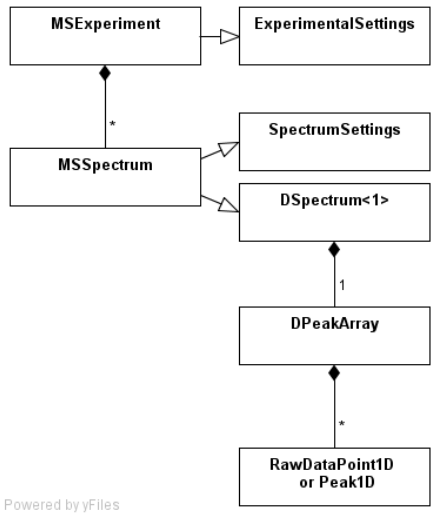

Figure 4: Overview of the main kernel datastructures

Typedefs

For convenience, the following map types are defined in *OpenMS/KERNEL/StandardTypes.h*.

```
typedef MSExperiment<Peak1D> PeakMap;
typedef MSExperiment<RawDataPoint1D> RawMap;
```

The following example program (Tutorial_MSExperiment.C) creates a *MSExperiment* containing four *MSSpectrum* instances. Then it iterates over an area and prints the peak positions in the area:

First we create the spectra in a for-loop and set the retention time and MS level. Survey scans have a MS level of 1, MS/MS scans would have a MS level of 2, and so on.

```
Int main()
{
  PeakMap exp;

  for (UInt i=0; i<4; ++i)
  {
    PeakSpectrum spectrum;
    spectrum.setRT(i);
    spectrum.setMSLevel(1);
```

Then we fill each spectrum with several peaks. As all spectra would have the same peaks otherwise, we add the retention time to the mass-to-charge ratio of each peak.

```
    for (Real mz=500.0; mz<=900; mz+=100.0)
    {
      Peak1D peak;
      peak.setMZ(mz+i);
      spectrum.push_back(peak);
    }
    exp.push_back(spectrum);
  } //end of creation
```

Finally, we iterate over the RT range (2,3) and the m/z range (603,802) and print the peak positions.

```
  for(PeakMap::AreaIterator it=exp.areaBegin(2.0, 3.0, 603.0, 802.0); it!=exp.areaEnd(); ++it)
  {
    cout << it.getRT() << " - " << it->getMZ() << endl;
  }
```

The output of this loop is:

```
  2 - 702
  2 - 802
  3 - 603
  3 - 703
```

For printing all the peaks in the experiment, we could have used the STL-iterators of the experiment to iterate over the spectra and the STL-iterators of the spectra to iterate over the peaks:

```
  for(PeakMap::Iterator s_it=exp.begin(); s_it!=exp.end(); ++s_it)
  {
    for (PeakSpectrum::Iterator p_it=s_it->begin(); p_it!=s_it->end(); ++p_it)
    {
      cout << s_it->getRT() << " - " << p_it->getMZ() << endl;
    }
  }

  return 0;
} //end of main
```

FeatureMap

FeatureMap, the container for features, is simply a *vector<Feature>*. Additionally, it is derived from *ExperimentalSettings*, to store the meta information. Just like *MSExperiment*, it is a template class. It takes the feature type as template argument.

The following example (Tutorial_FeatureMap.C) shows how to insert two features into a map and iterate over the features.

```
Int main()
{
  FeatureMap<> map;

  Feature feature;
  feature.setRT(15.0);
  feature.setMZ(571.3);
  map.push_back(feature); //append feature 1
  feature.setRT(23.3);
  feature.setMZ(1311.3);
  map.push_back(feature); //append feature 2


  for (FeatureMap<>::Iterator it=map.begin(); it!=map.end(); ++it)
  {
    cout << it->getRT() << " - " << it->getMZ() << endl;
  }

  return 0;
} //end of main
```

RangeManager

All peak and feature containers (*DSpectrum*, *MSExperiment*, *FeatureMap*) are also derived from *RangeManager*. This class facilitates the handling of MS data ranges. It allows to calculate and store both the position range and the intensity range of the container.

The following example (Tutorial_RangeManager.C) shows the functionality of the class *RangeManger* using a *FeatureMap*. First a *FeatureMap* with two features is created, then the ranges are calulated and printed:

```
Int main()
{
  FeatureMap<> map;

  Feature feature;
  feature.setIntensity(461.3);
  feature.setRT(15.0);
  feature.setMZ(571.3);
  map.push_back(feature);
  feature.setIntensity(12213.5);
  feature.setRT(23.3);
  feature.setMZ(1311.3);
  map.push_back(feature);

  //calculate the ranges
  map.updateRanges();

  cout << "Int: " << map.getMinInt() << " - " << map.getMaxInt() << endl;
  cout << "RT:  " << map.getMin()[0] << " - " << map.getMax()[0] << endl;
  cout << "m/z: " << map.getMin()[1] << " - " << map.getMax()[1] << endl;

  return 0;
} //end of main
```

The output of this program is:

```
  Int: 461.3 - 12213.5
  RT:  15 - 23.3
  m/z: 571.3 - 1311.3
```

2.4 How meta data is stored

The meta informations about an HPLC-MS experiment are stored in *ExperimentalSettings* and *SpectrumSettings*. All information that is not covered by these classes can be stored in the type-name-value datastructure *MetaInfo*. All classes described in this section can be found in the *METADATA* folder.

2.4.1 MetaInfo

DataValue is a data structure that can store any numerical or string information. It also supports casting of the stored value back to its original type.

MetaInfo is used to easily store information of any type, that does not fit into the the other classes. It implements type-name-value triplets. The main datastructure is an associative container that stores *DataValue* instances as values associated to string keys. Internally, the string keys are converted to integer keys for performance resaons i.e. a *map<UInt,DataValue>* is used.

The *MetaInfoRegistry* associates the string keys used in *MetaValue* with the integer values that are used for internal storage. The *MetaInfoRegistry* is a singleton.

If you want a class to have a *MetaInfo* member, simply derive it from *MetaInfoInterface*. This class provides a *MetaInfo* member and the interface to access it.

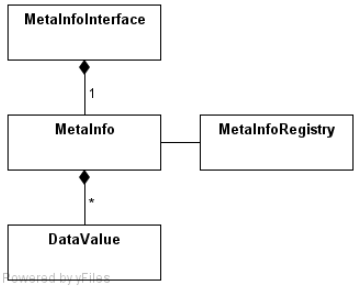

Figure 5: The classes involved in meta information storage

The following example (Tutorial_MetaInfo.C) shows how to use *Metadata*. We can simply set values for the string keys, and *setMetaValue* registers these names automatically. In order to access the values, we can either use the registered name or the index of the name. The *getMetaValue* method returns a *DataValue*, which has to be casted to the right type. If you do not know the type, you can use the *DataValue::valueType()* method.

```
Int main()
{
  MetaInfoInterface info;

  //insert meta data
  info.setMetaValue("color",String("#ff0000"));
  info.setMetaValue("id",112131415);

  //access id by index
  UInt id_index = info.metaRegistry().getIndex("id");
  cout << "id   : " << (UInt)(info.getMetaValue(id_index)) << endl;
  //access color by name
  cout << "color: " << (String)(info.getMetaValue("color")) << endl;

  return 0;
} //end of main
```

2.4.2 Meta data of a map

This class holds meta information about the experiment that is valid for the whole experiment:

- protein identifications
- preprocessing performed on the data
- MS instrument
- source file
- contact person
- sample description
- instrument software
- HPLC settings

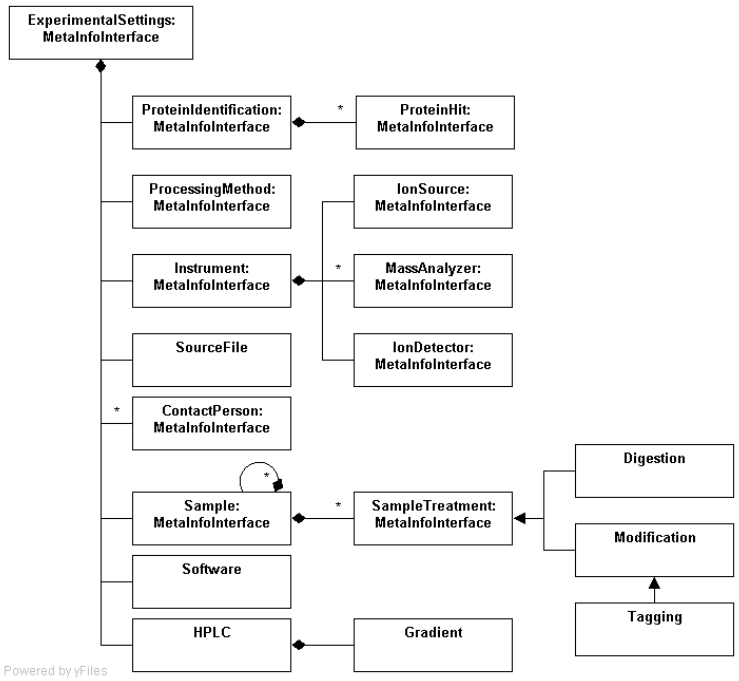

Figure 6: Map meta information

2.4.3 Meta data of a spectrum

This class contains meta information about settings specific to one spectrum:

- spectrum-specific instrument settings
- peptide and protein identifications
- precursor information (of MS/MS spectra)

- description of the *MetaInfo* present for each data point
- information on the acquisition

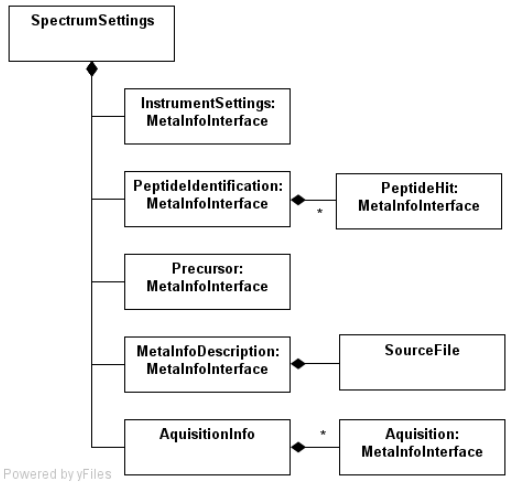

Figure 7: Spectrum meta information

2.5 File and DB access

All classes for file and database IO can be found in the *FORMAT* folder.

2.5.1 File adapter classes

The interface of most file adapter classes is very similar. They implement a *load* and a *store* method, that take a file name and the appropriate data structure.

The following example (Tutorial_FileIO.C) demonstrates the use of *MzDataFile* and *MzXMLFile* to convert one format into another using *MSExperiment* to hold the temporary data:

```
Int main()
{

  MzXMLFile mzxml;
  MzDataFile mzdata;

  // temporary data storage
  MSExperiment<RawDataPoint1D> map;

  // convert MzXML to MzData
  mzxml.load("Tutorial_FileIO.mzXML",map);
  mzdata.store("Tutorial_FileIO.mzData",map);

  return 0;
} //end of main
```

FileHandler

In order to make the handling of different file types easier, the class *FileHandler* can be used. It loads a file into the appropriate data structure independently of the file type. The file type is determined from the file extension or the file contents:

```
MSExperiment<> in;
FileHandler handler();
handler.loadExperiment("input.mzData",in);
```

2.5.2 DB access

For database access, the class *DBAdapter* is used. As its interface is very similar to the interface of the file adapters, no example is shown here.

2.5.3 PeakFileOptions

In order to have more control over loading data from files or databases, most adapters can be configured using *PeakFileOptions*. The following options are available:

- only a specific retention time range is loaded
- only a specific mass-to.charge ratio range is loaded
- only a specific intensity range is loaded
- only spectra with a given MS level are loaded
- only meta data of the whole experiment is loaded (*ExperimentalSettings*)

2.5.4 Param

Most algorithms of OpenMS and some of the TOPP tools have many parameters. The parameters are stored in instances of *Param*. This class is similar to a Windows INI files. The actual parameters (type, name and value) are stored in sections. Sections can contain parameters and sub-sections, which leads to a tree-like structure. The values are stored in *DataValue*, but *Param* supports only the types *string*, *int* and *float*.

Parameter names are given as a string including the sections and subsections in which ':' is used as a delimiter.

The following example (Tutorial_Param.C) shows how a file description is given.

```
Int main()
{
  Param param;

  param.setValue("file:name","test.xml");
  param.setValue("file:size(MB)",572.3);
  param.setValue("file:data:min_int",0);
  param.setValue("file:data:max_int",16459);

  cout << "Name   : " << (String)(param.getValue("file:name")) << endl;
  cout << "Size   : " << (Real)(param.getValue("file:size(MB)")) << endl;
  cout << "Min int: " << (UInt)(param.getValue("file:data:min_int")) << endl;
  cout << "Max int: " << (UInt)(param.getValue("file:data:max_int")) << endl;

  return 0;
} //end of main
```

2.6 Signal processing

OpenMS offers several filters for the reduction of noise and baseline which disturb LC-MS measurements. These filters work spectra-wise and can therefore be applied to a whole raw data map as well as to a single raw spectrum. All filters offer functions for the filtering of raw data containers (e.g. *RawSpectrum*) "filter" as well as functions for the processing of a collection of raw data containers (e.g. *RawMap*) "filterExperiment". The functions "filter" and "filterExperiment" can both be invoked with an input container along with an output container or with iterators that define a range on the input container along with an output container. The classes described in this section can be found in the *FILTERING* folder.

2.6.1 Baseline filters

Baseline reduction can be perfomed by the *TopHatFilter*. The top-hat filter is a morphological filter which uses the basic morphological operations "erosion" and "dilatation" to remove the baseline in raw data. Because both operations are implemented as described by Van Herk the top-hat filter expects equally spaced raw data points. If your data is not uniform yet, please use the *LinearResampler* to generate equally spaced data.

The *TopHatFilter* removes signal structures in the raw data which are broader than the size of the structuring element.

The following example (Tutorial_TopHatFilter.C) shows how to instantiate a tophat filter, set the length of the structuring element and remove the base line in a raw LC-MS map.

```
Int main()
{
  RawMap exp_raw;
  RawMap exp_filtered;

  MzDataFile mzdata_file;
  mzdata_file.load("../TEST/data/PeakPicker_test.mzData",exp_raw);

  TopHatFilter th;
  Param param;
  param.setValue("struc_elem_length",1.0);
  th.setParameters(param);

  th.filterExperiment(exp_raw,exp_filtered);

  return 0;
} //end of main
```

Note:

In order to remove the baseline, the width of the structuring element should be greater than the width of a peak.

2.6.2 Smoothing filters

We offer two smoothing filters to reduce noise in LC-MS measurements.

2.6.2.1 Gaussian filter The class *GaussFilter* is a gaussian filter. The wider the kernel width, the smoother the signal (the more detail information gets lost).

We show in the following example (Tutorial_GaussFilter.C) how to smooth a raw data map. The gaussian kernel width is set to 1 m/z.

```
Int main()
{
  RawMap exp_raw;
  RawMap exp_filtered;

  MzDataFile mzdata_file;
  mzdata_file.load("../TEST/data/PeakPicker_test.mzData",exp_raw);

  GaussFilter g;
  Param param;
  param.setValue("gaussian_width",1.0);
  g.setParameters(param);

  g.filterExperiment(exp_raw,exp_filtered);

  return 0;
} //end of main
```

Note:

Use a gaussian filter kernel which has approximately the same width as your mass peaks.

2.6.2.2 Savitzky Golay filter The Savitzky Golay filter is implemented in two ways *SavitzkyGolaySVDFilter* and *SavitzkyGolayQRFilter*. Both filters come to the same result but in most cases the *SavitzkyGolaySVDFilter* has a better run time. The Savitzky Golay filter works only on equally spaced data. If your data is not uniform use the *LinearResampler* to generate equally spaced data. The smoothing degree depends on two parameters: the frame size and the order of the polynomial used for smoothing. The frame size corresponds to the number of filter coefficients, so the width of the smoothing interval is given by frame_size∗spacing of the raw data. The bigger the frame size or the smaller the order, the smoother the signal (the more detail information gets lost!).

The following example (Tutorial_SavitzkyGolayFilter.C) shows how to use a *SavitzkyGolaySVDFilter* (the *SavitzkyGolayQRFilter* has the same interface) to smooth a single spectrum. The single raw data spectrum is loaded and resampled to uniform data with a spacing of 0.01 /m/z. The frame size of the Savitzky Golay filter is set to 21 data points and the polynomial order is set to 3. Afterwards the filter is applied to the resampled spectrum.

```
Int main()
{
  RawSpectrum spec_raw;
  RawSpectrum spec_resampled;
  RawSpectrum spec_filtered;

  DTAFile dta_file;
  dta_file.load("../TEST/data/PeakTypeEstimator_rawTOF.dta",spec_raw);

  LinearResampler lr;
  Param param_lr;
  param_lr.setValue("spacing",0.01);
  lr.setParameters(param_lr);
  lr.raster(spec_raw,spec_resampled);

  SavitzkyGolayFilter sg;
  Param param_sg;
  param_sg.setValue("frame_length",21);
  param_sg.setValue("polynomial_order",3);
  sg.setParameters(param_sg);
  sg.filter(spec_resampled,spec_filtered);

  return 0;
} //end of main
```

2.6.3 Calibration

OpenMS offers methods for external and internal calibration of raw or peak data.

2.6.3.1 Internal Calibration The InternalCalibration uses reference masses for calibration. At least two reference masses have to exist in each spectrum, otherwise it is not calibrated. The data to be calibrated can be raw data or already picked data. If we have raw data, a peak picking step is necessary. For the important peak picking parameters, have a look at the Peak picking section.

The following example (Tutorial_InternalCalibration.C) shows how to use the InternalCalibration for raw data. First the data and reference masses are loaded.

```
Int main()
{
  InternalCalibration ic;
  RawMap exp_raw;
  MzDataFile mzdata_file;
  mzdata_file.load("../TEST/data/InternalCalibration_test.mzData",exp_raw);

  std::vector<double> ref_masses;
  ref_masses.push_back(1296.68476942);
  ref_masses.push_back(2465.19833942);
```

Then we set the important peak picking parameters and run the internal calibration:

```
  Param param;
  param.setValue("PeakPicker:thresholds:peak_bound",800);
  param.setValue("PeakPicker:thresholds:fwhm_bound",0.1);
  param.setValue("PeakPicker:wavelet_transform:scale",0.12);
  ic.setParameters(param);

  ic.calibrate(exp_raw,ref_masses);

  return 0;
} //end of main
```

2.6.3.2 TOF Calibration The TOFCalibration uses calibrant spectra to convert a spectrum containing time-of-flight values into one with m/z values. For the calibrant spectra, the expected masses need to be known as well as the calibration constants in order to convert the calibrant spectra tof into m/z (determined by the instrument). Using the calibrant spectra's tof and m/z-values, first a quadratic curve fitting is done. The remaining error is estimated by a spline curve fitting. The quadratic function and the splines are used to determine the calibration equation for the conversion of the experimental data.

The following example (Tutorial_TOFCalibration.C) shows how to use the TOFCalibration for raw data. First the spectra and reference masses are loaded.

```
Int main()
{
  TOFCalibration ec;
  RawMap exp_raw,calib_exp;
  MzDataFile mzdata_file;
  mzdata_file.load("../TEST/data/TOFCalibration_test_calibrants.mzData",calib_exp);
  mzdata_file.load("../TEST/data/TOFCalibration_test.mzData",exp_raw);

  vector<DoubleReal> ref_masses;
  TextFile ref_file;
  ref_file.load("../TEST/data/TOFCalibration_test_calibrant_masses.txt",true);
  for(TextFile::Iterator iter = ref_file.begin(); iter != ref_file.end(); ++iter)
  {
    ref_masses.push_back(atof(iter->c_str()));
  }
```

Then we set the calibration constants for the calibrant spectra.

```
  std::vector<DoubleReal> ml1;
  ml1.push_back(418327.924993827);

  std::vector<DoubleReal> ml2;
  ml2.push_back(253.645187196031);

  std::vector<DoubleReal> ml3;
  ml3.push_back(-0.0414243465397252);

  ec.setML1s(ml1);
  ec.setML2s(ml2);
  ec.setML3s(ml3);
```

Finally, we set the important peak picking parameters and run the external calibration:

```
  Param param;
  param.setValue("PeakPicker:thresholds:peak_bound",800);
  param.setValue("PeakPicker:thresholds:fwhm_bound",0.1);
  param.setValue("PeakPicker:wavelet_transform:scale",0.12);
  ec.setParameters(param);
  ec.calibrate(calib_exp,exp_raw,ref_masses);

  return 0;
} //end of main
```

2.7 Data reduction

Data reduction in LC-MS analysis mostly consists of two steps. In the first step, called "peak picking", important information of the mass spectrometric peaks (e.g. peaks' mass centroid positions, their areas under curve and full-width-at-half-maxima) are extracted from the raw LC-MS data. The second data reduction step, called "feature finding", represents the quantification of all peptides in a proteomic sample. Therefore, the signals in a LC-MS map caused by all charge and isotopic variants of the peptide are detected and summarized resulting in a list of compounds or features, each characterized by mass, retention time and abundance. The classes described in this section can be found in the *TRANSFORMATIONS* folder.

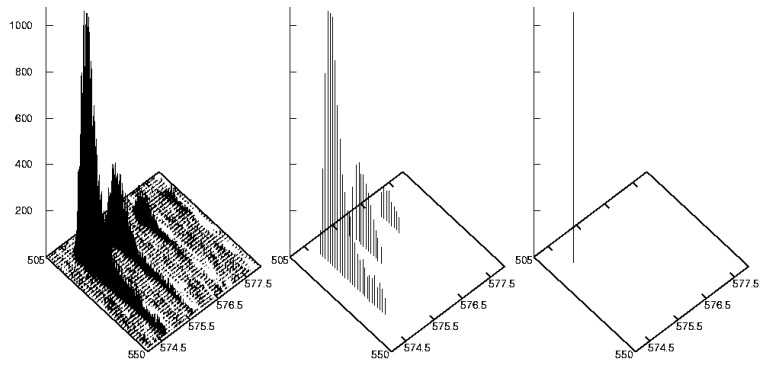

Figure 8: Part of an LC-MS map at different stages of data reduction. Axes depict retention time, m/z, and intensity. From left to right raw data points, peak picked data points and a feature are shown.

2.7.1 Peak picking

For peak picking, the class *PeakPickerCWT* is used. Because this class detects and extracts mass spectrometric peaks it is applicable to LC-MS as well as MALDI raw data.

The following example (Tutorial_PeakPickerCWT.C) shows how to open a raw map (in mzData format), initialize a PeakPickerCWT object, set the most important parameters (the scale of the wavelet, a peak's minimal height and fwhm), and start the peak picking process.

```
Int main()
{
  RawMap exp_raw;
  PeakMap exp_picked;

  MzDataFile mzdata_file;
  mzdata_file.load("../TEST/data/PeakPicker_test.mzData",exp_raw);

  PeakPickerCWT pp;
  Param param;
  param.setValue("thresholds:peak_bound",500.0);
  param.setValue("thresholds:fwhm_bound",0.1);
  param.setValue("wavelet_transform:scale",0.2);
  pp.setParameters(param);

  pp.pickExperiment(exp_raw,exp_picked);
  exp_picked.updateRanges();

  cout << "Scale of the wavelet: " << (DoubleReal)param.getValue("wavelet_transform:scale")
       << "\nMinimal fwhm of a mass spectrometric peak: " << (DoubleReal)param.getValue("thresholds:fwhm_b
       << "\nMinimal intensity of a mass spectrometric peak " << (DoubleReal)param.getValue("thresholds:pe
       << "\n\nNumber of picked peaks " << exp_picked.getSize() << std::endl;
```

```
  return 0;
} //end of main
```

The output of the program is:

```
    Scale of the wavelet: 0.2
    Minimal fwhm of a mass spectrometric peak: 0.1
    Minimal intensity of a mass spectrometric peak 500

    Number of picked peaks 14
```

Note:

A rough standard value for the peak's scale is the average fwhm of a mass spectrometric peak.

2.7.2 Peptide Quantification

The FeatureFinder implements algorithms for the detection and quantification of peptides from LC-MS maps. In contrast to the previous step (peak picking), we do not only search for pronounced signals (peak) in the LC-MS map but search explicitly for peptides which can be recognized by their isotopic pattern.

OpenMS offers different algorithms for this task.

Writing a FeatureFinder application of your own is straightforward to do. A short example (Tutorial_-FeatureFinder.C) is given below. First we need to instantiate the FeatureFinder, its parameters and the input/output data:

```
  FeatureFinder ff;
  // ... set parameters (e.g. from INI file)
  Param parameters;
  // ... set input data (e.g. from mzData file)
  MSExperiment<> input;
  // ... set output data structure
```

Then we run the FeatureFinder. The first argument is the algorithm name (here 'simple'). Using the second and third parameter, the peak and feature data is handed to the algorithm. The fourth argument sets the parameters used by the algorithm.

```
  FeatureMap<> output;

  ff.run("simple", input, output, parameters);
```

Now the FeatureMap is filled with the found features.

2.8 High-level data analysis

OpenMS offers a number of classes for high-level data analysis. This covers map alignment, peptide/protein identification, clustering, etc. The classes described in this section can be found in the *ANALYSIS* folder.

2.8.1 Map alignment

2.8.1.1 Pairwise map alignment The class *PoseClusteringPairwiseMapMatcher* can be used to map the element of one LC-MS map onto the elements of another LC-MS map. Corresponding elements of the two maps are moved closer together and the retention time as well as the m/z dimensions of the two maps become comparable. The transformation which maps one map onto the other is computed during the so called **superposition phase**.

Superposition phase: In the first step of the superpositon phase, an approximation of the transformation is estimated which is used to determine reliable landmarks in the two maps. These landmarks enable in the second step the evaluation of a more precise transformation. For both steps, different classes can be chosen. Using the *PoseClusteringSuperimposerAffine*, an affine transformation can be estimated and the *PoseClusteringSuperimposerShift* estimates a transformation consisting of only a translation in rt and m/z. Given a first approximation of the underlying warp, the *SimplePairFinder* or the *DelaunayPairFinder* can be used to determine landmarks in the two maps which represent potential corresponding elements. These landmarks are used by the *MapMatcherRegression* to improve the inital transformation. The following example (Tutorial_PairwiseAlignment.C) shows how to use the *PoseClusteringPairwiseMapMatcher* and the *MapMatcherRegression* for the pairwise alignment of two maps. The *PoseClusteringPairwiseMapMatcher* is based on the *PoseClusteringSuperimposerAffine*, and a *DelaunayPairFinder*. The initial as well as the final transformation is stored in gridXML format.

We load two feature maps and instantiate a *PoseClusteringPairwiseMapMatcher* object. The *PoseClusteringPairwiseMapMatcher* gets references to both maps and a *Param* object which defines the type of transformation and pairfinder. Additionally, the value of `mz_bucket_size` is set, which represents the maximum deviation in m/z of two corresponding elements.

```
Int main()
{
  FeatureMap<> exp_feature_1;
  FeatureMap<> exp_feature_2;

  FeatureXMLFile featurexml_file;
  featurexml_file.load("../TEST/TOPP/MapAlignmentFeatureMap1.xml",exp_feature_1);
  featurexml_file.load("../TEST/TOPP/MapAlignmentFeatureMap2.xml",exp_feature_2);

  Param param;
  param.setValue("superimposer:type","poseclustering_affine");
  param.setValue("superimposer:tuple_search:mz_bucket_size",0.3);
  param.setValue("pairfinder:type","DelaunayPairFinder");

  std::vector < ElementPair<Feature> > landmarks;
  PoseClusteringPairwiseMapMatcher< FeatureMap<> > pcpm;
  pcpm.setParameters(param);
  pcpm.setElementMap(0,exp_feature_1);
  pcpm.setElementMap(1,exp_feature_2);
  pcpm.run();
```

The *PoseClusteringPairwiseMapMatcher* determines a vector of element pairs and an initial estimate of the transformation. We store this initial transformation under "FirstAffineTransformation.gridXML" and pass the element pairs along with the initial transformation to a *MapMatcher* object.

```
  GridFile grid_file;
```

```
  grid_file.store("FirstAffineTransformation.gridXML",pcpm.getGrid());

  MapMatcherRegression< Feature > lr;
  lr.setElementPairs(pcpm.getElementPairs());
  lr.setGrid(pcpm.getGrid());
```

Using these landmarks, an improved transformation is estimated and stored under "SecondAffineTransformation.gridXML"

```
  lr.estimateTransform();
  grid_file.store("SecondAffineTransformation.gridXML",lr.getGrid());

  return 0;
} //end of main
```

Note:

The class *MapDewarper* can be used to apply the transformation to the elements of a map.

2.8.1.2 Multiple map alignment The *StarAlignment* class performs a star-like progressive multiple LC-MS map alignment based upon pairwise alignments as described above. Depending on the processing state of the input maps, the output of a multiple alignment varies. Peak maps are iteratively mapped onto one reference map and the result of the multiple peak map alignment are the dewarped maps themselves. In case of multiple feature maps, corresponding elements in all maps are determined during the so called **consensus phase** and are combined to a *ConsensusMap* using *DelaunayPairFinder*. The *StarAlignment* is able to compute the alignment of multiple peak, feature, or consensus maps. It provides the warps of all maps relative to the reference map as a result for the alignment of peak maps and computes a *ConsensusMap* as a result of a multiple feature, or consensus map alignment. The transformations can be stored in gridXML format using the *GridFile* as mentioned in the example above and can be applied to the maps using the *MapDewarper*. The *ConsensusMap* can be stored in ConsensusXML using a *ConsensusXMLFile*.

2.8.2 Identification

2.9 Chemistry

Especially for peptide/protein identification, a lot of data and data structures for chemical entities are needed. OpenMS offers classes for elements, formulas, peptides, etc. The classes described in this section can be found in the *CHEMISTRY* folder.

2.9.1 Elements

There is a representation of Elements implemented in OpenMS. The correcsponding class is named *Element*. This class stores the relevant information about an element. The handling of the Elements is done by the class ElementDB, which is implemented as a singleton. This means there is only one instance of the class in OpenMS. This is straightforward because the Elements do not change during execution. Data stored in an Element spans its name, symbol, atomic weight, and isotope distribution beside others.

```
const ElementDB* db = ElementDB::getInstance();

Element carbon = *db->getElement("Carbon"); // .getResidue("C") would also be ok

cout  << carbon.getName() << " "
      << carbon.getSymbol() << " "
      << carbon.getMonoWeight() << " "
      << carbon.getAverageWeight() << endl;
```

Elements can be accessed by the *ElementDB* class. As it is implemented as a singleton, only a pointer of the singleton can be used, via *getInstance()*. The example program writes the following output to the console.

```
  Carbon C 12 12.0107
```

2.9.2 EmpiricalFormula

The Elements described in the section above can be combined to empirical formulas. Application are the exact weights of molecules, like peptides and their isotopic distributions. The class supports a large number of operations like addition and subtraction. A simple example is given in the next few lines of code.

```
EmpiricalFormula methanol("CH3OH"), water("H2O");

EmpiricalFormula sum = methanol + water;

cout  << sum << " "
      << sum.getNumberOf("Carbon") << " "
      << sum.getAverageWeight() << endl;
```

Two instances of empirical formula are created. They are summed up, and some information about the new formula is printed to the terminal. The next lines show how to create and handle a isotopic distribution of a given formula.

```
IsotopeDistribution iso_dist = sum.getIsotopeDistribution(3);

for (IsotopeDistribution::ConstIterator it = iso_dist.begin(); it != iso_dist.end(); ++it)
{
  cout << it->first << " " << it->second << endl;
}
```

The isotopic distribution can be simply accessed by the *getIsotopeDistribution()* function. The parameter of this function describes how many isotopes should be reported. In our case, 3 are enough, as the following

numbers get very small. On larger molecules, or when one want to have the exact distribution, this number can be set much higher. The output of the code snipped might look like this.

```
O2CH6 1 50.0571
50 0.98387
51 0.0120698
52 0.00406
```

2.9.3 Residue

A residue is represented in OpenMS by the class *Residue*. It provides a container for the amino acids as well as some functionality. The class is able to provide information such as the isotope distribution of the residue, the average and monoisotopic weight. The residues can be identified by their full name, their three letter abbreviation or the single letter abreviation. The residue can also be modified, which is implemented in the Modification class. Additional less frequently used parameters of a residue like the gas-phase basicity and pk values are also available.

```
ResidueDB res_db;

Residue lys = *res_db.getResidue("Lysine"); // .getResidue("K") would also be ok

cout  << lys.getName() << " "
      << lys.getThreeLetterCode() << " "
      << lys.getOneLetterCode() << " "
      << lys.getAverageWeight() << endl;
```

This small example show how to create a instance of ResidueDB were all Residues are stored in. The amino acids themselves can be accessed via the getResidue function. ResidueDB reads its amino acid and modification data from data/CHEMISTRY/.

The output of the example would look like this

```
Lysine LYS K 146.188
```

2.9.4 AASequence

This class handles the amino acid sequences in OpenMS. A string of amino acid residues can be turned into a instance of *AASequence* to provide some commonly used operations and data. The implementation supports mathematical operations like addition or subtraction. Also, average and mono isotopic weight and isotope distributions are accessible.

Weights, formulas and isotope distribution can be calculated depending on the charge state (additional proton count in case of positive ions) and ion type. Therefore, the class allows for a flexible handling of amino acid strings.

A very simple example of handling amino acid sequence with AASequence is given in the next few lines.

```
AASequence seq("DFPIANGER");

AASequence prefix(seq.getPrefix(4));
AASequence suffix(seq.getSuffix(5));

cout  << seq << " "
      << prefix << " "
      << suffix << " "
      << seq.getAverageWeight() << endl;
```

Not only the prefix, suffix and subsequence accession is supported, but also most of the features of EmpiricalFormulas and Residues given above. Additionally, a number of predicates like hasSuffix are supported. The output of the code snippet looks like this.

```
DFPIANGER DFPI ANGER 1018.08
```

2.9.5 TheoreticalSpectrumGenerator

This class implements a simple generator which generates tandem MS spectra from a given peptide charge combination. There are various options which influence the occurring ions and their intensities.

```
TheoreticalSpectrumGenerator tsg;
PeakSpectrum spec1, spec2;
AASequence peptide("DFPIANGER");

tsg.addPeaks(spec1, peptide, Residue::YIon, 1);

tsg.getSpectrum(spec2, peptide, 2);

cout << "Spectrum 1 has " << spec1.size() << " peaks. " << endl;
cout << "Spectrum 2 has " << spec2.size() << " peaks. " << endl;
```

The example shows how to put peaks of a certain type, y-ions in this case, into a spectrum. Spectrum 2 is filled with a complete spectrum of all peaks (a-, b-, y-ions and losses). The TheoreticalSpectrumGenerator has many parameters which have a detailed description located in the class documentation. The output of the program looks like the following two lines.

```
Spectrum 1 has 8 peaks.
Spectrum 2 has 32 peaks.
```

2.10 Visualization

Visualization in OpenMS is based on Qt.

2.10.1 1D view

All types of peak or feature visualization share a common interface. So here only an example how to visualize a single spectrum is given (Tutorial_Spectrum1D.C).

First we need to create a *QApplication* in order to be able to use Qt widgets in out application.

```
Int main(int argc, const char** argv)
{
  QApplication app(argc,const_cast<char**>(argv));
```

Then we load a DTA file (the first command line argument of our application).

```
  MSExperiment<> exp;
  exp.resize(1);
  DTAFile().load(argv[1],exp[0]);
```

Then we create a widget for 1D visualization and hand over the data.

```
  Spectrum1DWidget* widget = new Spectrum1DWidget(Param(),0);
  widget->canvas()->addLayer(exp);
  widget->show();
```

Finally we start the application.

```
  return app.exec();
} //end of main
```

2.10.2 Visual editing of parameters

Param objects are used to set algorithm parameters in OpenMS. In order to be able to visually edit them, the *ParamEditor* class can be used. The following example (Tutorial_ParamEditor.C) show how to use it.

We need to create a QApplication, load the data from a file (e.g. the parameters file of any TOPP tool), create the *ParamEditor* and execute the application:

```
Int main(int argc, const char** argv)
{
  QApplication app(argc,const_cast<char**>(argv));

  Param param;
  param.load(argv[1]);

  ParamEditor* editor = new ParamEditor(0);
  editor->load(param);
  editor->show();

  app.exec();
```

When it is closed, we store the result back to the *Param* object and then to the file.

```
  editor->store();
  param.store(argv[1]);

  return 0;
} //end of main
```

2.11 HowTo

2.11.1 Creating a new algorithm

Most of the algorithms in OpenMS share the following base classes:

- *ProgressLogger* is used to report the progress of the algorithm.
- *DefaultParamHandler* is used to make the handling of parameters (and their defaults) easy.
  In most cases, you will not even need accessors for single parameters.

The interfaces of an algorithm depend on the datastructures it works on. For an algorithm that works on peak data, a non-template class should be used that provides template methods operating on *MSExperiment* or *MSSpectrum*, no matter which peak type is used. See *PeakPickerCWT* for an example.

For algorithms that do not work on peak data, templates should be avoided.