MzJava: An open source library for mass spectrometry data processing

Horlacher, Oliver; Nikitin, Frédéric; Alocci, Davide; Mariethoz, Julien; Muller, Markus Johann; Lisacek, Frédérique

How to cite

HORLACHER, Oliver et al. MzJava: An open source library for mass spectrometry data processing. In: Journal of proteomics, 2015, vol. 129, p. 63–70. doi: 10.1016/j.jprot.2015.06.013

This publication URL: https://archive-ouverte.unige.ch/unige:159524
Publication DOI: 10.1016/j.jprot.2015.06.013

MzJava: an open source library for mass spectrometry data processing

Oliver Horlacher[1,2], Frederic Nikitin[1], Davide Alocci[1,2], Julien Mariethoz[1], Markus Müller[1,2] and Frederique Lisacek[1,2]

[1]Proteome Informatics Group, SIB Swiss Institute of Bioinformatics, Geneva, 1211, Switzerland

[2]Centre Universitaire de Bioinformatique, University of Geneva, Geneva, 1211, Switzerland

Corresponding authors:

Markus Müller, Frederique Lisacek

Swiss Institute of Bioinformatics, University of Geneva

CMU

Rue Michel-Servet 1

1211 Geneva, Switzerland

Email: markus.mueller@isb-sib.ch, Frederique.Lisacek@isb-sib.ch

Abstract

Mass Spectrometry (MS) is a widely used and evolving technique for the high-throughput identification of molecules in biological samples. The need for sharing and reuse of code among bioinformaticians working with MS data prompted the design and implementation of MzJava, an open-source Java application programming interface (API) for MS related data processing. MzJava provides data structures and algorithms for representing and processing mass spectra and their associated biological molecules, such as metabolites, glycans and peptides. MzJava includes functionality to perform mass calculation, peak processing (e.g. centroiding, filtering, transforming), spectrum alignment and clustering, protein digestion, fragmentation of peptides and glycans as well as scoring functions for spectrum-spectrum and peptide/glycan-spectrum matches. For data import and export MzJava implements readers and writers for commonly used data formats. For many classes support for the Hadoop MapReduce (hadoop.apache.org) and Apache Spark (spark.apache.org) frameworks for cluster computing was implemented. The library has been developed applying best practises of software engineering. To ensure that MzJava contains code that is correct and easy to use the library's API was carefully designed and thoroughly tested. MzJava is an open-source project distributed under the AGPL v3.0 license. MzJava requires Java 1.7 or higher. Binaries, source code and documentation can be downloaded from http://mzjava.expasy.org and https://bitbucket.org/sib-pig/mzjava.

Introduction

Mass spectrometry (MS) has become a central analytical technique to characterise proteins, lipids, carbohydrates and metabolites in complex samples [1] [2]. The diversity of biological questions possibly addressed with MS is reflected in a wide range of experimental workflows. Analysing data generated through these workflows is automated though most of the time, the variability of applications requires software customisation and/or extension. This situation is well described in a recent review [3] and justifies the development of libraries of MS related software to facilitate code reuse. Software libraries are meant to benefit the developers' own group and collaborators as well as the wider computational proteomics and glycomics communities.

Many early open source contributions dedicated to the automated analysis of proteomic data were coded in C++ or Perl. The trans-proteomic pipeline (TPP) [4] is an assembly of C++ programs and Perl scripts to process and statistically validate MS/MS search results from different search engines and integrate these with quantitative data. Later, TOPP was introduced as a management system for generic proteomics workflows [5], based on OpenMS [6]. OpenMS contains a well-designed Application Programming Interface (API), which makes it useful not only as a toolbox but also as a code base for software developers. ProteoWizard [7] is another C++ open source project for the conversion of proteomic MS/MS file formats and processing of MS/MS spectra. More recently, the Java programming language gained popularity in many fields and especially in bioinformatics due to its portability across different computer platforms and the availability of powerful and comprehensive class libraries that facilitate and accelerate software development. For example, the Chemistry Development Kit (CDK, http://sourceforge.net/projects/cdk/) is an open source library for cheminformatics and computational chemistry [8]. Biojava (http://biojava.org/, [9]) addresses the bioinformatics community and provides classes and tools for protein structure comparison, alignments of DNA and protein sequences, analysis of amino acid properties, protein modifications and prediction of disordered regions in proteins as well as parsers for common file formats.

A broad range of Java based open source solutions was developed for the proteomics community as comprehensively reviewed in [3]. Our focus being mainly on MS/MS data processing, the following refers to such dedicated toolboxes. Compomics is a collection of tools mainly for MS/MS data analysis [10][11]. It also contains the Compomics-utilities class library [12], which provides the code base tools. The Java Proteomics Library (JPL, javaprotlib.sourceforge.net) provides an API to process MS/MS spectra and their annotation. It served as the code base for several software projects dealing with MS/MS searches [13], spectral libraries and open modification searches [14], and data-independent quantification [15]. The PRIDE tool suite (http://pride-toolsuite.googlecode.com) contains a set of pure Java libraries, tools and packages designed to process and visualize versatile MS proteomics data [3]. It also contains a well-designed Java API (ms-data-core-api) facilitating the implementation of customized solutions [16]. In recent years, several open source Java-based laboratory information systems (LIMS) storing and processing proteomic or glycomic data were described [17],[18],[19].

Glycomics MS data are still seldom produced in high throughput set-ups but this field evolves quickly and the need for automation is growing. A reference software for glycan MS and MS/MS annotation is GlycoWorkbench [20]. This tool is modular and mostly known for its convenient user interface to support glycan structural assignment. Theoretical glycan spectra can be calculated with its fragmentation tool based on the mechanisms and nomenclature described by Domon and Costello [21]. Importantly it relies on recognised standard description of monosaccharides and full structures [22].

The scalability of existing solutions is currently one of the greatest challenges. The ever-growing size of MS datasets and the need to process spectra by the tens of millions imposes the use of distributed data processing frameworks such as Hadoop MapReduce [23] (hadoop.apache.org) and Apache Spark [24] (spark.apache.org). Hadoop is already used in bioinformatics, primarily for next-generation sequencing analysis [25] but also for proteomics [26][27][28], while Spark was more recently introduced [29][30]. Hadoop MapReduce is an implementation of the MapReduce

programming model described by Dean and Ghemawat [23]. Spark extends on the functionality and performance of Hadoop by allowing in-memory data storage and providing additional functions [24].

We introduce MzJava a Java class library designed to ease the development of custom data analysis software by providing building blocks that are common to most MS data processing software. MzJava addresses the scaling issues by adding classes to interface with Hadoop and Spark. Furthermore, new code was included for processing glycomics data. In fact, MzJava originates from merging JPL and another unpublished Java MS codebase. During this merge the code was comprehensively refactored and refined in an effort to produce a consistent and well-designed API. Best practices in software engineering such as test driven development and continuous integration were applied during the implementation. Code quality metrics of MzJava are continuously tracked to maintain high quality standards. These metrics are used to benchmark MzJava in relation to other packages.

Materials and methods

Development aims

MzJava is mostly centred on MS/MS identification and annotation. This bias towards an identification-related API reflects our research focus. The use of the MzJava API is intended for writing software that is capable of processing large data sets. Consequently the API is designed to be extensible, flexible and efficient. During development we found that flexibility often comes at the cost of performance. Where we identified performance hot-spots we implemented solutions that allow either flexible or high performance code to be written, while in non performance critical code we kept flexibility as a major criterion. Additional design aims were to make the API not only easy to use, but also hard to misuse as well as prompt to fail whenever there are errors [31].

The development of MzJava entailed refactoring a substantial part of the JPL aiming at producing high quality and efficient code. The outcome is meant to be structured as a coherent API

as opposed to bundling a collection of code pieces. MzJava follows Java naming and behaviour conventions and provides builders with fluent interfaces for constructing complex objects (http://en.wikipedia.org/wiki/Fluent_interface). To help prevent misuse and to make MzJava easy to use in multithreaded environments, mutable objects are avoided as much as possible. Objects that need to be mutable were designed to always be in a valid state.

Development methodology

The methodology that was employed to develop MzJava follows best practice for scientific computing [32][33][34][35] and is influenced by agile software development, especially test driven development (TDD) [36] and continuous integration (CI) [37]. In brief, TDD describes a short development cycle for adding improvements or new functionality. First the developer writes an, initially failing, automated test that defines the improvement or new functionality. Code is then written to make the test pass. This is followed by refactoring the new code to bring it up to acceptable quality. Figure 1A summarizes the TDD cycle. Automated tests are written using JUnit (http://junit.org/) and Mockito (http://mockito.org/). CI is a software engineering practice in which changes to the code are automatically tested whenever they are added to the codebase. The goal of CI is to provide rapid feedback so that changes that introduce issues or break the build can be corrected rapidly. Jenkins (http://jenkins-ci.org/) is used to automate the CI.

Code quality scores are tracked using SonarQube (http://www.sonarqube.org/). Quality profiles were slightly altered from the Sonar way with Findbugs profiles. SonarQube is also used to evaluate the quality of comparable libraries such as BioJava, ms-data-core-api, jmzml (a library to handle mzML files [38]), and compomics-utilities. MzJava is a Maven project developed using IntelliJ Idea (https://www.jetbrains.com/idea/). Figure 1B provides a more detailed view on the development cycle and the tools used.

Architecture

The MzJava architecture is modular and consists of three main modules:

1. The *core module* contains functionality that is common to all MS data
2. The *proteomics module* contains functionality specific to peptides and proteins
3. The *glycomics module* contains functionality specific to glycans.

Figure 2 illustrates the organisation of the modules and highlights the central position of the core module that overlaps with the proteomics and glycomics modules.

Results

The MzJava core consists of three main parts: *mol*, *ms* and *io* (Figure 2). The *mol*-core comprises classes to work with chemical compositions and their masses. The *Atom* class for example represents the mass and isotopic abundances of a chemical element. The *Composition* class deals with assemblies of atoms defined by their stoichiometric chemical formulae. *NumericMass* is used to represent objects that have a mass but no known composition. The main classes in *ms*-core deal with peak lists (*PeakList*, list of *m/z*-intensity pairs) and their associated meta data (*Spectrum*). There are a number of *Spectrum* subclasses to capture the meta data associated with particular types of spectra. For example, *MsnSpectrum* contains meta data such as scan number and retention time and *ConsensusSpectrum* captures meta data that is associated with a consensus spectrum such as the ids of the spectra from which the consensus was built, and the structure of the peptide/glycan that the consensus spectrum represents. The peak list associated with each spectrum is stored in subclasses of *PeakList*. There are 5 different flavors of *PeakList*. This allows the m/z values to be stored as either 64 or 32 bit numbers and the intensity as 64 bit, 32 bit or as a constant. To provide a common interface for all the subclasses, the *Spectrum* class both wraps and implements *PeakList*. The only time that the precision needs to be taken into account is when the spectrum is created; thereafter it is just an instance of *Spectrum*. In addition to the *m/z* and intensity values of peaks, *PeakList* can also store one or more annotations for each peak. MzJava has three built-in peak annotations, *PepFragAnnotation*, *GlycanFragAnnotation* and *LibPeakAnnotation*.

PepFragAnnotation and *GlycanFragAnnotation* are used to annotate peaks with peptide or glycan fragments and contain information such as the charge and ion type of the fragment. *LibPeakAnnotation* is used to annotate peaks in *LibrarySpectrum* with consensus statistics. We make use of generics to specify the type of annotation that a *PeakList* contains.

The *ms-core* package also contains classes to match spectra and score these matches. The first step in most spectra matching workflows is to remove noise or to extract features from mass spectra by applying one or more filters or transformers to the peaks in the peak list. MzJava provides filters to bin and centroid raw spectra (*BinnedSpectrumFilter, CentroidFilter*) and to select peaks based on their *m/z* range or peak annotation (*MzRangeFilter, FragmentAnnotationFilter*). Further it includes processors that simplify spectra and retain only the n most intense peaks of the entire spectrum or remove peaks that fall below a given threshold (*ThresholdFilter*). In order to adapt to varying noise levels other filters only keep a peak if it is among the n most intense peaks in a local bin or sliding window (*NPeaksFilter, NPeaksPerBinFilter, NPeaksPerSlidingWindowFilter*). With the exception of the *m/z* affine transformer all transformers operate on peak intensities and replace the original values by their log-, square root-, affine- and arcsinh-transformed values (*LogTransformer, Log10Transformer, AffineTransformer, ArcsinhTransformer*). MzJava also has classes to normalize the total or maximal ion current in a spectrum (*IonCurrentNormalizer, UnitVectorNormalizer, NthPeakNormalizer, RankNormalizer, HighestPeakPerBinNormalizer*).

Peak processing is a performance hotspot, since it may be applied to millions of spectra, but also needs to be flexible so that it is possible to experiment with different filters and transformers. To achieve both of these requirements, peaks can be processed by applying a *PeakProcessorChain* to a *PeakList*. A *PeakProcesorChain* is a linked list of objects that implement the *PeakProcessor* interface. A *PeakList* is processed by pushing the *m/z*, intensity and annotations of each peak through the chain of *PeakProcessor*. This way, stateless peak processors such as *Log10Transformer* do not have to copy the *m/z* or intensity arrays. Peak processors that have state, i.e. where the result for each peak depends on the other peaks present in the peak list, can extend

DelayedPeakProcessor, which reuses *m/z* and intensity arrays to improve performance. Custom peak processors can easily be created by extending *AbstractPeakProcessor* or *DelayedPeakProcessor*. For a comprehensive list of peak processors see Table 1.

MzJava has a number of spectrum-spectrum scoring functions such as the normalized dot product (*NdpSimFunc*), Pearson's correlation coefficient (*PearsonsSimFunc*) and shared peak count (*SharedPeakSimFunc*). For a comprehensive list see Supplementary Table S1. Custom scoring functions can easily be created by implementing *AbstractSimFunc*. *AbstractSimFunc* takes care of peak alignment and supports plugging in peak filters and transformers.

The *io-core* package provides readers and writers for commonly used spectra and peptide spectrum matches (PSM) file formats. Spectra are modelled by the *MsnSpectrum* class, which contains the most important meta data that the formats support. The PSM readers split the information into two parts, a part to identify the spectrum (*SpectrumIdentifier*) and another that contains the peptide match information (*PeptideMatch*). This division reflects the diversity in the spectrum descriptors in the different PSM file formats. A list of PSM readers and their capabilities can be found in Table 2. The readers have been designed to be easily customized through object composition and inheritance. Object composition is used where there is a lot of variability in the format, for example the *mgf* title or *sptxt* comment tag. Inheritance is used where data has been added or removed from the file format. For example the *mgf* parser can be customized to read new tags or additional information from the peaks by overriding the *parseUnknownTag* or *handlePeakLine* method. Setting a *PeakProcessorChain* on spectrum readers allows the peaks to be transformed and filtered while reading. The list of spectrum readers and their attributes can be found in Supplementary Table S2 and S3.

Glycomics and proteomics

To support proteomics and glycomics applications MzJava has data structures to store, manipulate and generate theoretical spectra for proteins, peptides and glycans. Peptides are

represented by the *Peptide* class which holds an amino acid sequence and one or more post translational modifications for each residue or terminus. Modifications are defined in the code either by their chemical composition or mass. They can also be retrieved from the *UnimodModificationResolver* which wraps an *xml* dump of the UniMod database [39]. Peptides can be generated by digesting a *Protein* instance using *ProteinDigester* and associated classes. *ProteinDigester* can be configured to output peptides that have fixed and variable modifications. Support for the most important proteases is built-in and custom proteases can easily be added. Theoretical peptide spectra can be constructed by using the *PeptideFragmenter* class. *PeptideFragmenter* can be configured to produce spectra that contain backbone and neutral loss fragment peaks. To generate peaks for custom fragments an implementation of the *PeptidePeakGenerator* interface can be added to the *PeptideFragmenter*.

In MzJava polysaccharides are represented by the *Glycan* class, which holds a directed acyclic graph of *Monosaccharide* and *Substituent* nodes that are connected by *GlycosidicLinkage* edges. Glycan objects can be created by reading glycan structures in the GlycoCT format [40] using the *GlycoCTReader* class. The *GlycanFragmenter* class can be used to generate theoretical glycan spectra [21]. By default the *GlycanFragmenter* returns glycosidic, cross ring and neutral loss fragments, however custom fragments can be generated by implementing the *GlycanPeakGenerator* interface. The user can specify the list of *Monosaccharides* and *Substituents* in a configuration file. Glycan classes rely on recognised standard description of monosaccharides and full structures [22].

Hadoop and Spark support

A brief description of the Hadoop [23] and Spark [24] frameworks can be found in the Supplementary Material Section S1. The efficiency of the serialization can have a large impact on the performance of a distributed application. MzJava provides an efficient serialization mechanism for MzJava data classes that is based on Apache Avro (https://avro.apache.org/). Adapter classes are provided to integrate the MzJava serialization with the mechanisms that are used by Hadoop and Spark. This allows MzJava to be used within these frameworks directly without requiring any

additional serialization code. Hadoop provides a pluggable API allowing different serialization frameworks to be used. The *MzJavaSerialization* class implements this API to allow MzJava data objects to be automatically serialized by Hadoop.

Spark uses native Java or Kryo (https://github.com/EsotericSoftware/kryo) serialization. MzJava interoperates with Spark by providing a bridge between its classes and Kryo. In order to automatically serialize MzJava data objects Spark needs to be configured to use Kryo serialization and to use *MzJavaKryoRegistrator* for the registration of serializable MzJava classes. The MzJava serialization can be extended to handle new classes by implementing the AvroReader and AvroWriter interfaces and tagging them with the *ServiceProvider* interface. *MzJavaSerialization* and *MzJavaKryoRegistrator* then automatically use the reader and writer to serialize the new class.

Code quality

At the time of writing the MzJava library contains 26,793 lines of code in 529 classes. Since comparable libraries such as BioJava or Compomics have different purposes it appears rather difficult to benchmark performance. However since SonarQube calculates code quality scores, which are independent of the specific purpose of a library, these scores can be used to compare different projects. Thus we run SonarQube on some Java libraries and this process is regularly repeated to provide the time evolution of the scores. The results are displayed on a webpage (https://glycoproteome.isb-sib.ch/sonar). A subset of the code quality scores for some analysed Java libraries (BioJava, ms-data-core-api, jmzml, and Compomics) is displayed in Table 3. The scores are: the number of lines of code, number of classes, code complexity and issues per class, documentation content, test coverage and level of package tangle. Code complexity is a score calculated by SonarQube, which evaluates how large the methods are and how many nested code blocks they contain. Low complexity indicates clear code, which is less prone to error. Code issues are potential weak points in the code pointed out by SonarQube. MzJava compares well to the other libraries, and it has the highest test coverage and the lowest package tangle index.

Discussions

Code example MS/MS protein database search

Figure 3 shows the code required to perform a database search using the normalized dot product as a scoring function. The components that are required for a database search are: a database of theoretical spectra, a reader to read the experimental query spectra, a similarity function to calculate the score and a writer to write the results. First a PeptideFragmenter is created that generates b and y ion backbone peaks that have an intensity of 50 (line 3) and water loss peak for b and y ion fragments that contain a S, T, D or E amino acid (line 4 to 6). The generated water loss peaks have an intensity of 10. The fragmenter is then used while building a PeptideSpectrumDB that creates tryptic peptides with 6 to 60 amino acids from a fasta file containing protein sequences (line 9 to 13). Spectra are read from a mgf file using a MgfReader that pre-processes the raw spectra on the fly (line 15 to 20). The pre-processing is specified by providing a PeakProcessorChain. In this example the processor chain will process the raw spectra by centroiding (line 17), removing noise by keeping the most intense 2 peaks in a 10 m/z sliding window (line 18) and then square root transforming the intensities (line 19). The normalized dot product similarity function is initialized to require at least 5 peaks and use a tolerance of 0.02 Da when aligning peaks (line 22) and a result writer is built (line 23). The query spectra are then read one at a time (while loop line 24) and the similarity score is calculated (line 30) for each database spectrum whose precursor *m/z* is within tolerance of the query spectrum precursor mass (for loop line 28). This example could be extended to calculate a PSM score that is more sophisticated by providing a similarity function that is more appropriate.

Code example Spark

The next code example illustrates how to use MzJava in the Apache Spark framework in order to search query spectra against a spectrum library (Figure 4). First we need to configure Spark to use Kryo and MzJava serialization to serialize MzJava classes automatically (lines 2 and 3). Then

the library spectra are read from a.*sptxt* file and these spectra are made available to all nodes on the computer cluster (lines 6-8). Line 10 defines a *PairFunction* instance to map a query spectrum to a list of score-peptide pairs; one pair for each library spectrum that is within tolerance of the query spectrum precursor mass and has a high similarity score. In order to calculate the score of a match between query and library spectrum, a normalized dot product similarity function is created with a fragment tolerance of 0.02Da (line 20) and an instance of *DefaultSpectrumLibrary* (line 21), which retrieves all broadcasted library spectra within 20 ppm of the query spectrum precursor mass (line 35). For each retrieved library spectrum, the similarity to the query spectrum is calculated and if this similarity is larger than 0.6, the library spectrum will be added to the list of results (lines 30-32). On line 41 Spark reads the query spectra from a Hadoop sequence file into a Spark RDD, and partitions these spectra across the nodes of the cluster. Finally, Spark sends the library search function to all nodes of the cluster and applies it to the library spectra on these nodes. Line 43 saves the results to a Spark object file. Supplementary Figure S1 shows an example of how to use MzJava within the Hadoop *map-reduce* framework.

Code example glycomics

Supplementary Figure S2A describes how to build a glycan molecule using the MzJava *Glycan.Builder* class. In this example, the N-glycan core, i.e. Man(a1-3)[Man(a1-6)] Man(b1-4)GlcNAc(b1-4)GlcNAc(b1-, is built. First, monosaccharides and substitutes are read from a configuration file. The glucose root node is added and a NAcetyl is attached to the second carbon of glucose. Then one more GlcNAc and three mannoses are added to obtain the N-glycan core structure. In Supplementary Figure S2B we describe a code example for the fragmentation of GalNAc(b1-?)Gal. The glycan structure is read from a GlycoCT encoded string using the *GlycoCTReader* class. Then the ion types and fragment types are set. A *GlycanFragmenter* object is created with the maximal numbers of glycosidic and crossring fragments set to 1. Finally a glycan fragment spectrum of singly charged fragments is constructed.

Code quality

The SonarQube results show that the MzJava code quality compares favourably to that of other widely used open source projects. We have found that using SonarQube is an effective method for discovering problems in the MzJava code before they lead to bugs in research code. We also use SonarQube to track test coverage, technical debt and documentation coverage. The good test coverage and low technical debt of MzJava makes the code more robust and maintainable. High test coverage means that errors introduced during development are more likely to make some test fail and can therefore be detected. The documentation in MzJava is fairly available and should keep on growing.

Conclusions

MzJava provides a well-engineered and well-tested API with building blocks commonly required to write software processing mass spectrometry data in research project. This speeds up the development of new mass spectrometry software, allows bioinformaticians to focus on writing new algorithms and innovative solutions and helps avoiding boilerplate code. In terms of code quality it compares favourably to other open source Java projects. Besides classes to deal with proteomics data, MzJava also provides new functionality to fragment glycans and annotate glycan spectra. Interfaces to Hadoop MapReduce and Apache Spark facilitate large scale computing and parallelization of the code often required in systems biology applications.

API: Application Programming Interface

MS: Mass Spectrometry

LC: Liquid Chromatography

Availability

MzJava is an open-source project distributed under the AGPL v3.0 license. MzJava requires Java 1.7 or higher. Binaries, source code and documentation can be downloaded from http://mzjava.expasy.org and https://bitbucket.org/sib-pig/mzjava.

Acknowledgments

This work has been and is supported by the Swiss National Science Foundation [SNSF 315230 130830, 31003A 141215 and CRSII3 136282] and EU (FP7-PEOPLE-2012-ITN #316929).

References

[1] R. Aebersold and M. Mann, "Mass spectrometry-based proteomics," *Nature*, vol. 422, no. 6928, pp. 198–207, Mar. 2003.

[2] M. Wuhrer, "Glycomics using mass spectrometry," *Glycoconj. J.*, vol. 30, no. 1, pp. 11–22, Jan. 2013.

[3] Y. Perez-Riverol, R. Wang, H. Hermjakob, M. Müller, V. Vesada, and J. A. Vizcaíno, "Open source libraries and frameworks for mass spectrometry based proteomics: A developer's perspective," *Biochim. Biophys. Acta BBA - Proteins Proteomics*, vol. 1844, no. 1, Part A, pp. 63–76, Jan. 2014.

[4] A. Keller, A. I. Nesvizhskii, E. Kolker, and R. Aebersold, "Empirical statistical model to estimate the accuracy of peptide identifications made by MS/MS and database search," *Anal. Chem.*, vol. 74, no. 20, pp. 5383–5392, Oct. 2002.

[5] O. Kohlbacher, K. Reinert, C. Gröpl, E. Lange, N. Pfeifer, O. Schulz-Trieglaff, and M. Sturm, "TOPP—the OpenMS proteomics pipeline," *Bioinformatics*, vol. 23, no. 2, pp. e191–e197, Jan. 2007.

[6] M. Sturm, A. Bertsch, C. Gröpl, A. Hildebrandt, R. Hussong, E. Lange, N. Pfeifer, O. Schulz-Trieglaff, A. Zerck, K. Reinert, and O. Kohlbacher, "OpenMS – An open-source software framework for mass spectrometry," *BMC Bioinformatics*, vol. 9, p. 163, Mar. 2008.

[7] D. Kessner, M. Chambers, R. Burke, D. Agus, and P. Mallick, "ProteoWizard: open source software for rapid proteomics tools development," *Bioinformatics*, vol. 24, no. 21, pp. 2534–2536, Nov. 2008.

[8] C. Steinbeck, Y. Han, S. Kuhn, O. Horlacher, E. Luttmann, and E. Willighagen, "The Chemistry Development Kit (CDK): An Open-Source Java Library for Chemo- and Bioinformatics," *J. Chem. Inf. Comput. Sci.*, vol. 43, no. 2, pp. 493–500, Mar. 2003.

[9] A. Prlić, A. Yates, S. E. Bliven, P. W. Rose, J. Jacobsen, P. V. Troshin, M. Chapman, J. Gao, C. H. Koh, S. Foisy, R. Holland, G. Rimša, M. L. Heuer, H. Brandstätter–Müller, P. E. Bourne, and S. Willis, "BioJava: an open-source framework for bioinformatics in 2012," *Bioinformatics*, vol. 28, no. 20, pp. 2693–2695, Oct. 2012.

[10] M. Vaudel, H. Barsnes, F. S. Berven, A. Sickmann, and L. Martens, “SearchGUI: An open-source graphical user interface for simultaneous OMSSA and X!Tandem searches,” *PROTEOMICS*, vol. 11, no. 5, pp. 996–999, Mar. 2011.
[11] M. Vaudel, J. M. Burkhart, R. P. Zahedi, E. Oveland, F. S. Berven, A. Sickmann, L. Martens, and H. Barsnes, “PeptideShaker enables reanalysis of MS-derived proteomics data sets,” *Nat. Biotechnol.*, vol. 33, no. 1, pp. 22–24, Jan. 2015.
[12] H. Barsnes, M. Vaudel, N. Colaert, K. Helsens, A. Sickmann, F. S. Berven, and L. Martens, “compomics-utilities: an open-source Java library for computational proteomics,” *BMC Bioinformatics*, vol. 12, no. 1, p. 70, Mar. 2011.
[13] F. Gluck, C. Hoogland, P. Antinori, X. Robin, F. Nikitin, A. Zufferey, C. Pasquarello, V. Fétaud, L. Dayon, M. Müller, F. Lisacek, L. Geiser, D. Hochstrasser, J.-C. Sanchez, and A. Scherl, “EasyProt — An easy-to-use graphical platform for proteomics data analysis,” *J. Proteomics*, vol. 79, pp. 146–160, Feb. 2013.
[14] E. Ahrné, F. Nikitin, F. Lisacek, and M. Müller, “QuickMod: A Tool for Open Modification Spectrum Library Searches,” *J. Proteome Res.*, vol. 10, no. 7, pp. 2913–2921, Jul. 2011.
[15] H. Pak, F. Nikitin, F. Gluck, F. Lisacek, A. Scherl, and M. Muller, “Clustering and Filtering Tandem Mass Spectra Acquired in Data-Independent Mode,” *J. Am. Soc. Mass Spectrom.*, vol. 24, no. 12, pp. 1862–1871, Dec. 2013.
[16] Y. Perez-Riverol, J. Uszkoreit, A. Sanchez, T. Ternent, N. del Toro, H. Hermjakob, J. A. Vizcaíno, and R. Wang, “ms-data-core-api: An open-source, metadata-oriented library for computational proteomics,” *Bioinformatics*, p. btv250, Apr. 2015.
[17] E. K. Nelson, B. Piehler, J. Eckels, A. Rauch, M. Bellew, P. Hussey, S. Ramsay, C. Nathe, K. Lum, K. Krouse, D. Stearns, B. Connolly, T. Skillman, and M. Igra, “LabKey Server: An open source platform for scientific data integration, analysis and collaboration,” *BMC Bioinformatics*, vol. 12, no. 1, p. 71, Mar. 2011.
[18] A. Bauch, I. Adamczyk, P. Buczek, F.-J. Elmer, K. Enimanev, P. Glyzewski, M. Kohler, T. Pylak, A. Quandt, C. Ramakrishnan, C. Beisel, L. Malmström, R. Aebersold, and B. Rinn, “openBIS: a flexible framework for managing and analyzing complex data in biology research,” *BMC Bioinformatics*, vol. 12, no. 1, p. 468, Dec. 2011.
[19] J. Häkkinen, G. Vincic, O. Månsson, K. Wårell, and F. Levander, “The Proteios Software Environment: An Extensible Multiuser Platform for Management and Analysis of Proteomics Data,” *J. Proteome Res.*, vol. 8, no. 6, pp. 3037–3043, Jun. 2009.
[20] T. Lütteke and M. Frank, Eds., “Annotation of Glycomics MS and MS/MS Spectra Using the GlycoWorkbench Software Tool - Springer,” Springer New York, 2015.
[21] B. Domon and C. E. Costello, “A systematic nomenclature for carbohydrate fragmentations in FAB-MS/MS spectra of glycoconjugates,” *Glycoconj. J.*, vol. 5, no. 4, pp. 397–409, Dec. 1988.
[22] M. P. Campbell, R. Ranzinger, T. Lütteke, J. Mariethoz, C. A. Hayes, J. Zhang, Y. Akune, K. F. Aoki-Kinoshita, D. Damerell, G. Carta, W. S. York, S. M. Haslam, H. Narimatsu, P. M. Rudd, N. G. Karlsson, N. H. Packer, and F. Lisacek, “Toolboxes for a standardised and systematic study of glycans,” *BMC Bioinformatics*, vol. 15 Suppl 1, p. S9, 2014.
[23] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Commun ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.
[24] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: cluster computing with working sets.,” *Proc. 2nd USENIX Conf. Hot Top. Cloud Comput.*, 2010.
[25] R. C. Taylor, “An overview of the Hadoop/MapReduce/HBase framework and its current applications in bioinformatics,” *BMC Bioinformatics*, vol. 11, no. Suppl 12, p. S1, Dec. 2010.
[26] B. Pratt, J. J. Howbert, N. I. Tasman, and E. J. Nilsson, “MR-Tandem: parallel X!Tandem using Hadoop MapReduce on Amazon Web Services,” *Bioinformatics*, vol. 28, no. 1, pp. 136–137, Jan. 2012.

[27] A. Kalyanaraman, W. R. Cannon, B. Latt, and D. J. Baxter, “MapReduce Implementation of a Hybrid Spectral Library-Database Search Method for Large-scale Peptide Identification,” *Bioinformatics*, 2011.
[28] C.-L. Hung and G.-J. Hua, “Cloud Computing for Protein-Ligand Binding Site Comparison,” *BioMed Res. Int.*, vol. 2013, 2013.
[29] M. S. Wiewiórka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, “SparkSeq: fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision,” *Bioinformatics*, vol. 30, no. 18, pp. 2652–2653, Sep. 2014.
[30] J. Freeman, N. Vladimirov, T. Kawashima, Y. Mu, N. J. Sofroniew, D. V. Bennett, J. Rosen, C.-T. Yang, L. L. Looger, and M. B. Ahrens, “Mapping brain activity at scale with cluster computing,” *Nat. Methods*, vol. 11, no. 9, pp. 941–950, Sep. 2014.
[31] J. Bloch, “How to design a good API and why it matters.,” *OOPSLA 06*, 2006.
[32] J. T. Dudley and A. J. Butte, “A Quick Guide for Developing Effective Bioinformatics Programming Skills,” *PLoS Comput Biol*, vol. 5, no. 12, p. e1000589, Dec. 2009.
[33] G. K. Sandve, A. Nekrutenko, J. Taylor, and E. Hovig, “Ten Simple Rules for Reproducible Computational Research,” *PLoS Comput Biol*, vol. 9, no. 10, p. e1003285, Oct. 2013.
[34] G. Wilson, D. A. Aruliah, C. T. Brown, N. P. Chue Hong, M. Davis, R. T. Guy, S. H. D. Haddock, K. D. Huff, I. M. Mitchell, M. D. Plumbley, B. Waugh, E. P. White, and P. Wilson, “Best Practices for Scientific Computing,” *PLoS Biol*, vol. 12, no. 1, p. e1001745, Jan. 2014.
[35] F. da V. Leprevost, V. C. Barbosa, E. L. Francisco, Y. Perez-Riverol, and P. C. Carvalho, “On best practices in the development of bioinformatics software,” *Bioinforma. Comput. Biol.*, vol. 5, p. 199, 2014.
[36] K. Beck, *Test-Driven Development: by Example*. Addison Wesley, 2003.
[37] P. M. Duvall, S. Matyas, and A. Glover, *Continuous integration: improving software quality and reducing risk.* Addison-Wesley Professional, 2007.
[38] R. G. Côté, F. Reisinger, and L. Martens, “jmzML, an open-source Java API for mzML, the PSI standard for MS data,” *PROTEOMICS*, vol. 10, no. 7, pp. 1332–1335, Apr. 2010.
[39] D. M. Creasy and J. S. Cottrell, “Unimod: Protein modifications for mass spectrometry,” *PROTEOMICS*, vol. 4, no. 6, pp. 1534–1536, Jun. 2004.
[40] S. Herget, R. Ranzinger, K. Maass, and C.-W. v. d. Lieth, “GlycoCT—a unifying sequence format for carbohydrates,” *Carbohydr. Res.*, vol. 343, no. 12, pp. 2162–2171, Aug. 2008.

Figure legends

Figure 1

A) This cycle illustrates the overall development process of MzJava based on unit tests (JUnit) and subsequent checks to produce robust code. This figure is an adaptation of an image from the blog http://technicalknowledgeabstracts.blogspot.ch/2014/02/tdd-test-driven-development-in-practice.html B) This cycle provides details of tasks undertaken in the cycle described in A).

Figure 2

This chart represents the 3 modules composing MzJava. The core module contains the code for MS data handling and processing. The proteomics and glycomics modules contain the code specific for peptides/proteins and glycans, respectively. io: input-output, mol: molecules, ms: mass spectrometry, utils: utilities, stats: statistics

Figure 3

MzJava code example for a MS/MS protein database search.

Figure 4

MzJava code example for a spectrum library search using the Spark framework.

Table 1

Class	Description
Filters	
BinnedSpectrumFilter	Bins a spectrum. Peaks will still be stored as a PeakList, but the mz values will be regularly spaced on the centres of the bins
CentroidFilter	Selects all peaks that are local maxima and replaces their mass/intensity values by the centroid values calculated within a neighbourhood
FragmentAnnotationFilter	Select peaks with annotations that fulfil a certain condition
LibraryMergePeakFilter	Cluster peaks with similar *m/z* values
MzRangeFilter	Select or exclude peaks in certain *m/z* ranges.
NPeaksFilter	Selects the N most intense peaks
NPeaksPerBinFilter	Retains the top N peaks for every fixed *m/z* bin
NPeaksPerSlidingWindowFilter	Retains the top N peaks in a sliding *m/z* window
ThresholdFilter	Removes any peaks that have an intensity that is smaller than a threshold
Transformers	
AffineTransformer	Performs affine transformation on the intensities
ArcsinhTransformer	Performs first an affine transformation of peak intensities followed by a arcsinh transformation. This transformation was shown to stabilize the intensity variance.
ContrastEnhancingTransformer	For each peak, the average of all intensities within an m/z window centred at the peak is subtracted. This can be used for an efficient calculation of the Sequest XCorr
InverseArcsinhTransformer	Performs the inverse of the ArcsinhTransformer
Log10Transformer	Takes log10 values of all (intensities+1)
LogTransformer	Takes log values of all (intensities+1)
MzAffineTransformer	Performs affine transformation on the *m/z* values
SqrtTransformer	Square root transforms all intensity values
Normalizers	
IonCurrentNormalizer	Sum of all peak intensities is set to 1
UnitVectorNormalizer	Euclidian norm of peak intensities is set to 1
NthPeakNormalizer	Peak intensities are divided by the intensity of the N-th highest peak
RankNormalizer	Replaces peak intensities by 1- *r/N*, where *N* is the number of peaks and *r* the rank of the peak intensity (*r* = 0:highest, … *r* = N-1:lowest)
HighestPeakPerBinNormalizer	Normalizes peak intensities by dividing the *m/z* range into bins and scaling the peaks in each bin such that the most intense peaks in each bin have the same value

Table 1

Table 2

	MaxQuantPsmReader	MODaPsmReader	MzIdentMlReader	PepXmlReader	ProteinPilotPsmReader
File Type	MaxQuant csv	MODa	PSI mzIdentML	ISB pepXML	ProteinPilot csv
Spectrum Name	✓	✓	✓	✓	✓
Scan numbers	✓	✓	✓	✓	×
Retention times	✓	×	✓	×	✓
Precursor neutral mass	✓	✓	✓	✓	✓
Precursor intensity	×	×	×	✓	×
Assumed charge	✓	✓	✓	✓	✓
Index	✓	✓	✓	✓	×
Spectrum source file	×	✓	✓	×	×
Rank	✓	✓	✓	✓	✓
Peptide Sequence	✓	✓	✓	✓	✓
Number matched ions	×	×	×	✓	×
Total number of ions	✓	×	×	✓	×
Mass difference	✓	✓	✓	✓	✓
Missed cleavages	✓	×	×	✓	×
Rejected	×	×	✓	✓	×
Modifications	✓	✓	✓	✓	✓
Neutral peptide mass	×	✓	×	×	×
Protein	✓	✓	✓	✓	✓

Table 2 PSM Readers

Table 3

	mzjava	biojava	ms-data-core-api	jmzml	compomics
	1.1.0	4.0.0-SNAPSHOT	0.11.27-SNAPSHOT	1.7.2-SNAPSHOT	3.49.7
Lines of code	26,793	100,734	14,243	6,382	89,866
Classes	529	1,024	126	144	523
Public documented API (%)	61.0%	48.5%	55.6%	70.6%	79.3%
Duplicated lines (%)	3.1%	5.3%	4.4%	3.8%	7.3%
Complexity / class	12.5	22.2	32.3	8.7	39.4
Issues / class	1	12	5	2	21
Test Coverage	85.0%	36.2%	34.3%	43.4%	11.6%
Package tangle index	0.0%	17.7%	6.4%	4.5%	23.7%

Table 3

Figure 1A

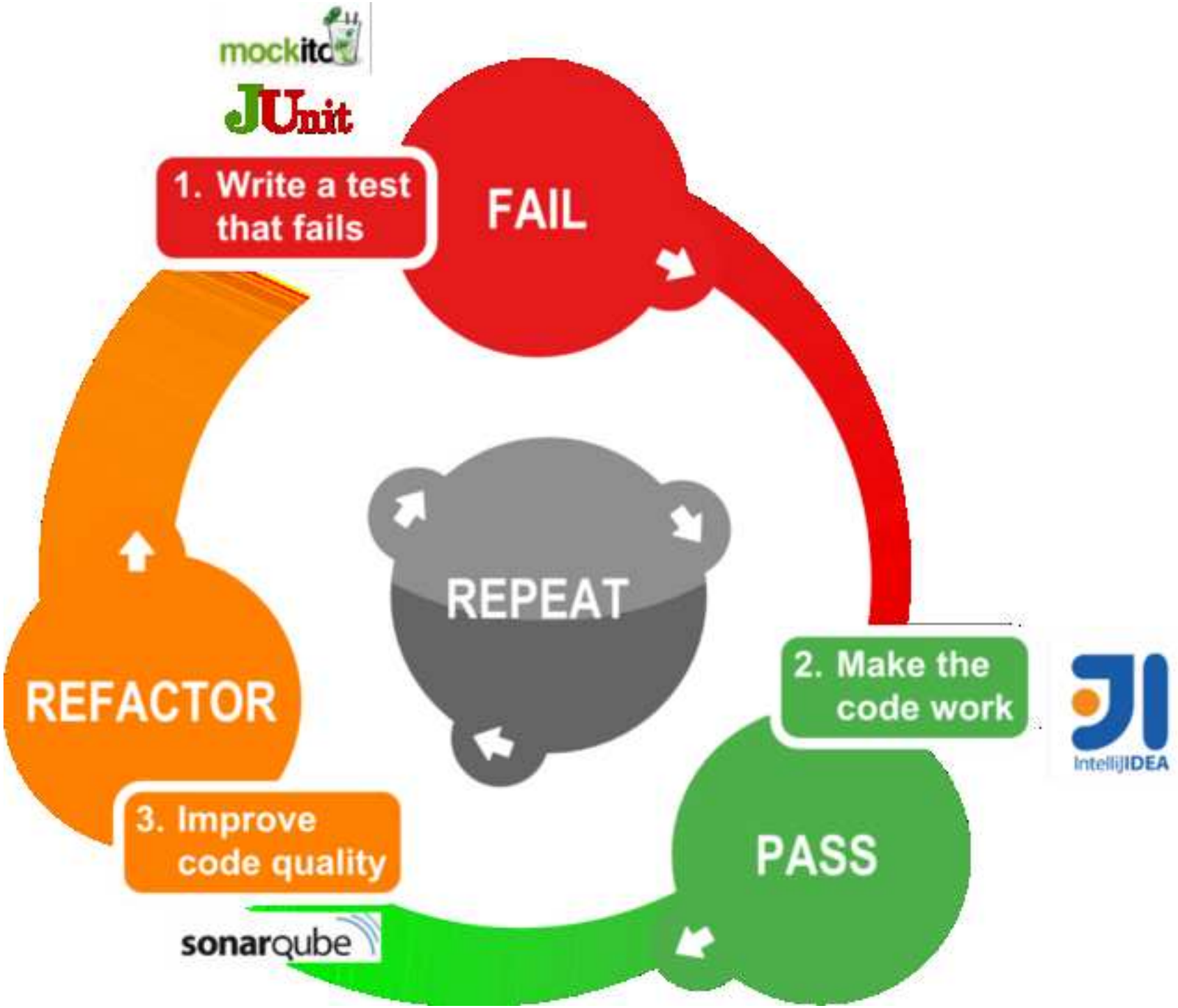
mockito
JUnit
1. Write a test that fails
FAIL
2. Make the code work
PASS
3. Improve code quality
REFACTOR
REPEAT
sonarqube
IntelliJIDEA

Figure 1B

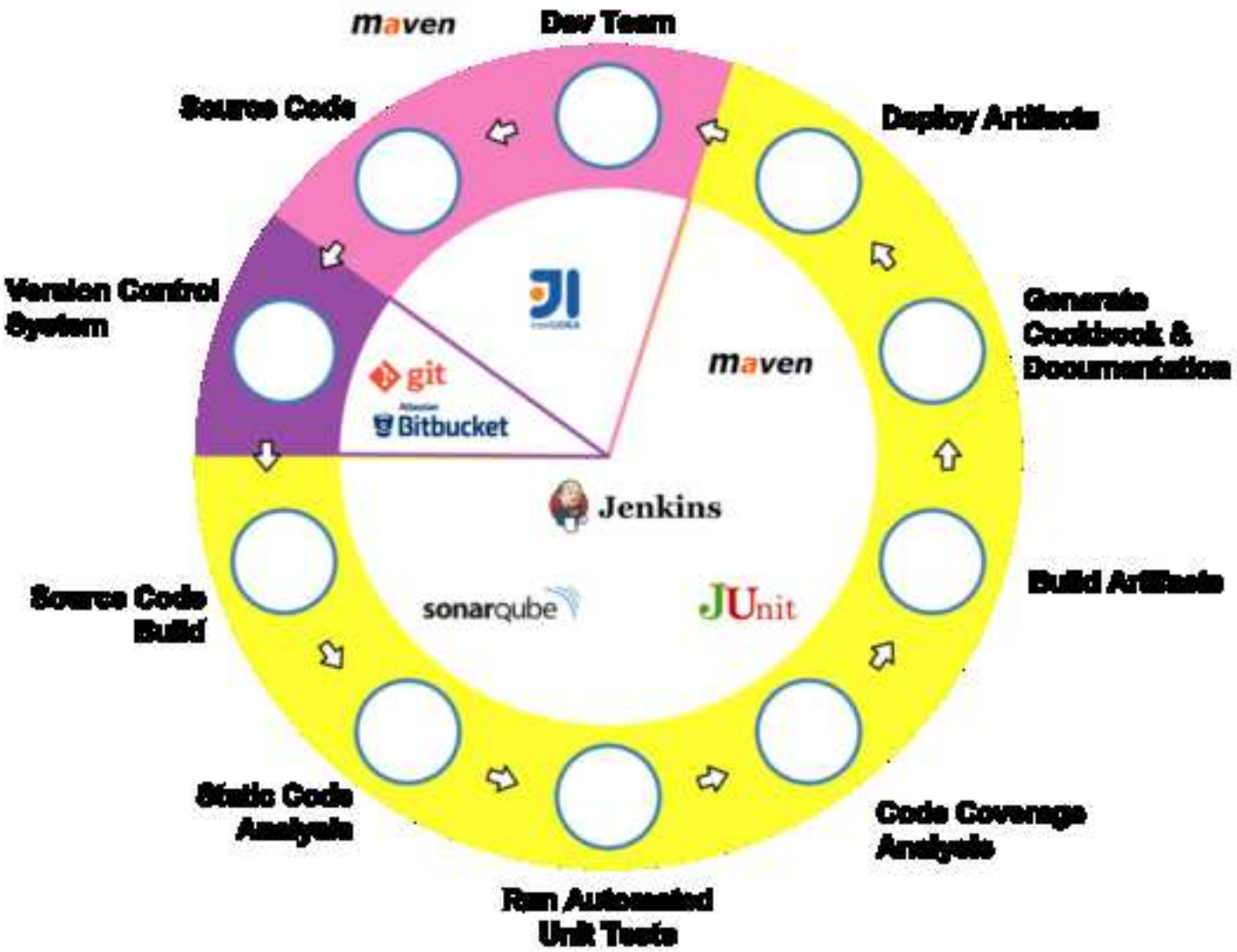
maven
Dev Team
Source Code
Deploy Artifacts
Version Control System
Generate Cookbook & Documentation
git
Bitbucket
maven
Jenkins
Source Code Build
Build Artifacts
sonarqube
JUnit
Static Code Analysis
Code Coverage Analysis
Run Automated Unit Tests

Figure 2

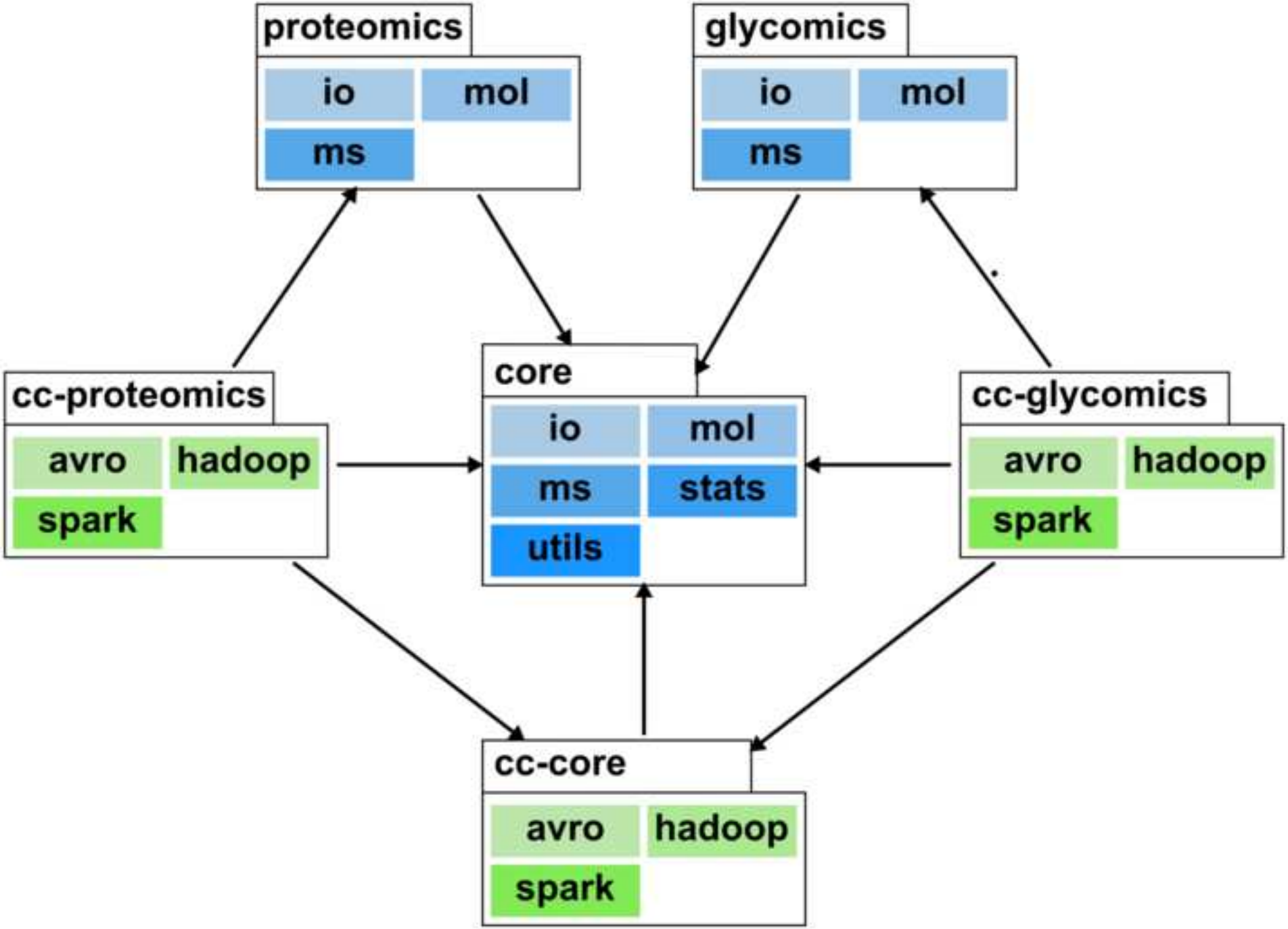
proteomics
io
mol
ms
glycomics
io
mol
ms
core
io
mol
ms
stats
utils
cc-proteomics
avro
hadoop
spark
cc-glycomics
avro
hadoop
spark
cc-core
avro
hadoop
spark

Figure 3

```
final PeptideFragmenter peptideFragmenter = new PeptideFragmenter(
        Lists.newArrayList(
                new BackbonePeakGenerator(EnumSet.of(IonType.b, IonType.y), 50),
                new PeptideNeutralLossPeakGenerator(Composition.parseComposition("H-2O-1"),
                        EnumSet.of(AminoAcid.S, AminoAcid.T, AminoAcid.D, AminoAcid.E),
                        EnumSet.of(IonType.b, IonType.y), 10.0)
        ), PeakList.Precision.FLOAT);

final PeptideSpectrumDB peptideSpectrumDB = PeptideSpectrumDB.newBuilder()
        .setPrecursorTolerance(new PpmTolerance(10))
        .setProteinSource(new File("C:\\proteins.fasta")).digestWith(Protease.TRYPSIN).retainPeptidesOfLength(6, 60)
        .setMissedCleavagesTo(2).generateSpectraWith(peptideFragmenter)
        .build();

final MgfReader reader = new MgfReader(new File("C:\\spectra.mgf"), PeakList.Precision.FLOAT,
        new PeakProcessorChain<>()
                .add(new CentroidFilter<>(0.05, CentroidFilter.IntensityMode.HIGHEST))
                .add(new NPeaksPerSlidingWindowFilter<>(2, 10))
                .add(new SqrtTransformer<>())
);

final SimFunc<PepFragAnnotation, PeakAnnotation> simFunc = new NdpSimFunc<>(5, new AbsoluteTolerance(0.02));
final PepXmlWriter resultsWriter = PepXmlWriterBuilder.create(EngineType.CUSTOM, "ndp").build();
while (reader.hasNext()) {

    final MsnSpectrum querySpectrum = reader.next();
    final List<PeptideMatch> peptideMatchList = new ArrayList<>();
    for (PeptideSpectrum theoreticalSpectrum peptideSpectrumDB.getSpectra(querySpectrum.getPrecursor())){

        final double score = simFunc.calcSimilarity(theoreticalSpectrum, querySpectrum);
        peptideMatchList.add(new PeptideMatch(theoreticalSpectrum, "ndp", score));
    }
    resultsWriter.add(querySpectrum, peptideMatchList);
}
reader.close();
resultsWriter.write(new File("C:\\results.pep.xml"));
```

Figure 4

```
final SparkConf conf = new SparkConf()
        .set("spark.serializer", "org.apache.spark.serializer.KryoSerializer")
        .set("spark.kryo.registrator", "org.expasy.mzjava.spark.MzJavaKryoRegistrator");
final JavaSparkContext sc = new JavaSparkContext(conf);

final Broadcast<List<PeptideConsensusSpectrum>> libSpectraBroadcast = sc.broadcast(
        IterativeReaders.toArrayList(new SptxtReader(new File(libSpectraPath), PeakList.Precision.DOUBLE))
);

final PairFunction<MsnSpectrum, MsnSpectrum, List<Tuple2<Double, Peptide>>> libSearchFunction =
            new PairFunction<MsnSpectrum, MsnSpectrum, List<Tuple2<Double, Peptide>>>() {

   private transient SimFunc<PepLibPeakAnnotation, PeakAnnotation> simFunc;
   private transient SpectrumLibrary<PeptideConsensusSpectrum> library;

    @Override
    public Tuple2<MsnSpectrum, List<Tuple2<Double, Peptide>>> call(final MsnSpectrum querySpectrum) throws Exception {

        if (simFunc == null) {
            simFunc = new NdpSimFunc<>(0, new AbsoluteTolerance(0.02));
            library = new DefaultSpectrumLibrary<>(new PpmTolerance(20), libSpectraBroadcast.getValue());
        }

        final List<Tuple2<Double, Peptide>> peptideMatches = new ArrayList<>();
        final Procedure<PeptideConsensusSpectrum> simFuncProcedure = new Procedure<PeptideConsensusSpectrum>() {

            @Override
            public void execute(PeptideConsensusSpectrum libSpectrum) {

                double score = simFunc.calcSimilarity(libSpectrum, querySpectrum);
                if (score > 0.6)
                    peptideMatches.add(new Tuple2<>(score, libSpectrum.getPeptide()));
            }
        };
        library.forEach(querySpectrum.getPrecursor(), simFuncProcedure);

        return new Tuple2<>(querySpectrum, peptideMatches);
    }
};

final JavaRDD<MsnSpectrum> querySpectra = MzJavaSparkUtils.msnSpectra(sc, querySpectraPath);
final JavaPairRDD<MsnSpectrum, List<Tuple2<Double, Peptide>>> searchResults = querySpectra.mapToPair(libSearchFunction);
searchResults.saveAsObjectFile(outputPath);
```

Graphical Abstract (for review)