JOHANNES KEPLER
UNIVERSITY LINZ

Author
Anna Saibold

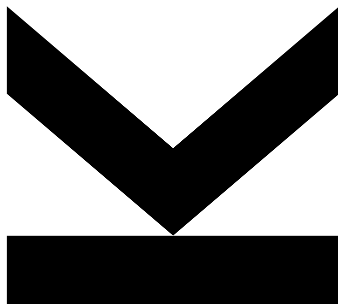
Submission
Institute of Bioinformatics

Thesis Supervisor
Werner Retschitzegger
Elisabeth Kapsammer

June 2018

Application of Data Engineering Technologies in Bioinformatics

Prototypical Implementation of a Selected Case Study



Bachelor’s Thesis

to confer the academic degree of

Bachelor of Science

in the Bachelor’s Program

Bioinformatics

JOHANNES KEPLER
UNIVERSITY LINZ
Altenberger Str. 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Abstract

Due to the huge amount of biological data, the various available data sources, and the diversity of their structure as well as content, data engineering technologies are required. They provide an important opportunity to support their exploitation. This thesis aims at applying several data engineering steps to a particular real-world data source to demonstrate the additional benefit with respect to utilization of the data by means of connecting to other data sources as well as querying and analyzing the data. Therefore, in the practical part of this thesis a continuous example showing several engineering steps is constructed, comprising the development of different schemata, the creation of a database as well as the mapping and integration of future heterogeneous data. Finally, processing queries against the engineered data source is compared to an online database search regarding different aspects like time, effort, and usability. As the example shows, an engineered database can have huge benefits over online search, especially for complex queries, processing data from several sources.

JOHANNES KEPLER UNIVERSITY LINZ

Contents

1. Introduction………………………………………………………….. 5
2. Data Resources in Bioinformatics………………………………… 6
   2.1. Data Bases……………………………………………….... 7
      2.1.1. NCBI – GeneBank……………………………………. 8
      2.1.2. UniProt – SwissProt………………………………….. 8
      2.1.3. RCSB – PDB………………………………………….. 8
      2.1.4. EMBL – ENA………………………………………….. 9
      2.1.5. Comparison of Data Bases………………………….. 9
   2.2. Data Formats…………………………………………….…. 10
      2.2.1. XML…………………………………………………….. 10
      2.2.2. JSON…………………………………………………… 12
      2.2.3. FASTA…………………………………………………. 13
      2.2.4. Relational Database Model……………..…………... 13
      2.2.5. Flat File Format………..…………………………….... 14
3. Integration of Biological Data……………………………….... 14
   3.1. Challenges…………………………………………….……. 15
   3.2. Approaches...…………………………………………..……16
   3.3. Example: KEGG………………………………………….… 20
4. Practical Work……………………………………………………... 23
   4.1. Aims……………………………………………………….… 23
   4.2. Comparison of Data Sources………………………….... 23
      4.2.1. Immunization…………………………………………. 24
      4.2.2. Allergy Intolerance…………………………………… 24
      4.2.3. Single Nucleotide Polymorphism………………….. 25
      4.2.4. Human Protein Atlas………………………………… 25
      4.2.5. Cancer Gene Disease………………………………. 26
      4.2.6. Comparison of Data Sources………………………. 27
   4.3. Example: Cancer Gene Disease...………………...…… 29
      4.3.1. UML Class Diagram……..………………………….. 29
      4.3.2. Data Set and XPath Queries……………………….. 31
      4.3.3. Schemata – DTD and XML Schema……………….. 36
      4.3.4. Database Creation and Queries…..………………... 43
      4.3.5. Online Database Search…………………………….. 50

JOHANNES KEPLER UNIVERSITY LINZ

4.3.6. Integration.. 52
a) Mapping of Database to XML Schema........... 52
b) Integration of SwissProt.............................. 54
c) Integration of Prosite.................................. 58
d) Mapping XML Schema to JSON................... 60
e) Integration of Gene Ontology GO................. 63
4.3.7. Results... 66
4.4. Discussion... 69
5. Conclusion.. 70
6. References.. 71
6.1. Literature... 71
6.2. Online Resources.. 74
7. List of Figures.. 76
8. List of Tables.. 76
9. List of Listings.. 76
10. Appendix A: UML Class Diagrams.................................. 77
11. Appendix B: Program Code.. 103

1. Introduction

Over the last decades a completely new intersection of biology and computer science developed: Bioinformatics. The goal for bioinformaticians is, for example, to analyze biological data and explore their processes. Furthermore, this interdisciplinary science can be supposed to give an overview of the processes by representative models [AnSt15]. Generally, today’s bioinformatic work can be divided in service, which means the retrieval and storage of biological data, and research, which means the analysis of this data by creating complex workflows or pipelines [AtPT16]. One important reason for evolving this discipline is the constantly growing and more sophisticated data and the resulting need for adequate storage and analyzation mechanisms. In this thesis, an exemplary process from storage of the data, over processing to the final step of a subsequent analysis is described and practically performed. In the practical work, a data set with optimal properties for the next steps is searched, the according UML class diagram is created, and schemata in XML and DTD are developed. Then, a relational database with an according structure is created and data is inserted. Database queries in different languages like SQL, XQuery, and XPath are performed. Different concepts of integration are shown: Schemata with and without heterogeneities are mapped, an XML file from SwissProt [SwPr] and a flat file from Prosite [Pros] are integrated to the data and Gene Ontology [GeOn] is used to extend knowledge. This workflow is compared to a similar online search. The purpose of this work is to demonstrate the actual challenges and benefits of data engineering on a real-world data example.

The outline of the thesis is as follows: Chapter 2 gives an overview on data resources in bioinformatics by presenting important data bases and widely-used data formats. Chapter 3 discusses challenges and approaches for integration of biological data, with KEGG as example. Chapter 4 comprehends the practical work from the comparison and selection of data, over the construction of an UML class diagram to the final revised version of the data. Further, schemata in XML and DTD are design, based on the structure of the UML class diagram.

Then, an according SQL database is created, and several queries are performed. Afterwards, an online database search is accomplished, and time and effort estimated. Then, various integration steps are described like mapping the database to the XML schema, integration of another schema from UniProt [UniP], integration of a flat file from Prosite [Pros], mapping of heterogeneous data and queries at each step. Another form of integration presented, is integration by means of ontology. An extract from Gene Ontology [GeOn] is used to gain more information. Next, the effort, time and usability from database and online search are set in comparison and the methods and results are discussed. Finally, chapter 5 concludes that data engineering in bioinformatics can have great benefits.

2. Data Resources in Bioinformatics

Since the technology to extract biological data evolved and the resulting data grows exponentially, there is the necessity for more storage capacity with more accuracy. For example, the genome of one person, determined by current sequencing methods, has around 1 terabyte [Neil14].

The total disc storage at EMBL-EBI centrum in 2015 contains more than 70 petabytes, which is more than 70 000 terabytes.

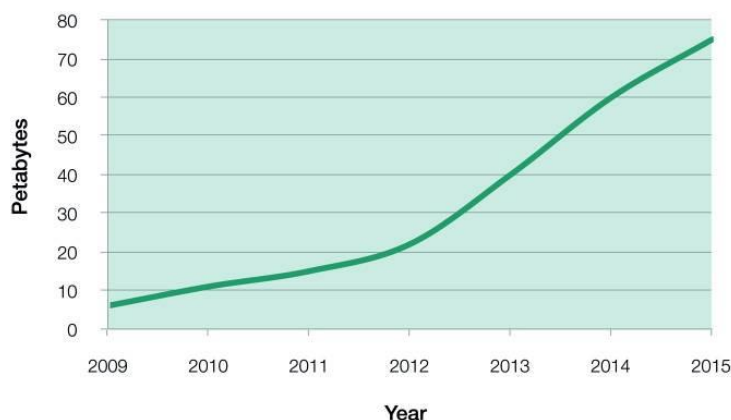


Fig. 1. Total Disk Storage at EMBL-EBI [CBFC16].

On the one hand, the storage facilities itself get improved and can store higher amounts of data more efficiently. On the other hand, the problem is not only the mass of data but also the accessibility and evaluability. If each laboratory and company store their results on private mass storages or local servers, there can hardly be an exchange of knowledge between researchers. “Historically there has been very limited cross-linking between biomaterial collections, registries, genomics, and trial data, with the exception of individual clinical research centers, where all the information may be held by a single investigator” [GaFR17].

To get optimal research results, it is necessary to share and cross evaluate data. In the following, the most typical data resources and formats in the field of bioinformatics are described. Afterwards, in chapter 4 the combination of data by integration is depicted.

2.1. Data Bases

To make biological data like genomes, their sequences, and their structures public, the world wide web is used. There are many platforms providing webservices to share the findings with the community. The data is commonly stored in different kinds of data bases. These data bases are usually the basis for data retrieval and analysis.
Some of them are ‘curated’, which means that they are checked for their correctness and quality regularly. All of them are focused on different areas of biology, since it is neither reasonable nor technically possible to create one large general-purpose database. Therefore, the number of databases enlarged incredibly over the last years. In 2017 there are 54 new databases registered according to the Database Issue of Nucleic Acids Research [GaFR17]. Through all these different data collections there exist great invariancies in semantics, data access, and variety of services and tools [LSJV15]. In the following, four of the pioneers in the area of bioinformatics are selected and compared. They were some of the first ones to offer web-based public data sharing platforms and are still under the most popular ones.

2.1.1. NCBI – GenBank

The National Center for Biotechnology Information released a genetic sequence database called ‘GenBank’. It is an annotated collection of all publicly available DNA sequences in nucleotide form and a daily data exchange is performed [GenB]. It is a primary database and not curated. This means, it has no restrictions for uploading genomes or parts of genomes. Errors and redundant (parts of) sequences cannot be ruled out. The database entries can be exported in various formats, among others their own ‘GenBank’ format, the sequence-format FASTA, and XML, which is not bioinformatic specific.

2.1.2. UniProt – SwissProt

Together with TrEMBL, SwissProt is part of the universal protein database. “[It] is the manually annotated and reviewed section of the UniProt Knowledgebase (UniProtKB). It is a high quality annotated and non-redundant protein sequence database, which brings together experimental results, computed features and scientific conclusions” [ExPA]. Furthermore, it is a hybrid database with entries in form of XML files stored across the database and a file system. The entries are downloadable in text and FASTA and few more data formats.

2.1.3. RCSB – PDB

The Research Collaboratory for Structural Bioinformatics (RCSB) prosecutes the Protein Data Base (PDB) with 3D structures from biomolecules. The service is public, cost-free, and the data is updated every week [RCSB]. Data can be submitted by scientists and the organization itself. The structures are stored in a crystallographic database, which is a relational database. Although it is complicated to store structure information as FASTA file, it is available, as well as many variations of their own PDB format, and an XML structure, called ‘PDBML’.

2.1.4. EMBL – ENA

The European Nucleotide Archive (ENA) from the European Molecular Biology Laboratory (EMBL) and the European Bioinformatics Institute (EBI) "captures and presents information relating to experimental workflows that are based around nucleotide sequencing" [ENAh]. It is composed of three main databases, each with own data formats and standards. Their data model covers experiment input, output machine data and interpretations. The input data arises from various sources, but predominantly from their institutes and their partners. Money from grants are used to provide the services for free to the public.

2.1.5. Comparison of Data Bases

The data bases provide different data models and data formats (compare Table 1).

	Name	*Content*	*Data model*	*Exportable as…*
NCBI	GenBank	Nucleotide Sequence	Centralized database (primary)	GenBank, GenBank(Full), **FASTA**, ASN.1, **XML**, INSDseq XML, TinySeq XML, Feature Table, Accession List, GI List, GFF3
UniProt	UniProtKB/ SwissProt	Protein Sequence	Centralized hybrid XML model (secondary)	Text, **FASTA** (canonical), **XML**, RDF/XML, GFF
RCSB	PDB - Protein Data Bank	Protein Structure	Relational database (primary)	**FASTA**, PDB Format, PDB Format (gz), PDBML/**XML** Format (gz), Structure Factors (CIF), Structure Factors (CIF – gz), Biological Assembly 1 (PDB Format - gz) (A), Assembly 2 (PDB Format – gz) (S)
EMBL-EBI	ENA - European Nucleotide Archive	Nucleotide Sequence	Federated database with three levels (primary)	Text, **FASTA**, **XML**

Table 1. Common Database Comparison.

They all try to fulfill the requirements of free access and data quality at the same time. They provide platforms for sharing and integrating data for comprehensive analysis purposes. They contain similar structured entries with descriptions and are readable for humans [Groß14].

Three of these listed services are marked as ‘primary’. This means they display directly the experimental data with hardly curation and annotation. Only one of them, SwissProt, is ‘secondary’ with semi-structured annotations and links [Groß14]. Therefore, SwissProt is used for integration in the practical work.

What can be recognized by comparing the rows in the table is that they make their data exportable in various different formats, but there are some formats through all of them, namely FASTA and XML.

2.2. Data formats

In the area of biology many different data formats are common. In this thesis the focus is on the most promising ones for a practical data engineering example.

2.2.1. XML

Extensible Markup Language (XML) can be used as a language basis for data exchange and integration. As seen in the previous chapter about common bio-data bases, XML is an export format for all of these four major data providers. By having the data in the same data format, it becomes easier to create mappings. XML is text based and therefore machine and human readable. The data for the subsequent continuous engineering example will be stored in a relational database and exported as XML file. The general syntax rules specify that data is represented by elements and their attributes, while the schema information is represented by tags, which are surrounded by ‘<’ and ‘>’ characters. There is a start tag and an end tag, in between is the represented data. This concept is called a ‘meta language’. The word ‘extensible’ indicates that it is an extensive concept [Pohj08]. There exist pre-defined tags, but the user can create his own vocabulary.
Each document must start with exactly one root element, which boxes the rest of the file. Then nested within the root, all elements are declared. Elements can contain further elements and attributes.

All opened tags need to be closed and all inner tags need to be closed before the outer one is closed. The tags can be named and defined freely, they describe the structure and the semantics of their content. XML is a structural language, tags can be nested arbitrarily. There can be semi-structured files with structured and unstructured parts.

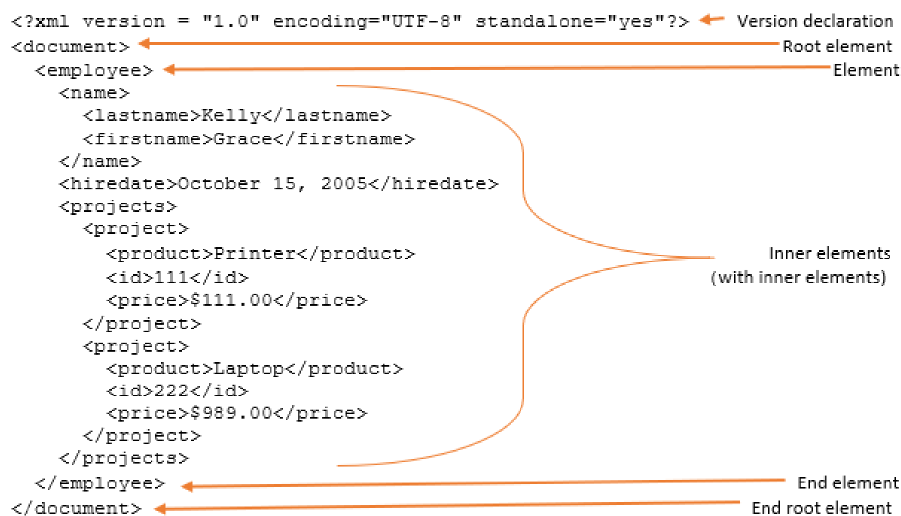
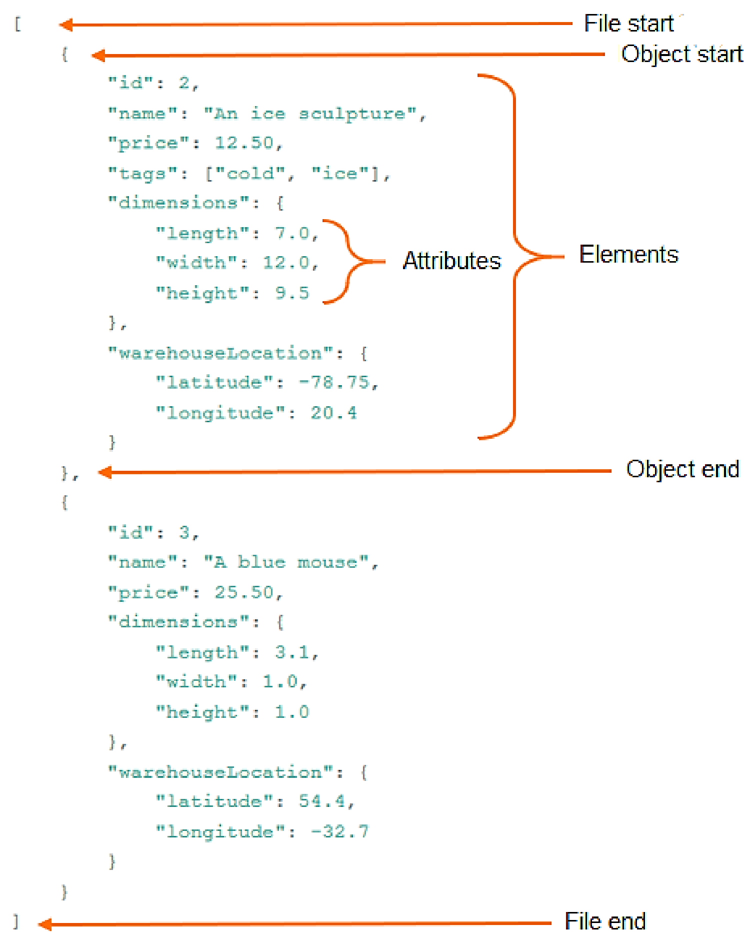


Fig. 2. Sample Fragment from a Well-formed XML Document [Holz03].

An XML document may be checked for well-formedness and validity by an XML processor. The document is well-formed, if all syntax rules are satisfied. It is valid, if it is well-formed and conforms to a particular schema provided by a schema document (DTD or XML schema). These files determine the names and structure of elements and attributes [Pohj08]. A 'Document Type Definition' (.dtd) ensures the quality of the document according to structure and syntax rules. [Siko14]. Nowadays XML Schemata (.xsd) replace DTDs mainly [LeNa07]. They define, for instance, elements and their attributes, data types, keys and structure. The advantages of XML schemata over DTDs are that they have XML as syntax, which does not require the knowledge of another language. Further, inheritance is possible, and the key concept is more flexible. Furthermore, there exist several pre-defined datatypes and the user can define simple and complex datatypes himself. The XML files can then be validated against these files and checked for their correctness.

2.2.2. JSON

Another format that can be used to represent biological data, is called JavaScript Object Notation (JSON). It uses the characters '{' and '}' to surround concepts and represents each kind of data as objects with properties and relations. The properties of an object can be elements and attributes. All objects in a file are surrounded by '[' and ']' characters.




```
[
  {
    "id": 2,
    "name": "An ice sculpture",
    "price": 12.50,
    "tags": ["cold", "ice"],
    "dimensions": {
        "length": 7.0,
        "width": 12.0,
        "height": 9.5
    },
    "warehouseLocation": {
        "latitude": -78.75,
        "longitude": 20.4
    }
  },
  {
    "id": 3,
    "name": "A blue mouse",
    "price": 25.50,
    "dimensions": {
        "length": 3.1,
        "width": 1.0,
        "height": 1.0
    },
    "warehouseLocation": {
        "latitude": 54.4,
        "longitude": -32.7
    }
  }
]
```

Fig. 3. Sample Fragment from a Valid JSON Document [JSON].

There are also schemata for JSON files. A JSON schema starts with so-called ‘keywords’ like the schema definition, title and description. Then, an object is started, and the properties are defined.
A property has a name, a datatype and can have attributes or a defined number of occurrences. At the end of an object all required properties, which cannot be null, are listed.
In the practical example a JSON schema is mapped to the XML schema, to demonstrate how to overcome heterogeneities between data formats and schemata.

2.2.3. FASTA

FASTA is a very often used export format in the area of biological data. It is a simple text-based format, usable for storing and displaying DNA sequences.

2.2.4. Relational Database Model

A relational database consists of tables with columns, rows and relations between tables. Each table needs a primary key, which means, it cannot be NULL and must be unique. Primary keys can be referenced from other tables by means of foreign key constraints. It can be necessary to store the key and the foreign key in some additional columns, if no existing one is suited.

To create and fill the database, the structured query language (SQL) can be used. It is designed to manage data in a relational database. For creating a relational table, the ‘CREATE TABLE’ statement is used, while for insertion there is a statement ‘INSERT TABLE’. The data from the practical work will be stored within a relational database.

2.2.5. Flat File Format

A flat file database is a collection of data, storing tables and records. Relations are only possible as references to other data sources. It is possible that all information is stored in one single table [FFDB].
An example for such a flat file database would be Prosite [Pros]. It stores entries with sequence patterns as a flat file. The single entries are nevertheless structured by a two-letter line code at the beginning of each line. This code defines the kind of information stored in this line. Entries are separated by '//'. A file from Prosite will be used for integration in the practical part.

3. Integration of Biological Data

To combine, analyze, and interpret data from more than one data source, it is important to integrate the required data sources. Integration can mean to represent information from overlapping systems free from redundancy (as far as possible) [LeNa07] or to integrate through data exchange. It is a necessary step in biological research, because many data make more sense in context. It allows access to several data bases and to find coherences or patterns, which provide conclusions about the investigated data sets. It is common that the answers for more complex queries are spread over several databases, which makes the performance of such a query challenging [BeLT16]. Furthermore, integration is "essential to ensure the reproducibility of the analysis and interpretation of the experimental findings" [LSJV15]. Without integration and cross-analyzation the collecting of biological data in large amounts might not lead to great results. Although the concept of integration is unavoidable for bio-databases, there are a lot of challenges, which have to be dealt with.

3.1. Challenges

There are three main challenges to cope with, namely redundancy, heterogeneity, and performance.

- Redundancy

Integration should be free from redundancies, which could occur from samples which are recognized in more than one of the sources. Because of the variety of different databases, very often sequences or fragments of sequence(s) are listed multiple times. If sources with identical or overlapping content are integrated, this may lead to redundancies and therefore further problems. A “common situation is that the same patient is associated with multiple entries in different systems […] but without any possibility of linking the data sets” [TJTM14]. This can even lead to “missed opportunities for discovery, diagnosis, or treatment” in the worst case [TJTM14].

- Heterogeneity

“One of the major challenges in the integration task is to address the heterogeneity of data” [PBJM15]. Data sources vary on many levels. Firstly, they can have diversity of syntax and data models. This means data can be stored as several data formats, for example as the relational model or the XML data format or simply as text format [LeNa07]. All data must be somehow transformed to a unified model. Then, there can be differences in the structure of the schemata, which can be solved by correct mapping. In addition, there can be discrepancies in semantics, i.e., “subject to different interpretations depending on the local context used” [LSJV15]. This can lead to misunderstandings, and not mapping same concepts because of naming interpretations. These are dangerous heterogeneities.
What also can become a problem, is the evolution of schemata and models. If, for instance, the structure of one of the integrated sources is changed, a query might fail.

- Performance

“The unanticipated need for additional performance and capacity is one of the most common challenges to data integration” [USDT10]. The term performance describes the time effort of processing a data framework. If a huge amount of new data is added to an environment, hardware and software may reach their limit.

3.2. Approaches

There are several common approaches for integration of multiple data sources, which try to avoid or decrease the mentioned problems. The integration can be done automated or manually. Often semi-automatic integration takes place. Some web services offer a kind of ‘search engine’ through various databases. They translate the queries to the required query-language and their results to a human readable format. Some provide an interface, where queries can be composed by clicking on diverse elements, to simplify the task for biologists, who are not familiar with the query languages.
Automatic processes are not perfect and can in some cases not fulfill the task satisfying. Often ‘semi’-automatic approaches are used, because refinement and corrections have to be done manually. Sometimes, manual integration is indispensable. There the integration is done based on the knowledge of the author, who creates a global schema and maps the concepts of different sources [LSJV15]. The author is responsible that redundancy is avoided on the level of the schema as well as on the level of the actual data [LeNa07]. To get a fitting and error free integration for a certain project, “it is particularly important for the biological research community to get acquainted with the conceptual basis of data integration, its limitations, challenges and actual terminology” [LSJV15].

Most bio-databases are connected only through links, linking objects from different sources by references. Most of the more than 500 public data sources are connected via hyperlinks to other bio-databases, without using a global schema [Rahm04].

As can be seen, there are different integration approaches applicable in the area of biologic databases. Some basic strategies for integrating data are explained in the following.

- Virtual Integration

Utilizing virtual integration, a universal global schema is possible, but not mandatory. One popular representative for virtual integration is a 'federated database', which has a global schema and defines mappings between elements from different local datasources. Queries to the global schema are translated into queries to the local schemata [LeNa07]. This concept can be found at the European Bioinformatics Institute (EBI), for instance.

The following graphic represents the connections between various databases from EMBL-EBI. The outside circle is indexed with the different resources. The arcs between the resources stand with proportional width for the number of interactions [CBFC16]. For all other data resources like UniProt, RCSB or NCBI, a similar graphic could be created, with a large number of links between the single databases.

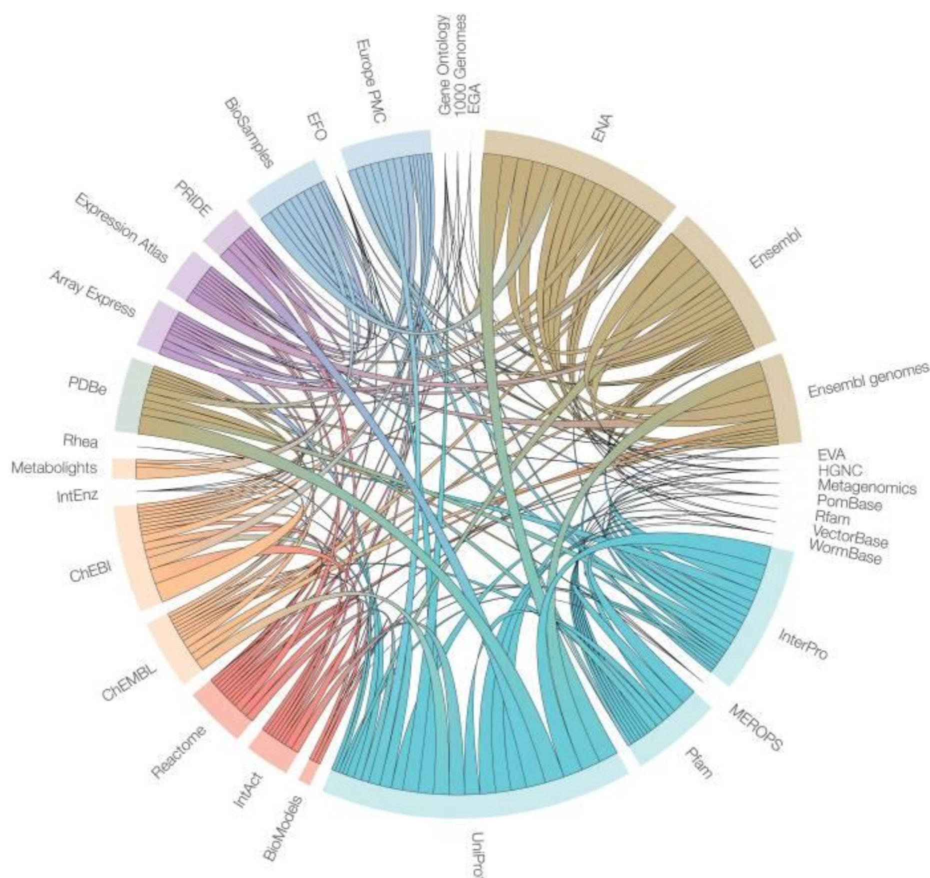


Fig. 4. Virtual Integration at EMBL-EBI [CBFC16].

- Materialized Integration

Utilizing materialized integration, the data from the distinct sources are replicated and stored together in one system, e.g., within a central relational database. A global schema can be used, but is not required. This approach is "increasingly used in the biological field because it is extremely well adapted to some needs of domain (confidentiality, treatment control, full data cleansing)" [LSJV15]. The disadvantages of this form of integration are the effort to keep the data updated, the difficulty of queries, and the increased storage mass.

- Ontologies

One helpful approach for improving and automating the step of data integration can be the usage of ontologies. The principle of ontologies is to provide a taxonomy of terms with relationships in between like synonymity, is-a, and part-of, representing similar causalities and their definitions. This enlarges the number of hits in a database query with high accuracy. Ontologies are machine understandable and were agreed between a group of researchers in the specific area. In biological databases "there is the need for consistent descriptions of gene products […], providing not only comprehensive coverage of biological concepts but also community-wide agreement on how those should be used to describe gene functions across all organisms" [Cons15].

Ontologies can help in the process of data integration by specifying how to overcome the heterogeneities between data sets. In a Relational-to-Ontology data integration the ontology "is 'connected' to databases with the help of mappings that are declarative specifications describing the relationship between the ontological vocabulary and the elements of the database schema" [PBJM15]. Through the mapping from an element in one data set to the ontology, it is at the same time connected to other terms with equivalent meaning, which are perhaps mapped to a similar element in another data set.

The Gene Ontology (GO) [GeOn] is one of the most popular ontologies for biologists, their effort - according to the consortium - consists of "the development and maintenance of the ontology, the annotation of gene products and the development and continuous improvement of tools and training that facilitate the creation, maintenance, and use of the ontologies" [Cons15]. It is published and updated regularly by the 'Gene Ontology Consortium' and contains more than 600.000 annotations (2017) [Cons17]. It has several branches from fields all over biology and is continuously growing. In the practical part a small extract from GO data is used to show a simple application of an ontology.

3.3. Example: KEGG

One important freely available integrated database, using internal integration with a good performance, is the Kyoto Encyclopedia of Genes and Genomes (KEGG) [KEGG], for instance. It is an “integrated database resource for biological interpretation of completely sequenced genomes” [KFTS17]. KEGG is a metabolic pathway database, which derives its data from 17 separated databases, containing metabolic, genomic, and chemical information. Metabolic pathways are all biochemical reactions occurring around and in an organism like respiration, digestion or fermentation. These 17 databases are connected by virtual integration, the contents are linked by references. The database ‘KEGG PATHWAY’ is the backbone of the whole system, containing pathway maps describing the ongoing biochemical processes. These maps integrate several information, as genes, proteins, diseases and drugs for instance, from entries in other KEGG databases. The data is reviewed and therefore non-redundant. Since all the databases come from the same authors, heterogeneities can be avoided or overcome while construction of the databases. One of the sub-databases worth mentioning is ‘KEGG BRITE’, which is an ontology not only for molecular interactions but also for various related topics. It is mapped to several of the 17 KEGG databases and builds a framework with all connections and relations between these databases and their entries. The other ones of the main important databases are ’KEGG Gene’ and ‘KEGG Ligands’. They store information about genomics and their chemical behavior [KGHA06].

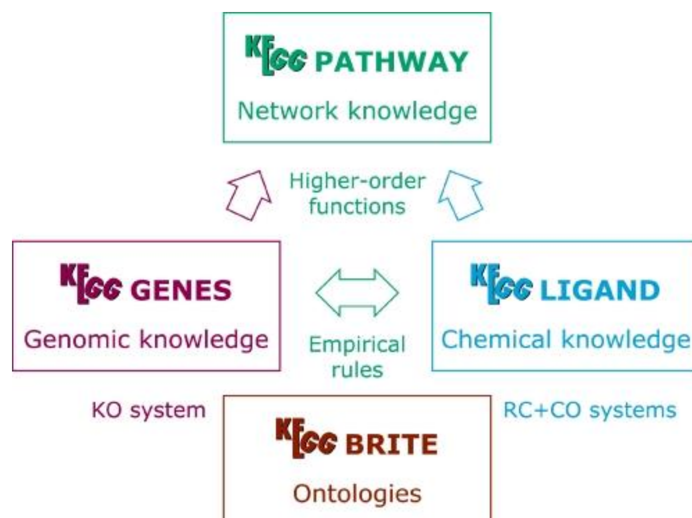


Fig. 5. KEGG Main Databases and their Relationships [KGHA06].

The chemical behavior is directly linked to the according genes by some defined rules. Genes and ligands both are expressed in the pathway maps and reached by forwarding to their detailed entries.
KEGGs mapping approach is well developed and allows an applicable integration of disease and drug data, even from sources outside of these KEGG internal databases [KGSF12].

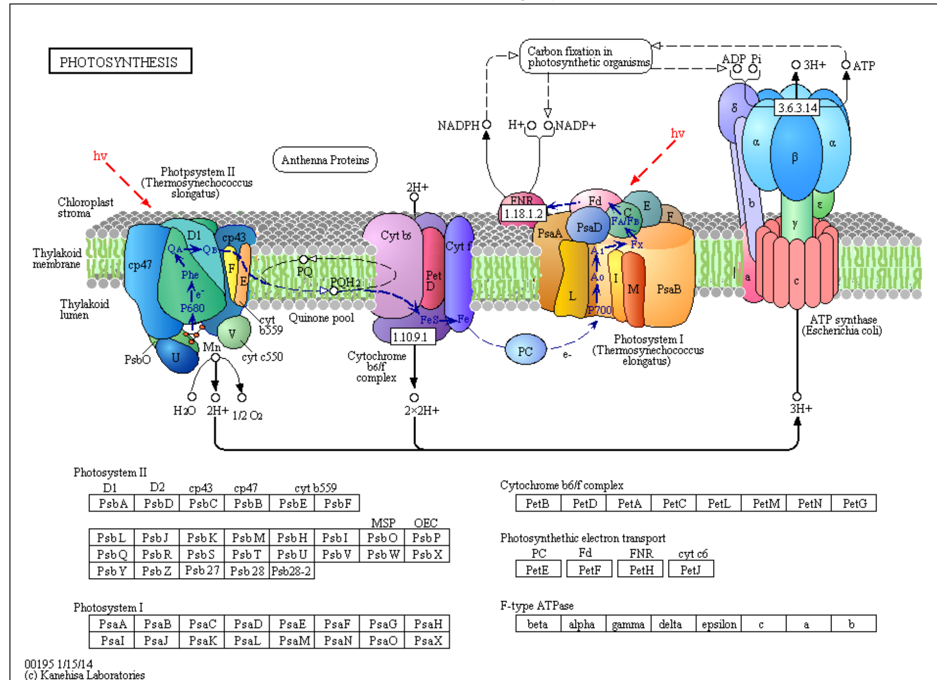


Fig. 6. Photosynthesis Pathway Map from KEGG.

Fig. 6. depicts a screenshot from the KEGG database showing the pathway map of the biochemical reactions during photosynthesis. By clicking on a compound, one can navigate to a different entry with detailed information.

4. Practical Work

4.1. Aims

The goal of the practical work is to construct a continuous example of engineered biological data from a real-world data source, resulting in the data itself, schemata in different formats, and queries, with the purpose to show the richness of information in a well-engineered integrated dataset. All kinds of engineering steps should be used to demonstrate the spectrum of opportunities. Further, the effort and the needed time is compared to an internet database search by using search functions and hyperlinks. Some further possible engineering steps, which cannot be observed in online database search, are stated and explained.

4.2. Comparison of Data Sources

As a first step, a data set is searched, which provides richness of information, which means several classes, various constraints and enough entries to construct interesting queries. Further, it should be easy to understand and not too complex for implementation. The data source should be used to demonstrate various steps in data engineering. Therefore, a data set providing the data format in XML format with approximately 10 classes and adequate number of XML entries would be appropriate. To support the next steps, it would be helpful, if there are different schemata and different export formats as well as a similar schema with heterogeneities, which can be mapped to the example with a reasonable number of correspondences. Five different online available data sets are described and the best suited one is figured out. The UML class diagrams for all data sources can be found in Appendix A.

4.2.1. Immunization

The first qualified data source is the Immunization data source, which provides an XML schema [Immu] for clinical data regarding immunization and vaccination with the according class diagram is online. The data can be exported as XML and as JSON format. There are three example entries. It has five classes with simple constraints.

The most important class is *Immunization*, it contains different information like the *patient*, the *vaccine manufacturer* and *date*, for example. It is connected to the class *Reaction* which yields *details* and *date* of the immune reaction. Furthermore, it is connected to the *VaccinationProtocol* which has more details about the *dose* and its *efficiency*. Another class is *Explanation* which contains reasons for the Immunization. The last class is *Practitioner* which contains the role of the practitioner in the course of the immunization.

A similar schema with just one class for mapping [ImmM] with approximately seven correspondences exists. This class contains information like *name*, *date*, *sequence*, and *manufacturer* of the immunization.

4.2.2. Allergy Intolerance

Another possible data source is the Allergy Intolerance data source, which provides an XML schema [AlIn] for allergies with the according class diagram containing two classes. The data can be exported as XML and as JSON format. There are three example entries available. The UML class diagram contains two classes with sub elements and a very simple constraint. Having two classes, only, is very small for the required purposes, since the construction of the class diagram is an important part.

The main class is *AllergyIntolerance* and it has the attributes *recorder*, *patient*, *reporter*, *type* and few more. It is connected to the class *Event*. Events cause intolerances and it contains information like *substance*, *duration* and *severity*.

The Allergy Intolerance schema could be mapped with another schema [AllM] having one class *allergy* with approximately the two correspondences *first-observed* and *allergen-type*.

4.2.3. Single Nucleotide Polymorphism

The next conceivable source is the Single Nucleotide Polymorphism data source, it provides an XML schema [SNPs] for single nucleotide polymorphism entries, which are single mutations in DNA sequences. The generated class diagram contains 31 classes. Unfortunately, there is just one sample entry. The other entries have to be created artificially and would not represent real scientific results.

One of the main classes is called *Identifiable* and is connected to a database reference class. Some attributes are *creation date*, *deletion date*, *name*, *source*, and similar. It is constituted from several subclasses like *location*, *description*, *methodology* and *taxon*. Another important class is *residue_change* which is another biological term for mutation. Additional attributes are *name* and *dates* and further has a *location* and the *new* and the *old residue*. The class is connected to the database reference class. Its subclasses are specific types of DNA fragments (oligo, exon and CDS). The location is constructed from different classes and different type of locations. Further there are the classes *Person*, *Contributor* and *Organization* but with no important role for the schema.

A similar schema for the mapping would have been one class from SNP schema documentation [SNPM] from the NCBI SNP database, namely class *Sequence*. But it is not possible to create a valid XML schema file for it. Therefore, there is no UML class diagram and no possibility for correspondences.

4.2.4. Human Protein Atlas

Another alternative is the Human Protein Atlas data source. The human protein atlas (HPA) is a database containing all proteins that occur in humans. Amongst others, information about cell, tissue, and pathology are stored and displayed graphically.

Its schema [HPAs] has 66 classes, which is really complex. But most of these classes represent complex types and the main classes are comprehensible. A lot of entries are downloadable, and some subset of the classes has to be chosen in order to get a reasonable example.

The root of the schema is the class *proteinAtlas* that contains the attributes *entry* and a *copyright*. The class for the *entry-type* consist of a lot of elements, for example *name*, *protein*, *tissue*, and *antibody*. All of them again have complex types containing many elements. Further classes, worth mentioning, are *patient*, which has *sex*, *age*, an ID and a *sample* as attribute. Such a sample can be the result of several analyses with some intermediate steps. The *analyses*, *images*, *dilution*, *arrays* or *staining*, for instance, are also complex types with many connections.

The XML schema for mapping could be the UniProt schema [UniP] having 52 classes. The UniProt schema contains also protein entries. There are approximately 13 correspondences between the schemata.

The schema begins with class *uniprot* which can reference a copyright and references any number of *entries*. Class *entry* stores basic information like *date*, *version*, and *accession*, but also biological data like *gene*, *organism*, *protein* and *sequence*. They all have types with further sub elements. There are also classes for *location*, *database reference*, and *disease*. All classes representing *names*, *comments* or other strings have very strict rules for the composition of the string.

4.2.5. Cancer Gene Disease

The last investigated data source is the Cancer Gene Disease Data Source. The 2017 newly created and detailed documented Cancer Gene Disease database [CaGD] provides a DTD schema file as well as entries. The XML file contains 7181 entries, from which a subset can be taken. Ignoring the types there are eight classes, defining the structure of an entry representing a gene associated with cancer.

The root element GeneEntry contains a minimum of one GeneAlias instance, which is another common name for the gene, and gene accession numbers from several other databases. It can have any number of sentences which have information regarding the organism, a gene term and a disease term. Further it can contain a statement and comments. It is related to the role of the gene in this disease. There are primary roles as well as other roles.

4.2.6. Comparison of Data Sources

Again, mapping to the Uniprot schema [UniP] is possible, having approximately ten correspondences.

		Immunization	Allergy Intolerance	Single Nucleotide Polymorphism	Human Protein Atlas	Cancer Index
Documentation	✓	✓	✓	✗	~	✓
Ontology	✗	✗	✗	SNPO	PR	NCIT
Exportable formats						
XML Files	4	3	3	1	9	1
Nr of XML Entries	554	3	3	1	>12,000	7181
JSON Files	✗	3	3	✗	✗	✗
Available schemas						
XSD Schema	✓	✓	✓	✓	✓	✓G
DTD Schema	✓	✓G	✓G	✓G	✓G	✓
JSON Schema	✗	✓	✓	✗	✗	✗
Year		?	?	2008	2003	2017
UML						
Class Diagram	✓	✓	✓	✓G	✓G	✓G
Nr of Classes	9	5	2	31	66	8
Mapping						
Schema		Immunization	Allergy	NCBI SNP	Uniprot	Uniprot
Nr of Correspondences	36	7	2	✗	13	10

G: generated

Table 2. Comparison of Data Sources.

Table 2 gives a quick overview about the five data sets. The availability of certain properties or features is marked by a check sign. The capital G indicates that this feature was generated by some program and was not available in the original data source.

Firstly, the Single Nucleotide Polymorphism dataset was excluded, because there is no good schema to be mapped to, it has just one entry, and the remaining entries would need to be created. Another small minus is that there is no documentation explaining the classes, which could be very helpful. Then, the Allergy Intolerance was excluded, since its schema has just two classes, what would be too simple for the example, also it has not enough XML files. Both, Human Protein Atlas and Cancer Gene Disease were preferable over Immunization, since Immunization has just three XML files and five classes. From the Human Protein Atlas, a subset from the 66 classes would have to be taken. Interesting parts would be *entry*, *patient*, *antibody*, *protein*, *western-blot*, *subcellData*, *tissue*, and *location*. From the very many XML files various subsets could be taken and used with according queries. The number of classes in Cancer Gene Disease can be taken as it is. From the more than 7000 entries a subset can be taken. One further advantage over HPA is the detailed documentation about the elements in the DTD schema. For both, there might be a biological ontology related to their topics. Finally, Cancer Gene Disease was chosen for the example, because the content of the example and the queries are more interesting.

4.3. Example – Cancer Gene Disease

To use the content of the Cancer Gene Disease data source for an analysis, further processing steps are necessary. All entries of the data set are stored in one file. As a first step the structure of the data needs to be investigated and represented as UML class diagram. Based on the knowledge about the structure, a relational schema can be created. Then a database, fitting this schema, has to be constructed and the data has to be inserted. In case, that the set does not fulfill all requirements needed for a relational database, the data has to be changed before. As soon as all data is entered, queries can be performed. An online database for some query is performed to be able to compare the two approaches. Further possibilities for an engineered database are to map to heterogeneous schemata and to integrate various data sources like SwissProt [SwPr], Prosite [Pros], and the Gene Ontology [GeOn].

4.3.1. UML Class Diagram

The Unified Modeling Language (UML) is a common modelling language for constructing and describing software artefacts. It offers graphical notation elements to represent diagrams in a unified way. Therefore, it is understandable and interpretable for others - not only data engineering specialists. It is also useful for documentation, to answer later questions regarding the architecture. By having a class diagram as common basis for all other subsequent steps, errors can be reduced and time saved.

The data from the Cancer Gene Disease dataset is missing some important attributes and constraints for the following engineering steps. Therefore, some optimizations have to be done. Following is the description and representation of the optimized class diagram. The according UML class diagram is depicted in Fig. 7.

Class *GeneEntry* contains all information of one entry and has two mandatory attributes *HUGOGeneSymbol* (that is a standardized name acronym by the Human Genome Organization) and *geneStatus* (the status shows the progress of the research of this entry, the corresponding enumeration comprises new, finished, withdrawn, and entry_withdrawn). It has five non-mandatory attributes. These are all accession numbers from other more or less known databases (*hgnc*, *LocusLink*, *genbank*, *refSeq* and *uniProt*). *GeneEntry* is related to any number of *GeneAlias* (which are other common names for this gene) and to none, one or several instances from class *Sentence*. Class *Sentence* contains more detailed information about the gene described by the entry plus the according disease and has ten attributes, namely a unique *sentenceID*, *pubMedID* (which shows an entry number for a related PubMed entry), compulsory *organism* (in which the gene was found), *negationIndicator* (Boolean, whether the findings of a gene-disease association within a sentence were negative), *cellLineIndicator* (Boolean, whether the data was collected from a cellline, a cellline are cells derived from one cell in laboratory), and necessarily *sentenceStatus* (signals the status of the Sentence, in the enumeration there are finished, no_fact, unclear, and redundant). Further, it contains a mandatory *matchedGeneTerm* (term matched to the gene from the NCI Thesaurus ontology) and an optional *NCIGeneConceptCode* (a digit code assigned by the NCI research team for this gene concept, namely gene-disease-pair), a compulsory *matchedDiseaseTerm* (a term for a disease connected to the gene, e.g., tumor found by their ontology) and an obligatory *NCIDiseaseConceptCode* (a digit code for this disease assigned by the NCI research team), as well as none, one or several instances from the class *Statement*, *Roles* (which has only an ID), *EvidenceCode* and *Comments*. *PrimaryNCIRoleCode* and *OtherRole* are parts of the class *Roles*. Both have an ID and *PrimaryNCIRoleCode*, states the relationship from the gene to the disease as a string (e.g. “Gene_Associated_ With_Disease”) while *OtherRole* describes further functions not related to the disease.

`Statement` is a summarizing text added by the researchers from National Cancer Institute. `EvidenceCode` is a string (referencing to another publication of the gene, to prove its existence) and `Comments` are some additional remarks by the researchers.

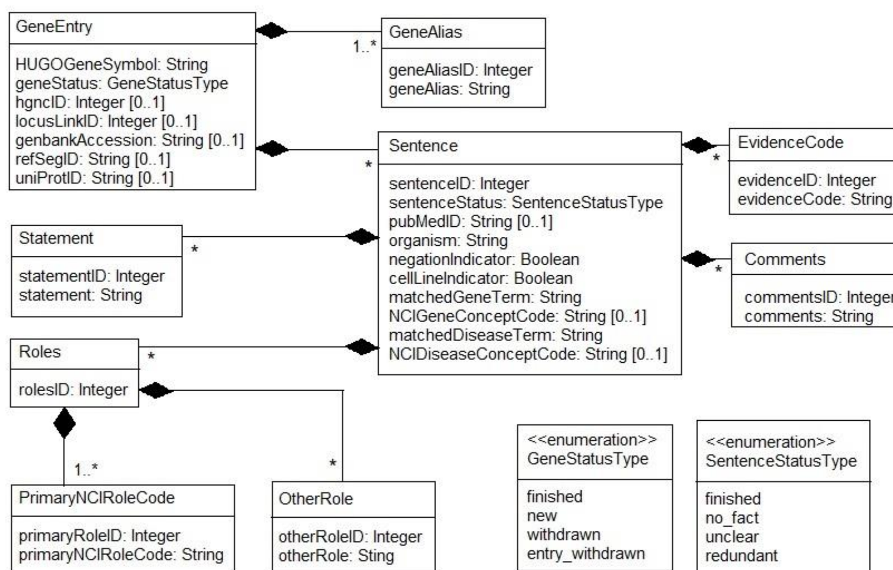


Fig. 7. Cancer Gene Disease UML.

4.3.2. Data Set and XPath Queries

The original data set is one XML file with 7181 entries from the National Cancer Institute (NCI) [CaDI]. One advantage of an XML file is that the document is readable for humans and for the computer. Because it is plain text, no special tools are needed to view it and it can be modified by any text editor. This makes the understanding and modification quick and easy. It is layout independent, the structure is defined by the document itself and its meta data. Then, as mentioned, it is extensible and has no restrictions for new areas of interest. [Cera06]. Another benefit of XML is the possible validation by DTDs or XML schemata. These external files can check if all rules were kept. This avoids spending time with debugging and provides though a valid document.

JKU JOHANNES KEPLER UNIVERSITY LINZ

In order to have the dataset according to all our requirements some optimizations were necessary:

- Some classes need to be renamed to be more plausible.
- Elements with more than one instance become an own class, while classes with 1:1 relation are merged.
- For some elements the cardinalities differ from the original document.
- The elements `NegationIndicator` and `CelllineIndicator` are changed to datatype Boolean, because their original values are "yes" and "no", which is more complicated to use in later queries.
- The datatype from `SentenceStatus` and `GeneStatus` is changed to an enumeration because all entries have one of the four enumerated values.
- In each class a surrogate ID is introduced which can be used as primary key later while construction of the database. Only in the class `GeneEntry` it is possible to use the attribute `HUGOGeneSymbol` as primary key because it cannot be null and is unique.
- For reasonable queries, a subset with only 55 of the entries is created.

```
<GeneEntry HUGOGeneSymbol="PRKX">
              <geneStatus>finished</geneStatus>
              <hgncID>9441</hgncID>
              <locusLinkID>5613</locusLinkID>
              <genbankAccession/>
              <refSeqID>NM_005044</refSeqID>
              <uniProtID>P51817</uniProtID>
              <GeneAlias geneAliasID="456">pkx1</GeneAlias>
              <GeneAlias geneAliasID="457">pkxi</GeneAlias>
              <GeneAlias geneAliasID="458">protein kinase,
              x-linked</GeneAlias>
              <GeneAlias geneAliasID="459">prkx</GeneAlias>
              <GeneAlias geneAliasID="460">pkx-i</GeneAlias>
              <GeneAlias geneAliasID="461">pkx 1</GeneAlias>
              <GeneAlias geneAliasID="462">pkx-1</GeneAlias>
       </GeneEntry>
```

Listing 1. Cancer Gene Disease Example Entry.

JOHANNES KEPLER UNIVERSITY LINZ

The Listing 1 shows the entry for gene PRKX, for example, it has the `HUGOGeneSymbol` "PRKX", four alternative names, four accession numbers from other databases and zero sentences. The status is finished.

- XPath

A typical language for querying such XML documents is XML Path Language (XPath) from the W3C [W3Ch]. XPath is a language that allows the retrieval of elements, attributes, and values from XML documents. In the following XPATH expression are used to find the answer to some queries.

Query:
"Select the overall number of gene entries."

Statement:

```
count(GeneEntryCollection/GeneEntry)
```

Result: 55

Query:
"Select all grandchildren of the context node."

Statement:

```
*/*
```

Result:
Resulting are all 55 entries with their `HUGOGeneSymbol`.

Query:
"Select the `GeneAlias` elements of `GeneEntry`."

Statement:

```
GeneEntryCollection/GeneEntry/GeneAlias
```

Result:
As a result, all 742 *`GeneAliases`* are shown.

Query:

"*Select the organisms from all `Sentences`.*"

Statement:

```
GeneEntryCollection/GeneEntry/Sentence/organism
```

Result:

The result is the *`organism`* from each entry.

Query:

"You want to finish an old entry. Select those gene entries which have the status 'withdrawn'."

Statement:

```
GeneEntryCollection/GeneEntry[geneStatus="withdrawn"]
```

Result:

There is just one withdrawn entry and its *`HUGOGeneSymbol`* is TP53L.

Query:

"Select the first Statement for the 24th gene entry."

Statement:

```
GeneEntryCollection/GeneEntry[24]/Sentence[1]
```

Result:

That is the `sentence` with the ID 66.

Query:

"Select all gene entries, which have exactly five sentences as child elements."

Statement:

```
GeneEntryCollection/GeneEntry[count(Sentence)=5]
```

Result:

There are eight entries with exactly five sentences, namely C4BPA, EED, NEURL, HOXB13, MN1, RBM17, FAM107A and UBC.

Query:

"You want to read more about a specific disease. Select the child element 'pubMedID' from those Sentence elements where the 'matchedDiseaseTerm' is "tumors"."

Statement:

```
GeneEntryCollection/GeneEntry/Sentence[matchedDiseaseTerm
='tumors']/pubMedID
```

Result:

There are ten sentences with disease term 'tumors' and as a result are the PubMed IDs displayed.

Query:

"Select the 3rd gene entry element if it has a child attribute `Sentence` with a cellLineIndicator 'false'."

Statement:

```
GeneEntryCollection/GeneEntry[3, /Sentence/cellLineIndicator eq 'false']
```

Result:

The ***`HUGOGeneSymbol`*** of the 3rd entry is DLEU7.

4.3.3. Schemata – DTD and XML Schema

As mentioned, DTD and XML Schema are schemata used to validate XML files. To validate the XML file from the example, a schema is required.

- DTD

In the Cancer Gene Disease DTD, the root element is called *GeneEntryCollection* and it contains at least one instance from element *GeneEntry*. A *GeneEntry* is composed of one *geneStatus*, optionally five IDs from different databases, a minimum of one element *GeneAlias* and zero or many instances of element *Sentence*. The *HUGOGeneSymbol* is a required ID, defined as attribute. The element *Sentence* is composed of one *sentenceStatus*, optionally one *pubMedID*, one *organism*, one *negationIndicator*, one *cellLineIndicator*, one *matchedGeneTerm*, one *NCIGeneConceptCode*, one *matchedDiseaseTerm* and one *NCIDiseaseConceptCode*. It references an arbitrary number of elements from class *Statement*, *Roles*, *EvidenceCode* and *Comments*. Its ID is called *sentenceID*. Element *Roles* contains a minimum of one instance *PrimaryNCIRoleCode* and any desired number of instances *OtherRole*. All of them have IDs, declared as attributes.

```
<!-- DTD for Cancer Gene Disease -->
<!ELEMENT GeneEntryCollection (GeneEntry+)>

<!ELEMENT GeneEntry (hgncID?, locusLinkID?, genbankAccession?, ref-
SeqID?, uniProtID?, GeneAlias+, Sentence*)>
<!ATTLIST GeneEntry HUGOGeneSymbol ID #REQUIRED>
<!ATTLIST GeneEntry geneStatus (finished|new|withdrawn|
entry_withdrawn) "new">

<!ELEMENT hgncID (#PCDATA)>
<!ELEMENT locusLinkID (#PCDATA)>
<!ELEMENT genbankAccession (#PCDATA)>
<!ELEMENT refSeqID (#PCDATA)>
<!ELEMENT uniProtID (#PCDATA)>

<!ELEMENT GeneAlias (#PCDATA)>
<!ATTLIST GeneAlias geneAliasID ID #REQUIRED>
```

JOHANNES KEPLER UNIVERSITY LINZ

```
<!ELEMENT Sentence (pubMedID?, organism, matchedGeneTerm, NCIGene-
ConceptCode?, matchedDiseaseTerm, NCIDiseaseConceptCode?, Statement*,
Roles*, EvidenceCode*, Comments*)>
<!ATTLIST Sentence sentenceID ID #REQUIRED>
<!ATTLIST Sentence sentenceStatus (fin-
ished|no_fact|unclear|redundant) "no_fact">
<!ATTLIST Sentence negationIndicator (true|false) "false">
<!ATTLIST Sentence cellLineIndicator (true|false) "false">
<!ELEMENT pubMedID (#PCDATA)>
<!ELEMENT organism (#PCDATA)>
<!ELEMENT matchedGeneTerm (#PCDATA)>
<!ELEMENT NCIGeneConceptCode (#PCDATA)>
<!ELEMENT matchedDiseaseTerm (#PCDATA)>
<!ELEMENT NCIDiseaseConceptCode (#PCDATA)>

<!ELEMENT Statement (#PCDATA)>
<!ATTLIST Statement statementID ID #REQUIRED>

<!ELEMENT Roles (PrimaryNCIRoleCode+, OtherRole*)>
<!ATTLIST Roles rolesID ID #REQUIRED>

<!ELEMENT PrimaryNCIRoleCode (#PCDATA)>
<!ATTLIST PrimaryNCIRoleCode primaryRoleID ID #REQUIRED>

<!ELEMENT OtherRole (#PCDATA)>
<!ATTLIST OtherRole otherRoleID ID #REQUIRED>

<!ELEMENT EvidenceCode (#PCDATA)>
<!ATTLIST EvidenceCode evidenceID ID #REQUIRED>

<!ELEMENT Comments (#PCDATA)>
<!ATTLIST Comments commentsID ID #REQUIRED>
```

Listing 2. DTD Schema for Cancer Gene Disease Example.

JOHANNES KEPLER
UNIVERSITY LINZ

- XML Schema

In the Cancer Gene Disease XML schema, the root element *GeneEntryCollection* contains a minimum of one reference to *GeneEntry*. Each *GeneEntry* has in a complex type construct mandatory one string *HUGOGeneSymbol* and one *geneStatus*, optionally it has *hgncID* and *locusLinkID* as integer and *genbankAccession*, *refSeqID* and *uniProtID* as string. Then, it references a minimum of one instance from *GeneAlias* and any number of instances from class *Sentence*. *GeneStatus* is a simple type *geneStatusType* and is an enumeration with four possibilities (finished, new, withdrawn, entry_withdrawn). The element *GeneAlias* is of type string. The element *Sentence* is composed of exactly one *sentenceStatus* and one *organism* as string. Optionally, there can be one integer *pubMedID*. Furthermore, *Sentence* contains two booleans *negationIndicator* and *cellLineIndicator* and mandatory elements *matchedGeneTerm* and *matchedDiseaseTerm* of type string and the optional strings *NCIGeneConceptCode* and *NCIDiseaseConceptCode*. *Sentence* can contain any number of elements from classes *Statement*, *Roles*, *Comments*, and *EvidenceCode*.

The simple type for *sentenceStatus* is called *sentenceStatusType* and is an enumeration with four possibilities (finished, no_fact, unclear, redundant). Element *Roles* has a minimum of one sub element *PrimaryNCIRoleCode* of type string and optionally sub elements from *OtherRole*.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
xmlns:cgd="http://www.cis.jku.at/CancerGeneDisease"
targetNamespace="http://www.cis.jku.at/CancerGeneDisease"
elementFormDefault="qualified" attributeFormDefault="unqualified">
	<!-- - - - - - - Root element - - - - - - - - - -  -->
	<!-- - - - - Gene Entry Collection - - - - - - - - -->
	<element name="GeneEntryCollection"
	type="cgd:GeneEntryCollectionType">
	<!-- - - - - - - - - - KEYs - - - - - - - - - - - -  -->
		<key name="GeneEntry_key">
			<selector xpath="cgd:GeneEntry"/>
			<field xpath="@HUGOGeneSymbol"/>
```

```
        </key>
        <key name="GeneAlias_key">
            <selector xpath="cgd:GeneEntry/cgd:GeneAlias"/>
            <field xpath="@geneAliasID"/>
        </key>
        <key name="Sentence_key">
            <selector xpath="cgd:GeneEntry/cgd:Sentence"/>
            <field xpath="@sentenceID"/>
        </key>
        <key name="Statment_key">
            <selector xpath="cgd:GeneEntry/cgd:Sentence/
            cgd:Statement"/>
            <field xpath="@statementID"/>
        </key>
        <key name="Role_key">
            <selector xpath="cgd:GeneEntry/cgd:Sentence/
            cgd:Roles"/>
            <field xpath="@rolesID"/>
        </key>
        <key name="PrimaryNCIRoleCode_key">
            <selector xpath="cgd:GeneEntry/cgd:Sentence/
            cgd:Roles/cgd:PrimaryNCIRoleCode"/>
            <field xpath="@primaryRoleID"/>
        </key>
        <key name="OtherRole_key">
            <selector xpath="cgd:GeneEntry/cgd:Sentence/
            cgd:Roles/cgd:OtherRole"/>
            <field xpath="@otherRoleID"/>
        </key>
        <key name="EvidenceCode_key">
            <selector xpath="cgd:GeneEntry/cgd:Sentence/
            cgd:EvidenceCode"/>
            <field xpath="@evidenceID"/>
        </key>
        <key name="Comments_key">
            <selector xpath="cgd:GeneEntry/cgd:Sentence/
            cgd:Comments"/>
            <field xpath="@commentsID"/>
        </key>
    </element>
    <complexType name="GeneEntryCollectionType">
        <sequence>
            <element ref="cgd:GeneEntry"
            maxOccurs="unbounded"/>
        </sequence>
    </complexType>
    <!-- - - - - - - - - - - - Gene Entry - - - - - - - - - - - - -->
    <element name="GeneEntry" type="cgd:GeneEntryType"/>
    <complexType name="GeneEntryType">
        <sequence>
```

```
                <element name="geneStatus"
                type="cgd:GeneStatusType"/>
                <element name="hgncID" type="integer"
                minOccurs="0"/>
                <element name="locusLinkID" type="integer"
                minOccurs="0"/>
                <element name="genbankAccession" type="string"
                minOccurs="0"/>
                <element name="refSeqID" type="string"
                minOccurs="0"/>
                <element name="uniProtID" type="string"
                minOccurs="0"/>
                <element ref="cgd:GeneAlias"
                maxOccurs="unbounded"/>
                <element ref="cgd:Sentence" minOccurs="0"
                maxOccurs="unbounded"/>
        </sequence>
        <attribute name="HUGOGeneSymbol" type="string"
        use="required"/>
</complexType>
<simpleType name="GeneStatusType" final="restriction">
        <restriction base="string">
                <enumeration value="finished"/>
                <enumeration value="new"/>
                <enumeration value="withdrawn"/>
                <enumeration value="entry_withdrawn"/>
        </restriction>
</simpleType>
<!-- - - - - - - - - - Gene Alias - - - - - - - - - - - - -->
<element name="GeneAlias" type="cgd:GeneAliasType"/>
<complexType name="GeneAliasType">
        <simpleContent>
                <extension base="string">
                        <attribute name="geneAliasID"
                        type="integer" use="required"/>
                </extension>
        </simpleContent>
</complexType>
<!-- - - - - - - - - - - Sentence - - - - - - - - - - -  -->
<element name="Sentence" type="cgd:SentenceType"/>
<complexType name="SentenceType">
        <sequence>
                <element name="sentenceStatus"
                type="cgd:SentenceStatusType"/>
                <element name="pubMedID" type="integer"
                minOccurs="0"/>
                <element name="organism" type="string"/>
                <element name="negationIndicator"
                type="boolean"/>
                <element name="cellLineIndicator"
                type="boolean"/>
```

```
                <element name="matchedGeneTerm" type="string"/>
                <element name="NCIGeneConceptCode" type="string"
                minOccurs="0"/>
                <element name="matchedDiseaseTerm" type="string"
                minOccurs="1"/>
                <element name="NCIDiseaseConceptCode"
                type="string" minOccurs="0"/>
                <element ref="cgd:Statement" minOccurs="0"
                maxOccurs="unbounded"/>
                <element ref="cgd:Roles" minOccurs="0"
                maxOccurs="unbounded"/>
                <element ref="cgd:EvidenceCode" minOccurs="0"
                maxOccurs="unbounded"/>
                <element ref="cgd:Comments" minOccurs="0"
                maxOccurs="unbounded"/>
        </sequence>
        <attribute name="sentenceID" type="integer"
        use="required"/>
</complexType>
<simpleType name="SentenceStatusType" final="restriction">
        <restriction base="string">
                <enumeration value="finished"/>
                <enumeration value="no_fact"/>
                <enumeration value="unclear"/>
                <enumeration value="redundant"/>
        </restriction>
</simpleType>
<!-- - - - - - - - - - - - Roles - - - - - - - - - - - - - -->
<element name="Roles" type="cgd:RolesType"/>
<complexType name="RolesType">
        <sequence>
                <element ref="cgd:PrimaryNCIRoleCode"
                maxOccurs="unbounded"/>
                <element ref="cgd:OtherRole" minOccurs="0"
                maxOccurs="unbounded"/>
        </sequence>
        <attribute name="rolesID" type="integer"
        use="required"/>
</complexType>
<element name="PrimaryNCIRoleCode"
type="cgd:PrimaryNCIRoleCodeType"/>
<complexType name="PrimaryNCIRoleCodeType">
        <simpleContent>
                <extension base="string">
                        <attribute name="primaryRoleID"
                        type="integer" use="required"/>
                </extension>
        </simpleContent>
</complexType>
<element name="OtherRole" type="cgd:OtherRoleType"/>
<complexType name="OtherRoleType">
```

```
            <simpleContent>
                <extension base="string">
                    <attribute name="otherRoleID"
                    type="integer" use="required"/>
                </extension>
            </simpleContent>
        </complexType>
        <!-- - - - - - - - - - - - Statement - - - - - - - - - - -->
        <element name="Statement" type="cgd:StatementType"/>
        <complexType name="StatementType">
            <simpleContent>
                <extension base="string">
                    <attribute name="statementID"
                    type="integer" use="required"/>
                </extension>
            </simpleContent>
        </complexType>
        <!-- - - - - - - - - - - - Comments - - - - - - - - - - - -->
        <element name="Comments" type="cgd:CommentsType"/>
        <complexType name="CommentsType">
            <simpleContent>
                <extension base="string">
                    <attribute name="commentsID"
                    type="integer" use="required"/>
                </extension>
            </simpleContent>
        </complexType>
        <!-- - - - - - - - - - Evidence Code - - - - - - - - - - -->
        <element name="EvidenceCode" type="cgd:EvidenceCodeType"/>
        <complexType name="EvidenceCodeType">
            <simpleContent>
                <extension base="string">
                    <attribute name="evidenceID"
                    type="integer" use="required"/>
                </extension>
            </simpleContent>
        </complexType>
</schema>
```

Listing 3. XML Schema for Cancer Gene Disease Example.

4.3.4. Database Creation and Queries

For further steps, a relational database is realized. To ensure to have unique keys, surrogate IDs were added to almost all classes, as already mentioned earlier.

The first class to implement is `GeneEnty`, it has one primary key and six further attributes.

Statement:

```
CREATE TABLE GeneEntry (
  HUGOGeneSymbol VARCHAR2(10),
  geneStatus VARCHAR2(10) NOT NULL CHECK (GeneStatus
  IN('finished','new','withdrawn','entry_withdrawn')),
  hgncID NUMBER(20),
  locusLinkID NUMBER(20),
  genbankAccession VARCHAR2(20),
  refSeqID VARCHAR2(20),
  uniProtID VARCHAR2(20),
  CONSTRAINT PK_GeneEntry PRIMARY KEY (HUGOGeneSymbol)
);
```

The name of the table is written after the 'CREATE TABLE' command. The columns and constraints are defined in the outer parenthesis. The first column is the *HUGOGeneSymbol* which constitutes the primary key. The second column is *geneStatus*, which needs to be one value defined by the check constraint, realizing an enumeration. As already mentioned, the status of the entry can be finished, new, withdrawn or entry_withdrawn. The other five columns store accession numbers from different databases. Next, table *GeneAlias* is implemented.

Statement:

```
CREATE TABLE GeneAlias (
  geneAliasID NUMBER(5),
  geneAlias VARCHAR2(90) NOT NULL,
  HUGOGeneSymbol VARCHAR2(10) NOT NULL,
  CONSTRAINT PK_GeneAliases PRIMARY KEY (GeneAliasID),
  CONSTRAINT FK_Alias_Entry FOREIGN KEY (HUGOGeneSymbol)
  REFERENCES GeneEntry
);
```

The first column is called *geneAliasID* and is used as primary key. The column *HUGOGeneSymbol* realizes foreign keys and allows to reference values from the table *GeneEntry*, which was created before.

Statement:

```
CREATE TABLE Sentence (
  sentenceID NUMBER(5),
  pubMedID NUMBER(20),
  organism VARCHAR2(20) NOT NULL,
  negationIndicator VARCHAR(5) NOT NULL CHECK (NegationIndicato
  IN('true','false')),
  cellLineIndicator VARCHAR(5) NOT NULL CHECK (CelllineIndicator
  IN('true','false')),
  sentenceStatus VARCHAR(10) NOT NULL CHECK (SentenceStatus
  IN('finished','no_fact','unclear','redundant')),
  matchedGeneTerm VARCHAR2(50) NOT NULL,
  NCIGeneConceptCode VARCHAR2(20),
  matchedDiseaseTerm VARCHAR2(50) NOT NULL,
  NCIDiseaseConceptCode VARCHAR2(20),
  HUGOGeneSymbol VARCHAR2(10) NOT NULL,
  CONSTRAINT PK_Sentence PRIMARY KEY (sentenceID),
  CONSTRAINT FK_Sentence_Entry FOREIGN KEY (HUGOGeneSymbol)
  REFERENCES GeneEntry
);
```

In the table *Sentence* the *sentenceID* is the primary key. Again, a check constraint is used, containing the values true and false. This realizes the datatype boolean. For the status an enumeration is used again. The foreign key references attribute *HUGOGeneSymbol* from class *GeneEntry*. The next table is *Statement*.

Statement:

```
CREATE TABLE Statement (
  statementID NUMBER(5),
  statement VARCHAR2(530) NOT NULL,
  sentenceID NUMBER(5) NOT NULL,
  CONSTRAINT PK_Statement PRIMARY KEY (statementID),
  CONSTRAINT FK_Statement_Sentence FOREIGN KEY (sentenceID)
  REFERENCES Sentence
);
```

It has as usual an ID, a foreign key and contains a statement with 530 characters length. The next table is called *Roles*.

Statement:

```
CREATE TABLE Roles (
  rolesID NUMBER(5),
  sentenceID NUMBER(5) NOT NULL,
  CONSTRAINT PK_Role PRIMARY KEY (rolesID),
  CONSTRAINT FK_Role_Sentence FOREIGN KEY (sentenceID)
  REFERENCES Sentence
);
```

This table has only an ID and a foreign key, no other information. *PrimaryNCIRoleCode* references this class.

Statement:

```
CREATE TABLE PrimaryNCIRoleCode (
  primaryRoleID NUMBER(5),
  primaryNCIRoleCode VARCHAR2(100) NOT NULL,
  rolesID NUMBER(5) NOT NULL,
  CONSTRAINT PK_PrimaryRole PRIMARY KEY (primaryRoleID),
  CONSTRAINT FK_PrimaryRole_Sentence FOREIGN KEY (rolesID)
  REFERENCES Roles
);
```

The table *PrimaryNCIRoleCode* contains additionally to the primary key constraint the information about the *primaryNCIRoleCode*. Table *OtherRole* also references class *Roles*.

Statement:

```
CREATE TABLE OtherRole (
  otherRoleID NUMBER(5),
  otherRole VARCHAR2(100) NOT NULL,
  rolesID NUMBER(5) NOT NULL,
  CONSTRAINT PK_OtherRole PRIMARY KEY (otherRoleID),
  CONSTRAINT FK_OtherRole_Sentence FOREIGN KEY (rolesID)
  REFERENCES Roles
);
```

Table *OtherRole* is of the same structure as table *PrimaryNCIRole-Code*. The following table is called *EvidenceCode* and contains a string, an ID and a foreign key.

Statement:

```
CREATE TABLE EvidenceCode (
  evidenceID NUMBER(5),
  evidenceCode VARCHAR2(50) NOT NULL,
  sentenceID NUMBER(5) NOT NULL,
  CONSTRAINT PK_EvidenceCode PRIMARY KEY (evidenceID),
  CONSTRAINT FK_EvidenceCode_Sentence FOREIGN KEY (sentenceID)
  REFERENCES Sentence
);
```

The same applies for table *Comments*.

Statement:

```
CREATE TABLE Comments (
  commentsID NUMBER(5),
  comments VARCHAR2(300) NOT NULL,
  sentenceID NUMBER(5) NOT NULL,
  CONSTRAINT PK_Comments PRIMARY KEY (commentsID),
  CONSTRAINT FK_Comments_Sentence FOREIGN KEY (sentenceID)
  REFERENCES Sentence
);
```

Then, after all classes are implemented as tables within the relational database, these tables need to be filled with values. This can be done by SQL 'INSERT' statements. To enter all entries, this needs to be done 55 times for table *GeneEntry*. For other tables there can be more or less statements. For table *GeneAlias*, these statements can look similar to the following.

Statement:

```
INSERT INTO GeneAlias (geneAliasID, geneAlias, HUGOGeneSymbol)
VALUES (1,'polm','POLM');
INSERT INTO GeneAlias (geneAliasID, geneAlias, HUGOGeneSymbol)
VALUES (2,'polymerase (dna directed), mu','POLM');
```

After ‘INSERT INTO’ comes the name of the table. In the first parenthesis are the names of the columns and in the second parenthesis are the values which should be entered in the according columns. In this example, *geneAliasID* becomes 1, *geneAlias* is set to “polm” and *HUGOGeneSymbol* gets the value “POLM” for the first row. In the second row there is the next ID another value for *geneAlias* and the next foreign key.

These steps need to be done for all tables with all tuples. To avoid writing all these statements by hand, some simple python script which processes the XML document, identifies the tags and creates the INSERT statements from them, is written. The python script to create the insert statements is attached in Appendix B.

The final statement of each SQL script, containing the insert statements for a particular table, counts the number of inserted tuples. If everything was successful, the following numbers are displayed:

GeneEntry	55
GeneAlias	742
Sentence	164
Statement	164
Roles	163
PrimaryNCIRoleCode	263
OtherRole	192
EvidenceCode	191
Comments	26

Finally, all data of the XML document is stored within the relational database.

Possible data management steps to keep the data up-to-date would be updates, deletes or further inserts. If specific data from such a database needs to be retrieved, corresponding queries have to be issued against the database. The so-called SQL/XML Generation Functions may be used in SQL statements, take relational data, retrieved from the database as input and produce data in XML format as elements and attributes as output. Thus, various data can be combined and returned as value of a column of type XMLType.

This approach can be used to issue queries against relational databases, from various integrated data sources. Utilizing such functions is beneficial for bioinformaticians, who want to investigate as much data as possible to draw conclusions about the coherencies of samples, for instance.

In the following, some queries that could be interesting to know from the database are specified and the according statements and results are presented.

Query:
“Which organisms (in upper or lower case) other than human (organism != human) are affected by the cancer diseases? Return the organism, how often it occurs in the database (count) and then the according acronym(s), given by the Human Genome Organization (HUGO) as simple value in XML format.”

The `organism` is selected and the occurrences counted. To combine all `HUGOGeneSymbols` in one column, the functions XMLAgg() and XMLElement() are used. In the WHERE-clause it is defined that non-human organisms are needed and it is grouped by organism to have one tuple per organism in the result.

Statement:

```
SELECT organism, COUNT (*),
XMLAgg(XMLElement("HUGO", s.HUGOGeneSymbol))result
FROM Sentence s
WHERE lower(organism) != 'human'
GROUP BY organism;
```

Result:

```
ORGANISM  COUNT(*) RESULT
------------------ ------------------------------------------------
Mouse           31 <HUGO>RBM17</HUGO><HUGO>CLDN2</HUGO><HUGO>CLDN2<
Rat              6 <HUGO>ARHGAP5</HUGO><HUGO>C4BPA</HUGO><HUGO>CLDN
others           1 <HUGO>ARHGAP5</HUGO>
```

Table 3. Result from Query Regarding Genes Sorted by Organism (not whole lines shown).

There are 31 mice, six rats and one other organism in the result, whereby the XML elements for the *HUGOGeneSymbol* are aggregated to a simple value of type XMLType.

Query:

“Find all genes associated with a disease as primary role. (PrimaryNCIRoleCode = “Gene_Associated_With_Disease”). For further assumptions the name of the disease (matchedDiseaseTerm) and of the gene (matchedGeneTerm), as well as the UniProt identification number and the PubMed entry number are required. The IDs should be nested within an element called IDs.”

MatchedDiseaseTerm, *matchedGeneTerm*, and *pubMedID* are selected from table *Sentence* and *uniProtID* from table *GeneEntry*. The WHERE-clause filters for the *PrimaryNCIRoleCode*. Therefore, *GeneEntry* needs to be joined to *Sentence*, *Sentence* to *Roles*, and *Roles* to *PrimaryNCIRoleCode*.

Statement:

```
SELECT s.matchedDiseaseTerm, s.matchedGeneTerm,
XMLElement ("IDs", XMLForest(g.uniProtID, s.pubMedID)) result
FROM GeneEntry g, Sentence s,  Roles r, PrimaryNCIRoleCode p
WHERE g.HUGOGeneSymbol=s.HUGOGeneSymbol AND s.sentenceID=r.sentenceID
AND r.rolesID=p.rolesID
AND p.primaryNCIRoleCode = 'Gene_Associated_With_Disease';
```

Result:

The result is a table with the disease and gene names and in XML format UniProt accession number and the PubMed number.

MATCHEDDISEASETERM	MATCHEDGENETERM	RESULT
breast cancer	claudin-6	<IDs><UNIPROTID>P56747</UNIPROTID><PUBMEDID>12896909</PUBMEDID></IDs>
tumor	dleu7	<IDs><UNIPROTID>Q6UYE1</UNIPROTID><PUBMEDID>14706829</PUBMEDID></IDs>
chronic lymphocytic leukaemia	medullasin	<IDs><UNIPROTID>Q8IZJ6</UNIPROTID><PUBMEDID>1572695</PUBMEDID></IDs>
lymphoblastic leukaemia	medullasin	<IDs><UNIPROTID>Q8IZJ6</UNIPROTID><PUBMEDID>1572695</PUBMEDID></IDs>
acute myelocytic leukaemia	medullasin	<IDs><UNIPROTID>Q8IZJ6</UNIPROTID><PUBMEDID>1572695</PUBMEDID></IDs>
acute myeloid leukemia	ela2	<IDs><UNIPROTID>Q8IZJ6</UNIPROTID><PUBMEDID>14962902</PUBMEDID></IDs>
acute myelogenous leukemia (aml)	ela2	<IDs><UNIPROTID>Q8IZJ6</UNIPROTID><PUBMEDID>14594802</PUBMEDID></IDs>
human leukemia	ela2	<IDs><UNIPROTID>Q8IZJ6</UNIPROTID><PUBMEDID>14594802</PUBMEDID></IDs>

Table 4. Result from Disease Query (only the First Lines Shown).

4.3.5. Online Database Research

To compare queries against the database with online search with respect to speed and effort, a similar task for an online search was created.

Query:
"Given the gene TDH, you want to know its function in the human metabolism and the diseases it can cause by searching various databases. Why is it a 'pseudogene'? For further researches save the accession numbers from common used databases. Research a reasonable disease in detail."

Firstly, the full name of the gene needs to be found, since TDH is just an acronym. Therefore, the HUGO (Human Genome Organization) homepage [HUGO] is visited, the abbreviation is entered, and the result is chosen. The full name is "L-threonine dehydrogenase (pseudogene)". Now the corresponding accession numbers in PDB (5L9A), SwissProt (Q8IZJ6 (TDH_HUMAN)), ENA (AY101186.1 Homo sapiens L-threonine 3-dehydrogenase (TDH)) and GenBank (157739 TDH L-threonine dehydrogenase (pseudogene) [*Homo sapiens* (human)]) are searched by entering the full name (without the pseudogene) in their search fields. If available, the human gene is chosen.
Now, to find out why it is called a "pseudogene", PubMed is used to search again for the full name together with the term pseudogene.
The full text is chosen, where this statement can be found: "There is a loss of the acceptor splice sites in exon 4 and 6 and an in-frame stop codon in exon 6 (from the expected CGA to TGA) resulting in arginine-214 being replaced by a stop codon. Together, these mutations suggested that the human TDH gene is a pseudogene" [Edga02]. As a next step, at KEGG PATHWAY webpage "TDH" is entered into the search field.

Then the resulting entry link is taken, which is the 'Glycine, serine and threonine metabolism'. In the section 'disease' these are listed: "Primary hyperoxaluria (HP), Cystathioninuria, Homocystinuria, Hypermethioninemia, Nonketotic hyperglycinemia, Guanidinoacetate methyltransferase (GAMT) deficiency, Creatine deficiency syndrome, Dimethylglycine dehydrogenase deficiency (DMGDHD), 3-Phosphoglycerate dehydrogenase (3-PGDH) deficiency, Phosphoserine aminotransferase (PSAT) deficiency, Glycogen storage disease type X". By clicking on the graphical pathway, the role of Threonine in the pathway can be seen. The main path reveals that Threonine can become Glycine and Glycine can become Serine, and the other way round.
OMIM can be used to get more information about one of these diseases. Therefore, the OMIM browser is opened and the name of the first disease is entered in the search field. There are three different types of this disease. "Type I is an autosomal recessive disorder characterized by an accumulation of calcium oxalate in various bodily tissues, especially the kidney, resulting in renal failure. Affected individuals have decreased or absent AGXT activity and a failure to transaminate glyoxylate, which causes the accumulated glyoxylate to be oxidized to oxalate." [WAAC09]. Type II primary hyperoxaluria is caused by mutation in the glyoxylate reductase/hydroxypyruvate reductase gene on chromosome 9. Type III primary hyperoxaluria is caused by mutation in the mitochondrial dihydrodipicolinate synthase-like gene on chromosome 10q24.

Time and effort of this online database research were evaluated: finding these results (without documentation) took slightly more than 40 minutes. In this time, 12 searches were performed and over 24 hyperlinks were used. Collecting and summarizing the data took in total approximately 1.5 hours.

The succeeding steps concerning the data engineering example demonstrate important methods to enrich a database by mapping to different schemata and integrating additional data sources. They are not considered in the time comparison between the engineered database and online databases, but they are worth to be investigated because of their benefits.

4.3.6. Integration

In addition to the previous discussed procedure, there can be further steps with respect to an engineered database, which otherwise would have to be done manually. One possible further step is the explicit integration of other data sources.

Integration is necessary to combine and cross-analyze data from different sources. Some data makes only sense in the context of other collections of information. Integration can be used to create output files with shared information or to construct complex queries against different data sources using intersections in the data.

One integration mechanism is called ‘mapping’. It means to connect all corresponding components at the schema level. No additional integrated schema is required. After mapping the schemata, it is possible to transform the instances from the first schema to instances of the second schema. All mandatory concepts in the destination schema must be connected to create a valid schema.

a) Mapping Database to XML Schema

One example of mapping is to map a relational database to an XML schema and auto generate an XML file with all instances from the database. This could be required, for example, to send a part of the dataset to another laboratory.

To demonstrate this step, the Cancer Gene Disease database is mapped to the Cancer Gene Disease XML schema. This is done graphically utilizing the tool MapForce [Alto]. Since the database was originally created from the XML schema, all corresponding concepts can be mapped, and a valid XML instance file is created.

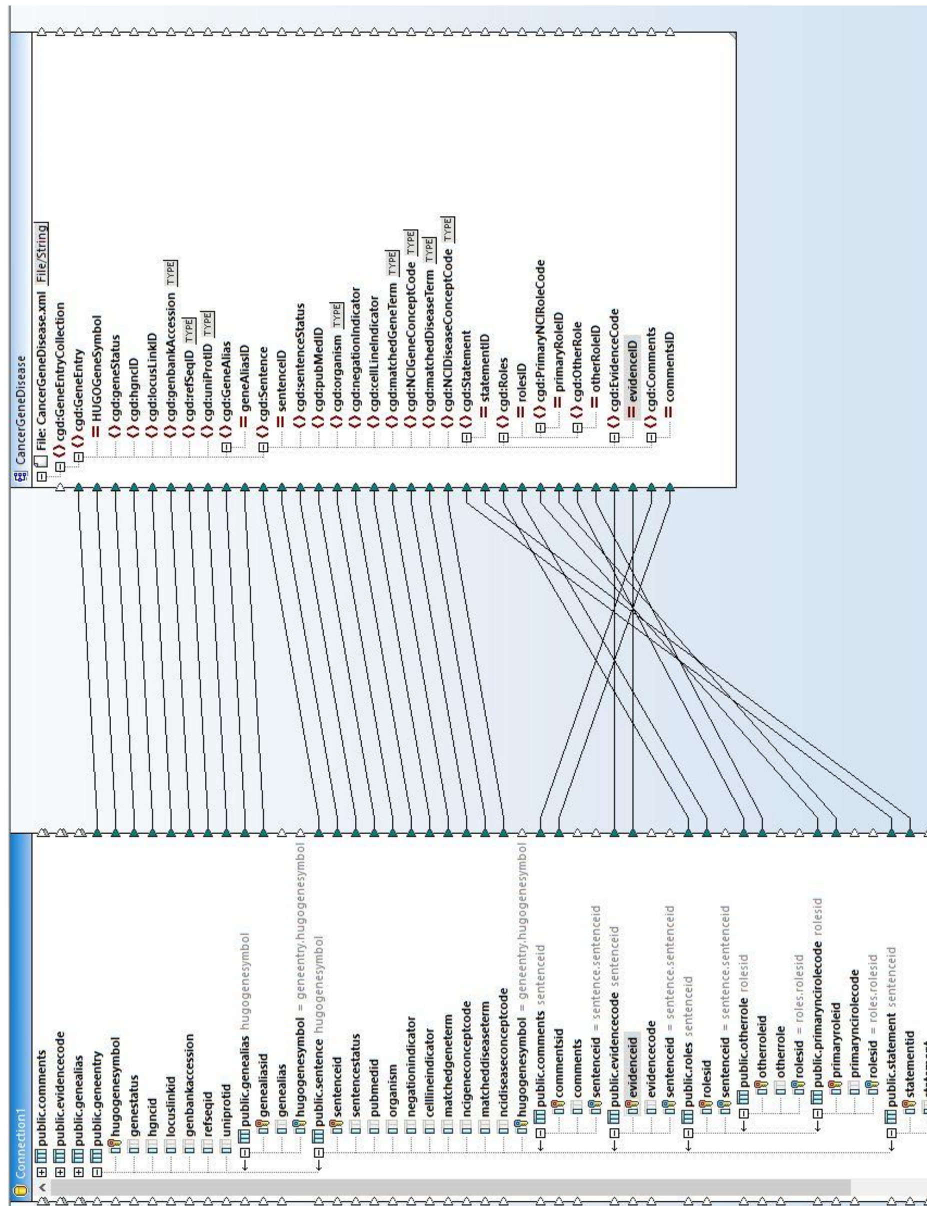


Fig. 8. Database to XML Schema Mapping.

b) Integration of SwissProt

If data from a different biological database should be considered in the particular research, this could be supported by data integration, too. For this example, data is taken from UniProt. For the purpose of integration, the Oracle XML DB functionality is used to register the uniprot XML schema in the example database. For storing instances of this XML schema, a table is created with a column `Version` which will contain a date, indicating the insertion date of the entry as well as a column for storing the XML instances. Then four sample entries are inserted with different dates. Two of these entries contain information about human genes, one about genes from mice and another one from rats. The samples were taken from SwissProt [SwPr], the curated part of UniProt. The statements for registration and insertion are attached in Appendix B. To demonstrate the operability of the added data, relating queries can be created.

Query:
For example, entries with the specific entry name SUH can be found by using the ‘contains’ function.

Statement:
```
SELECT XMLQuery('declare default element namespace
"http://uniprot.org/uniprot";
for $i in /uniprot/entry
where contains($i/name, "SUH")
return $i/gene'
PASSING xml_doc
RETURNING CONTENT) genes
FROM swissprot;
```

Result:
The only found gene with SUH in the name is “Rbpj”.

Query:

Next, all primary names and according synonyms can be displayed.

Statement:

```
SELECT gene_names.geneprimary, gene_names.genesynonym
FROM swissprot,
     XMLTable(XMLNAMESPACES(DEFAULT 'http://uniprot.org/uniprot'),
'for $i in /uniprot/entry/gene/name[@type="synonym"]
  return
    <temp>
      <primary>{$i/../name[@type="primary"]}</primary>
      <synonym>{$i}</synonym>
    </temp>'
PASSING xml_doc
COLUMNS geneprimary VARCHAR(60) PATH 'primary',
        genesynonym VARCHAR(60) PATH 'synonym') gene_names;
```

Result:

The result is a table listing the primary names and the according synonyms.

	GENEPRIMARY	GENESYNONYM
1	POLM	polmu
2	ATP11A	ATPIH
3	ATP11A	ATPIS
4	ATP11A	KIAA1021
5	ATP1A2	KIAA0778
6	ATP1A4	ATP1AL2
7	ATP1B1	ATP1B
8	KRTAP5-9	KAP5.9
9	KRTAP5-9	KRN1
10	KRTAP5-9	KRTAP5.9
11	KRTAP5-9	UHSK1
12	CGB3	CGB
13	TERF2IP	DRIP5

Table 5. Result from Query for Finding Synonym Names (only First Lines Shown).

Query:

Further, the number of entries for a specific organism, e.g. human, can be determined and shown for each version.

Statement:

```
SELECT Version, XMLQuery('declare default element namespace
 "http://uniprot.org/uniprot";
for $i in /uniprot/entry
return
<temp>
  <counter>
     {count(/uniprot/entry[organism/name[@type="common"]="Human"])}
  </counter>
</temp>'
PASSING xml_doc RETURNING CONTENT) Result
FROM swissprot;
```

Result:

In version 17-04-2018 and 18-04-2018 are eleven entries with human each. In version 19-04-2018 and 20-04-2018 are zero entries with organism human.

Since not all of the information contained in this XML file is needed, views on the important classes are created. The focus is set to five classes, namely *Protein*, *Accession*, *GeneNames*, *Keywords*, and *DatabaseReference*. The creation statements can be found in the Appendix B. In this way, the SwissProt data is virtually integrated with the relational database schema.

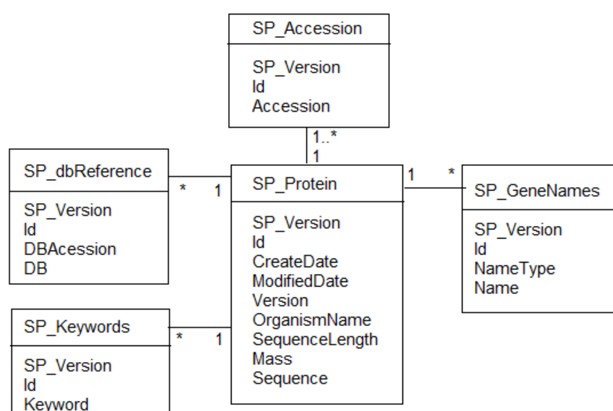


Fig. 9. UML SwissProt Views.

If these views are joined to the tables from Cancer Gene Disease, a query across the originally separated data can be performed. There are two correspondences in the data, these should be used as connectors, i.e., join attributes.

Query:

An example for an integrated query between the Cancer Gene Disease example and the SwissProt views would be to find all genes with organisms which are not human, together with the according UniProt accession number and the according names from both.

Statement:

```
SELECT DISTINCT GeneEntry.uniProtID, SP_Accession.Accession,
GeneEntry.HUGOGeneSymbol, SP_GeneNames.name AS SP_Genename,
SP_Protein.OrganismName
FROM GeneEntry
INNER JOIN Sentence ON GeneEntry.HUGOGeneSymbol=
                       Sentence.HUGOGeneSymbol
INNER JOIN SP_Accession ON GeneEntry.uniProtID=SP_Accession.Accession
INNER JOIN SP_Protein ON Sentence.organism=SP_Protein.OrganismName
INNER JOIN SP_GeneNames ON SP_Protein.ID=SP_GeneNames.ID
AND SP_Protein.SP_version=SP_GeneNames.SP_version
INNER JOIN SP_Accession ON SP_Protein.ID=SP_Accession.ID
AND SP_Protein.SP_version=SP_Accession.SP_version
WHERE Sentence.organism!='Human';
```

Result:

As a result, the *HUGOGeneSymbols* with the according name in SwissProt for entries with the same *UniProtID* and *organism* are displayed.

	UNIPROTID	ACCESSION	HUGOGENESYMBOL	SP_GENENAME	ORGANISMNAME
1	Q9NP87	Q9NP87	POLM	F9	Mouse
2	Q9NP87	Q9NP87	POLM	Snf2b	Mouse
3	Q9NP87	Q9NP87	POLM	Snf214	Mouse
4	Q9NP87	Q9NP87	POLM	Smarcd3	Mouse
5	P56747	P56747	CLDN6	Atp2a1	Rat
6	P56747	P56747	CLDN6	Atf3	Rat
7	P56747	P56747	CLDN6	Rah1	Rat
8	Q9NP87	Q9NP87	POLM	Smarca4	Mouse
9	P56747	P56747	CLDN6	Atf2	Rat
10	P56747	P56747	CLDN6	Atp5h	Rat
11	P56747	P56747	CLDN6	Atp5i	Rat
12	P56747	P56747	CLDN6	Atp5f1b	Rat
13	P56747	P56747	CLDN6	Atp5b	Rat

Table 6. Result from SwissProt joined with Cancer Gene Disease (Exemplary Lines).

c) Integration of Prosite

Integration can be done even between different formats. To demonstrate this step, a file from the Prosite [Pros] database in flat file format is selected. The lines need to be split up to be used in a database. For the integration with the Cancer Gene Disease example, three views are created: Entry, Pattern, and DatabaseReference. One entry in Prosite can contain several database references and several patterns, since they can be spread over more lines. All classes get an artificial ID ('AID'). The entire code for the creation of the external table and the views are located in Appendix B.

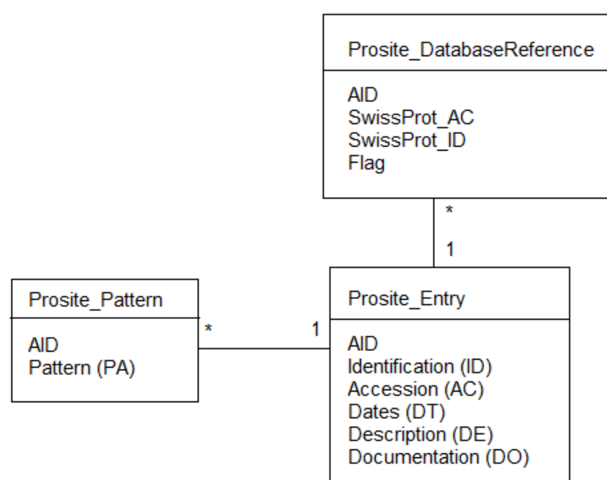


Fig. 10. UML Prosite Views.

Query:

The attribute *SwissProt_ID* from class *DatabaseReference* can be joined with the ID from class *Protein* from the SwissProt views. The intersection clause can be used to find entries available in both datasets with the same SwissProt ID.

Statement:

```
SELECT SwissProt_ID FROM Prosite_DatabaseReference
INTERSECT
SELECT ID FROM SP_Protein;
```

Result:

In the concrete example there are 37 intersections of IDs.

Query:

One of them is called ‘CREB3_HUMAN’, which will be investigated further. Hence, the artificial ID (AID), the value of the pattern, and the entry accession are selected for this gene. To join the tables, they need to have the same AID.

Statement:

```
SELECT p.aid, p.pattern, e.accession FROM Prosite_Pattern p, Pro-
site_Databasereference d, Prosite_Entry e
WHERE upper(d.Swissprot_ID) = 'CREB3_HUMAN'
AND d.aid = p.aid
AND e.aid = p.aid;
```

Result:

The displayed AID is 0031 and the pattern is [KR]-x(1,3)-[RKSAQ]-N-{VL}-x-[SAQ](2)-{L}-[RKTAENQ]-x-R-{S}-[RK].

Query:

As a next step it is counted how many Prosite database references belonging to this artificial ID truly have this pattern. These patterns have ‘T’ (short for true) as ‘Flag’.

Statement:

```
SELECT COUNT(*) FROM Prosite_Databasereference
where
aid = ‘00031’ and
flag = 'T';
```

Result:

For ‘CREB3_HUMAN’ there are 235 entries, which have the ID from the previous query before and ‘T’ as flag.

Query:

To examine, if there are entries in the SwissProt data fitting to this pattern, the pattern is searched in the sequence of the protein by a regular expressions function. Therefore, the pattern has to be translated according to the rules from Oracle XML DB [OXDB].

Statement:

```
SELECT * FROM SP_PROTEIN
WHERE
REGEXP_LIKE(sequence,'[KR].{1,3}[RKSAQ]N[^VL].[SAQ]{2}[^L][RKTAENQ].R
[^S][RK]');
```

Result:

The result of this query is the ‘CREB3_HUMAN’ itself and two more proteins with organism rat plus all other information from the various columns.

If files from different sources should be integrated and their schemata are available, these schemata can be mapped, and the instances may be transformed automatically according to these schemata. By the process of mapping corresponding concepts in both schemata are identified and connected. There are functions to transform elements, thus even not completely identical entries can be combined.

d) Mapping XML Schema to JSON

An often occurring case in biological research is that other research teams are dealing with the same topic. To share and compare data it is essential to have them in the same format. To demonstrate the process of mapping the current data to some possibly foreign schema, a schema with several heterogeneities was created. There are inconsistencies in schema structure and content. The new schema is written in JSON format and has a similar *GeneEntry* class, but the subclasses vary. In Fig. 11 the UML class diagram of the JSON Cancer Gene Disease schema can be seen. The accession numbers are stored in a separate class *SequenceIdentificationList*, while the classes *Statement* and *Comments* are only represented as string attributes in class *Sentence*. The classes except *GeneEntry* and *Sentence* have no unique IDs in this schema. Two additional classes are shown, one with the gene information, one with the disease information. *OtherRole* and *PrimaryNCIRoleCode* are not distinguished, while the attribute *date* and *ENAID* are not available in the schema from the previous example and need to be introduced.

In the new schema, *evidence* only needs the information about the source of evidence from *EvidenceCode*, which is the last part, for example NAS from EV-AS-NAS. The *sentenceID* in the JSON schema should be constituted from the first two letters from *organism* followed by a generated digit code. The indicators are not expressed as boolean but with 'yes' and 'no'. Further, the *sentenceStatus* is boolean in the JSON schema, which indicates 'true', only if the sentence is 'finished'. The datatype from *LocusLinkID* needs to be transformed from integer to string. And the enumeration strings from *geneStatus* and *entryStatus* in the JSON schema differ in nomenclature.

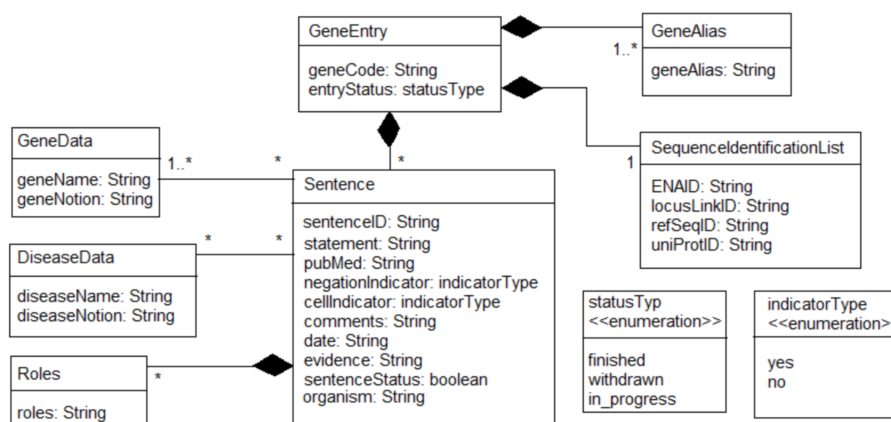


Fig. 11. UML JSON Schema.

The following functions from MapForce [Alto] were used:

- 'Value-map' to turn "true" in "yes" and "false" in "no" and to transform the enumeration strings for the statusTyp.
- 'String' to convert the integer from *LokusLinkID* to string.
- 'Equal' to test if the *sentenceStatus* is finished.
- 'Substring' to find the first two characters from *organism*.
- 'Auto-number' for generating consecutive numbers, which are added to the characters from substring and form the *sentenceID*.
- 'Concat' to concatenate *OtherRole* and *PrimaryNCIRoleCode* together, separated by a comma.
- 'Substring-after' to find the last part of the *EvidenceCode*, which occurs after the second '-' character.
- 'String-join' to store all elements to one string.

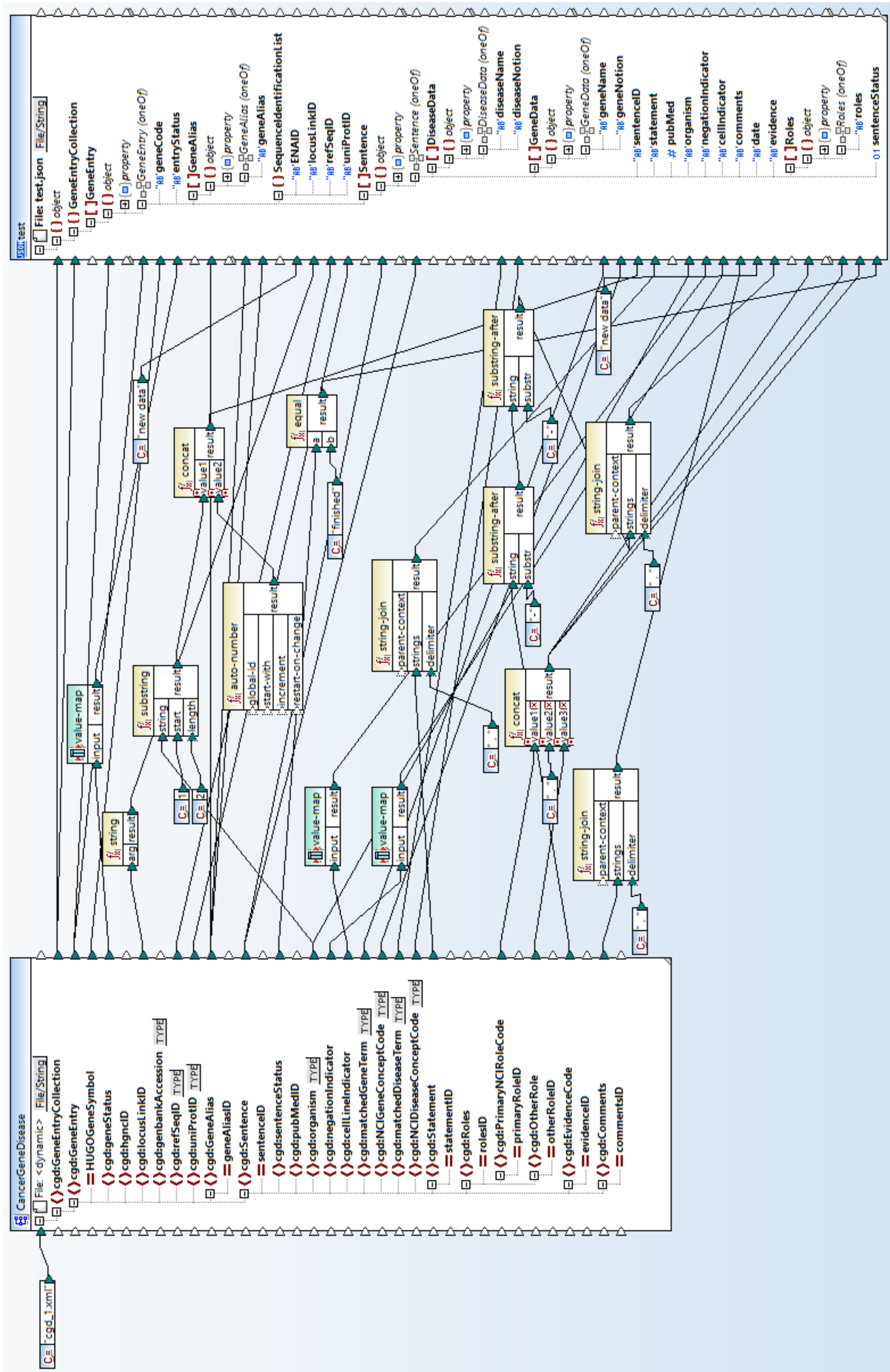


Fig. 12. Mapping XML Schema to JSON.

e) Integration of Gene Ontology GO

As mentioned in an earlier chapter, one special kind of integration is the integration utilizing ontologies. This is very useful in bioanalytical researches. Ontologies can be used to get further information about names or relationships of some concepts.

To demonstrate this function by means of the example database from the chapters before, a very small fragment from Gene Ontology (GO) [GeOn] in SQL format was selected and edited accordingly. This is not the typical format for ontologies but is more convenient to use in this example.

The selected part of the ontology consists of two tables. One table contains terms with attributes, like name, type and some short form describing different biological terms.

ID	NAME	TERM_TYPE	ACC	IS_OBSOLETE	IS_ROOT	IS_RE
1	is_a	relationship	is_a	0	0	1
2	consider	metadata	consider	0	0	1
3	replaced_by	metadata	replaced_by	0	0	1
4	Grouping classes that can be excluded	subset	goantislim_grouping	0	0	0
5	Term not to be used for direct annotation	subset	gocheck_do_not_annotate	0	0	0
6	Term not to be used for direct manual annotation	subset	gocheck_do_not_manually_annotate	0	0	0
7	AGR slim	subset	goslim_agr	0	0	0
8	Aspergillus GO slim	subset	goslim_aspergillus	0	0	0
9	Candida GO slim	subset	goslim_candida	0	0	0
10	ChEMBL protein targets summary (mouse)	subset	goslim_chembl	0	0	0
11	Generic GO slim	subset	goslim_generic	0	0	0
12	GOA and proteome slim	subset	goslim_goa	0	0	0

Table 7. Table term.

The second table is called term2term and contains IDs from two terms, that are in some relation, and the type of their relationship. The statements for creating the tables can be inspected in Appendix B.

ID	RELATIONSHIP_TYPE_ID	TERM1_ID	TERM2_ID	IS_COMPLETE
1	1	36	35	0
2	1	36	43	0
3	1	59	58	0
4	1	36	76	0
5	1	59	80	0
6	1	36	82	0
7	1	59	87	0
8	1	59	93	0
9	1	59	98	0
10	1	59	103	0

Table 8. Table term2term.

By joining this information to our dataset, a larger knowledge base is obtained. Firstly, all genes with organism mouse are selected.

Statement:

```
SELECT DISTINCT organism, HUGOGeneSymbol
FROM Sentence
WHERE organism='Mouse';
```

Result:

	ORGANISM	HUGOGENESYMBOL
1	Mouse	SAA3P
2	Mouse	ENAH
3	Mouse	TTK
4	Mouse	ARHGAP5
5	Mouse	POLM
6	Mouse	CLDN2
7	Mouse	RBM17
8	Mouse	EED
9	Mouse	STRA6
10	Mouse	UBC
11	Mouse	RAN
12	Mouse	TNFSF9

Table 9. All genes with Organism Mouse.

Then, in the table term it is searched for the occurrence of the word mouse using the like function. Accessing the term2term table allows to find all related terms for the resulting IDs.

Statement:

```
SELECT t1.id, t1.name AS term, t2.id AS relatedID, t2.name AS relat-
edTerm
FROM term t1, term2term tt, term t2
WHERE LOWER(t1.name) LIKE '%mouse%' AND
      t1.id = tt.term1_id AND
      tt.term2_id = t2.id
UNION
SELECT t1.id, t1.name AS term, NULL, NULL
FROM term t1
WHERE LOWER(t1.name) LIKE '%mouse%' AND
      t1.id NOT IN
      (SELECT tt.term1_id
       FROM term2term tt);
```

Result:

The result shows the found terms and their related terms found by the ID from table term2term.

	ID	TERM	RELATEDID	RELATEDTERM
1	10	ChEMBL protein targets summary (mouse)	(null)	(null)
2	14	Mouse GO slim	18	synapse GO slim

Table 10. Resulting Table for Sentence Joined with Term.

For the current example research this information might not be a significant improvement, but for subsequent queries in later research this increased data supply can be necessary or at least helpful.

4.3.7. Results

After proceeding all these steps, a comparison of the two approaches – data engineering and online search – can be done.

The effort itself is hard to compare, the online database search included 12 searches and over 24 hyperlinks to get the result for the query. The creation of the database itself was a huge effort of writing very roughly estimated 3000 lines of code. But accessing the database, by sending a query, is a perceptible less complicated task of writing about 5 to 10 lines on average. Neglecting the creation of the database, the engineered database has a clear advantage over the online database research.

A better factor to evaluate the utility of engineering an own database is the needed time per query. The evaluated time of the online database research for collecting and summarizing the data was approximately 1.5 hours. This time is needed for every single query.

Times estimated for the construction and processing of the engineered database:

Editing data set:	5 h	
Schema creation:	2 h	DTD
	3 h	XML schema
Data base creation + insertion:	4 h	
Every Query:	15 min	

This gives an approximate time of 14.15 hours for doing a single query by using data engineering. This is a considerably higher number than 1.5 on the first sight. But usually a biologist or bioinformatician works over years on the same data set and needs to do more than one query. Some more time periods were calculated for further queries and compared to the constant time from online database search. Therefore, the total time was divided through the number of queries, since the data framework is constructed only once. In case of more than ten queries, time can be saved. The eleventh query only took 1.42 hours according to these numbers.

For a high number of queries – which typically will be the case if more people work over years with the database – comparable low values are reached. For 1000 queries it would take approximately only 16 minutes per query.

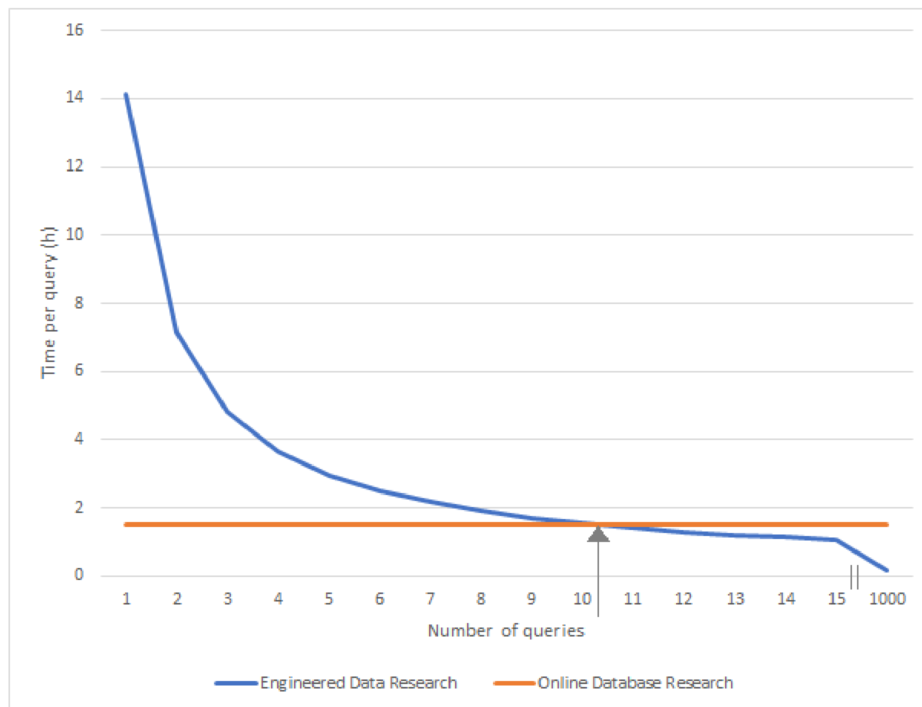


Fig. 13. Time Comparison Online vs. Engineered Database.

In Fig. 13 one can see that the time per query stays constant for the online database research, while the time per query decreases almost exponentially for the engineered data research. The lines are crossing between the tenth and the eleventh query. For ten queries and more the data engineering approach gets profitable. For over 1000 queries one can almost neglect the time needed for the pre-processing engineering steps.

Even more important are the opportunities given by an engineered database. More elaborated queries are possible by joining different tables through additional lines in the query. Searching online it is necessary to filter for all the requested data in several specialized databases manually. Even if there are functions to filter the results from the online database, it is more convenient to do this directly by the query.

The number of query results possibly increases, if more data sets were integrated together. Data from different data sources, from different labs for example, can be merged by mapping the schemata. All the instances unite accordingly. This means more entries are available and thus probably more results.

Further, the informativeness of the results enhances due to the enrichment by various data with miscellaneous content. If schemata which are not very similar but have at least one element in common are connected, the information from both is brought together in one place.

By the usage of an ontology the query hits can be improved immense. On the one hand the number of hits can be increased, because results with similar nomenclature are found. On the other hand, the quality is improved, since the result is expanded by the synonyms and definitions.

4.4. Discussion

The practical work of this thesis represents a demonstration of different engineering steps. In a real-world data set it is often not possible to change the structure afterwards according to the needs of the next step. Furthermore, the following steps might be unknown, thus the requirements cannot be fulfilled previously.

The analysis of time is a very rough valuation and does not consider several alternating cases. Firstly, the numbers were estimated for the construction of this specific dataset. They cannot be directly projected on every dataset. If the dataset is larger, for example, the computer will probably take longer to treat the query. Filtering through a huge database can take clearly longer than 15 minutes. The construction of the database depends on the complexity of the data. More classes with more constraints can lead to deviations in time. Taking into account that most biologists themselves do not have the necessary experience in programming and creating information systems, this task can be challenging. The needed time therefore depends also on the capability of the person constructing and using the database. Having a bioinformatician for these tasks saves time. Another aspect is the fact that adapting changes to an existing data framework can be very difficult and again time needs to be invested. If a new column, for instance, is required, all schemata, data entries and tables have to be changed. Thus, the modification of an existing dataset can be error-prone. Nevertheless, it pays off to invest the time for creating a database for long term projects. If a dataset needs to be investigated frequently, maybe even from more than one person or over a longer time span, it might be beneficial to use an own data framework.

In addition to time, data engineering provides huge benefits in the field of bioinformatics. These benefits are more query hits with higher quality and richer informativeness, for example. Another benefit is the possibility to create very comprehensive queries, also across tables storing data from different data sources.

5. Conclusion

This thesis deals with the application of data engineering technologies in bioinformatics. The theoretical part gives an overview on data bases and data formats common in bioinformatics and focuses on integration of biological data. Some approaches, resulting possibilities and challenges of data engineering in the field of bioinformatics are shown. In the practical part, an overall process for a real-world biological data set to an integrated database was accomplished. Various steps of a typical data engineering process were walked through and are analyzed. Through integration steps the knowledge base can be increased and the results can be more meaningful. If a database is often used, the average time per query decreases. There are useful query languages for databases, which make the process of searching uncomplicated. Time saving, mastering of complex queries and the convenience of an engineered database can have enormeous benefits over usual online search. Having a well-engineered data framework can enhance the work of a bioinformation and can lead to more sophisticated data analyses.

6. References

6.1. Literature

[AtPT16] Teresa K. Attwood, Stephen R. Pettifer, David Thorne. “Bioinformatics Challenges at the Interface of Biology and Computer Science”. John WileySons Ltd. 2016.

[AnSt15] Antony T. Vincent, Steve J. Charette. “Who qualifies to be a bioinformatician?”. Front Genetics, 6:164. 2015.

[Belt16] Michael Benedikt, Rodrigo Lopez-Serrano, Efthymia Tsamoura. “Biological Web Services: Integration, Optimization, and Reasoning”. BAI@IJCAI. 2016.

[CBFC16] Charles E. Cook, Mary Todd Bergman, Robert D. Finn, Guy Cochrane, Ewan Birney, and Rolf Apweiler. “The European Bioinformatics Institute in 2016: Data growth and integration”. Nucleic Acids Research, 44(Database issue), D20–D26. 2016.

[Cera06] Ethan Cerami. “XML for Bioinformatics”. Springer Science +Business Media, Inc. 2006.

[Codd70] E. F. Codd. “A relational model of data for large shared data banks”. Communications of the ACM 13(6), pp. 377-387. 1970.

[Cons15] The Gene Ontology Consortium. “Gene Ontology Consortium: going forward”. *Nucleic Acids Research*, 43(D1), D1049–D1056. 2015.

[Cons17] The Gene Ontology Consortium. “Expansion of the Gene Ontology knowledgebase and resources”. Nucleic Acids Research, 45(Database issue), D331–D338. 2017.

[Edga02] Edgar, A. J. “The human L-threonine 3-dehydrogenase gene is an expressed pseudogene”. BMC Genetics, 3, 18. 2002.

[GaFR17] Michael Y. Galperin, Xosé M. Fernández-Suárez, Daniel J. Rigden. “The 24th annual Nucleic Acids Research database issue: a look back

and upcoming changes”. *Nucleic Acids Research*, 45(D1), D1-D11. 2017.

[Groß14] Dr. Anika Groß. “BioDM”. University Leipzig. 2014.

[Holz03] Steven Holzner. “Creating Well-Formed XML Documents”. Informit. 2003.

[KFTS17] Minoru Kanehisa, Miho Furumichi, Mao Tanabe, Yoko Sato, Kanae Morishima. “KEGG: new perspectives on genomes, pathways, diseases and drugs”. *Nucleic Acids Research*, 45(D1), D353–D361. 2017.

[KGHA06] Minoru Kanehisa, Susumu Goto, Masahiro Hattori, Kiyoko Aoki-Kinoshita, Masumi Itoh, Shuichi Kawashima, Toshiaki Katayama, Michihiro Araki, Mika Hirakawa. “from genomics to chemical genomics: New developments in KEGG”. Nucleic Acids Research 34(Database issue):D354-7. 2006.

[KGSF12] Kanehisa, M., Goto, S., Sato, Y., Furumichi, M., & Tanabe, M. “KEGG for integration and interpretation of large-scale molecular data sets”. Nucleic Acids Research, 40(Database issue), D109–D114. 2012.

[LaWa15] Brendan Lawlor, Paul Walsh. “Engineering bioinformatics: building reliability, performance and productivity into bioinformatics software “. Bioengineered 6:4, 193—20.

[LeNa07] Ulf Leser, Felix Naumann (2007): “Informationsintegration”. dpunkt.verlag GmbH. 2015.

[LSJV15] Vasileios Lapatas, Michalis Stefanidakis, Rafael C. Jimenez, Allegra Via and Maria Victoria Schneider. “Data integration in biological research: an overview”. Journal of Biological Research, 22(1), 9. 2015.

[Neil14] Neil Savage. “Bioinformatics: Big data versus the big C”. Nature 509, S66–S67. 2014.

[PBJM15] Christoph Pinkel, Carsten Binnig, Ernesto Jiménez-Ruiz, Wolfgang May, Dominique Ritze, Martin G. Skjæveland, Alessandro Solimando, Evgeny Kharlamov. “RODI: A Benchmark for Automatic Mapping Generation in Relational-to-Ontology Data Integration”. Springer LNCS 9088, pp. 21-37. 2015.

[Pohj08] Jussi Pohjolainen. “Introduction to XML”. Slideshare. 2008.

[Rahm04] Erhard Rahm. “Data Integration in the Life Sciences”. Springer-Verlag Berlin Heidelberg. 226. 2004.

[Siko14] Leslie Sikos. “Web Standards: Mastering HTML5, CSS3, and XML”. Apress. 2014.

[TJTM14] Rachel Thompson, Louise Johnston, Domenica Taruscio, Lucia Monaco, Christophe Béroud, PharmD, Ivo G. Gut, Mats G. Hansson, Peter-Bram A. ’t Hoen, George P. Patrinos, Hugh Dawkins, Monica Ensini, Kurt Zatloukal, David Koubi, Emma Heslop, Justin E. Paschall, Manuel Posada, Peter N. Robinson, Kate Bushby, Hanns Lochmüller: “RD-Connect: An Integrated Platform Connecting Databases, Registries, Biobanks and Clinical Bioinformatics for Rare Disease Research”. Journal of General Internal Medicine, 29 (Suppl 3), 780–787. 2014.

[USDT10] U.S. Department of Transportation - Federal Highway Administration. “Data Integration Primer”. pp. 31-35. 2010.

[WAAC09] Williams, E. L., Acquaviva, C., Amoroso, A., Chevalier, F., Coulter-Mackie, M., Monico, C. G., Giachino, D., Owen, T., Robbiano, A., Salido, E., Waterham, H., Rumsby, G. “Primary hyperoxaluria type 1: update and additional mutation analysis of the AGXT gene”. Hum. Mutat. 30: 910-917. 2009.

6.2. Online Resources

[AlIn] Allergy Intolerance schema: https://www.hl7.org/FHIR/2015May/allergyintolerance.html, last access: 23.10.2017

[AllM] Allergy schema for mapping: https://docs.microsoft.com/en-us/healthvault/datatypes/allergy#xsd-schema, last access: 23.10.2017

[Alto] Altova MapForce: https://www.altova.com/de/mapforce, last access: 15.05.2018

[CaGD] Cancer Gene Disease schema: https://wiki.nci.nih.gov/display/cageneindex/The+Cancer+Gene+Index+Gene-Disease+and+Gene-Compound+XML+Documents, last access: 12.04.2018

[ENAh] ENA homepage: http://www.ebi.ac.uk/ena/about, last access: 08.12.2017

[ExPA] ExPASy homepage: https://www.expasy.org/, last access: 06.12.2017

[FFDB] Flat File Database: Definition & Example: https://study.com/academy/lesson/flat-file-database-definition-example.html, last access: 05.06.2018

[GenB] GenBank homepage: https://www.ncbi.nlm.nih.gov/genbank/, last access: 05.12.2017

[GeOn] GO homepage: www.geneontology.org/, last access 04.04.2018

[HPAs] HPA schema: https://www.proteinatlas.org/download/proteinatlas.xsd, last access: 30.11.2017

[HUGO] Human Genome Organisation homepage: http://www.hugo-international.org/, last access: 05.06.2018

[ImmM] Immunization schema for mapping: https://docs.microsoft.com/en-us/healthvault/datatypes/immunization#xsd-schema, last access: 19.10.2017

[Immu] Immunization schema: https://www.hl7.org/fhir/immunization.html, last access: 18.10.2017

[JSON] JSON example: http://json-schema.org/example1.html, last access: 05.06.2018

[KEGG] KEGG homepage: www.genome.jp/kegg/, last access: 08.01.2018

[OXDB] Oracle XML DB documentation: https://docs.oracle.com/database/121/ADXDB/xdb05sto.htm#ADXDB0600, last access: 24.04.2018

[Pros] Prosite homepage: https://prosite.expasy.org/, last access: 24.04.2018

[RCSB] RCSB homepage: www.rcsb.org/, last access: 06.12.2017

[SNPM] SNP schema for mapping: https://www.ncbi.nlm.nih.gov/projects/SNP/docsum/docsum0.html, last access:18.11.2017

[SNPs] SNP schema: http://dbarchive.biosciencedbc.jp/archive/openpml/2009/snp.xsd, last access: 16.11.2017

[SwPr] SwissProt homepage: http://www.uniprot.org/uniprot/?query=reviewed:yes, last access: 24.04.2018

[UniP] UniProt schema: http://www.uniprot.org/docs/uniprot.xsd, last access: 29.11.2017

[W3Ch] W3C homepage: https://www.w3.org/standards/xml/components.html, last access: 01.02.2018

7. List of Figures

Fig. 1. Total Disk Storage at EMBL-EBI [CBFC16]…………………………….. 6
Fig. 2. Sample Fragment from a Well-formed XML Document [Holz03]…….. 11
Fig. 3. Sample Fragment from a Valid JSON Document [JSON]…………….. 12
Fig. 4. Virtual Integration at EMBL-EBI [CBFC16]……………………………… 18
Fig. 5. KEGG Main Databases and their Relationship [KGHA06]……………. 21
Fig. 6. Photosynthesis Pathway Map from KEGG………………………………. 22
Fig. 7. Cancer Gene Disease UML……………………………………………….. 31
Fig. 8. Database to XML Schema Mapping………………………………………. 53
Fig. 9. UML SwissProt Views……………………………………………………… 56
Fig. 10. UML Prosite Views………………………………………………………... 58
Fig. 11. UML JSON Schema………………………………………………………… 61
Fig. 12. Mapping XML Schema to JSON………………………………………….. 62
Fig. 13. Time Comparison Online vs. Engineered Database……………………. 67

8. List of Tables

Table 1. Common Database Comparison………………………………………… 9
Table 2. Comparison of Data Sources……………………………………………... 27
Table 3. Result from Query Regarding Genes Sorted by Organism (not whole lines shown)……………………………………………………………... 49
Table 4. Result from Disease Query (only the First Lines Shown).……………... 50
Table 5. Result from query for finding synonym names (only First Lines Shown)…………………………………………………………………… 55
Table 6. Result from SwissProt joined with Cancer Gene Disease (Exemplary Lines)……………………………………………………………………... 57
Table 7. Table term………………………………………………………………….. 63
Table 8. Table term2term……………………………………………………………. 64
Table 9. All Genes with Organism Mouse..……………………………………….. 64
Table 10. Resulting Table for Sentence Joined with Term…………..…………… 65

9. List of Listings

Listing 1. Cancer Gene Disease Example Entry………………………………… 32
Listing 2. DTD Schema for Cancer Gene Disease Example…………………… 36
Listing 3. XML Schema for Cancer Gene Disease Example…………………… 38

10. Appendix A: UML Class Diagrams

a) Immunization

<<XSDelement>>
immunization

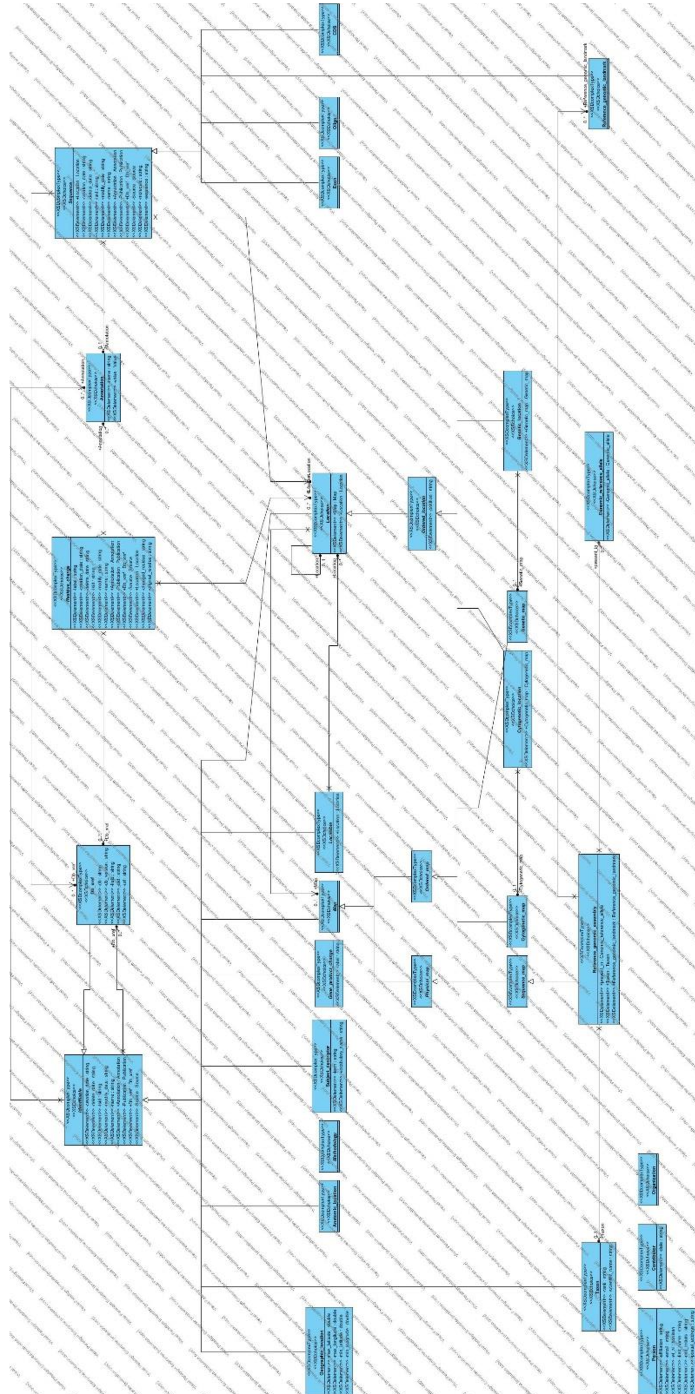
<<XSDelement>> -name : codable-value
<<XSDelement>> -administration-date : approx-date-time
<<XSDelement>> -administrator : person
<<XSDelement>> -manufacturer : 2001.XMLSchema.string
<<XSDelement>> -lot : 2001.XMLSchema.string
<<XSDelement>> -route : codable-value
<<XSDelement>> -expiration-date : approx-date
<<XSDelement>> -sequence : 2001.XMLSchema.string
<<XSDelement>> -anatomic-surface : codable-value
<<XSDelement>> -adverse-event : 2001.XMLSchema.string
<<XSDelement>> -consent : 2001.XMLSchema.string

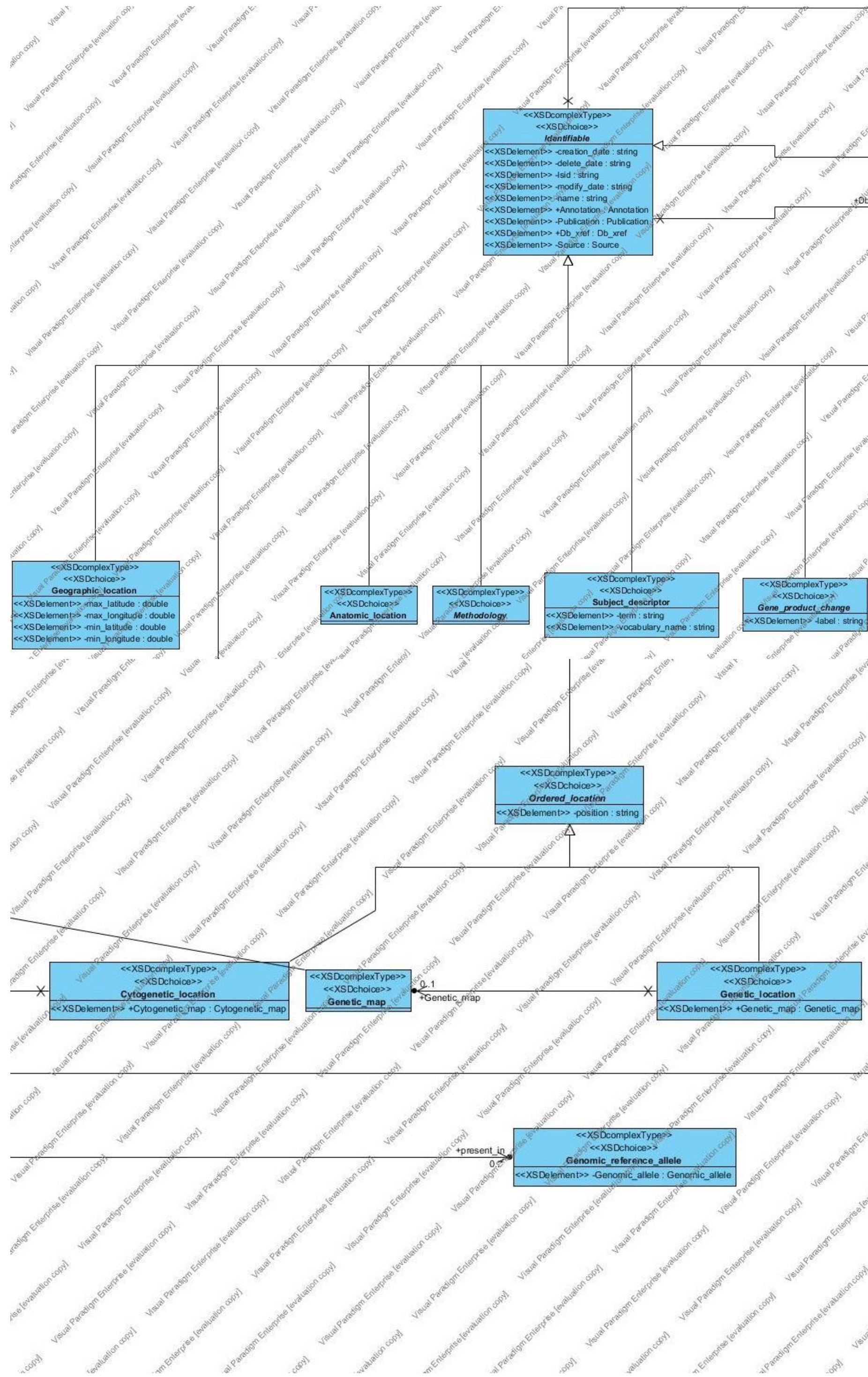
b) Allergy

<<XSDelement>>
allergy

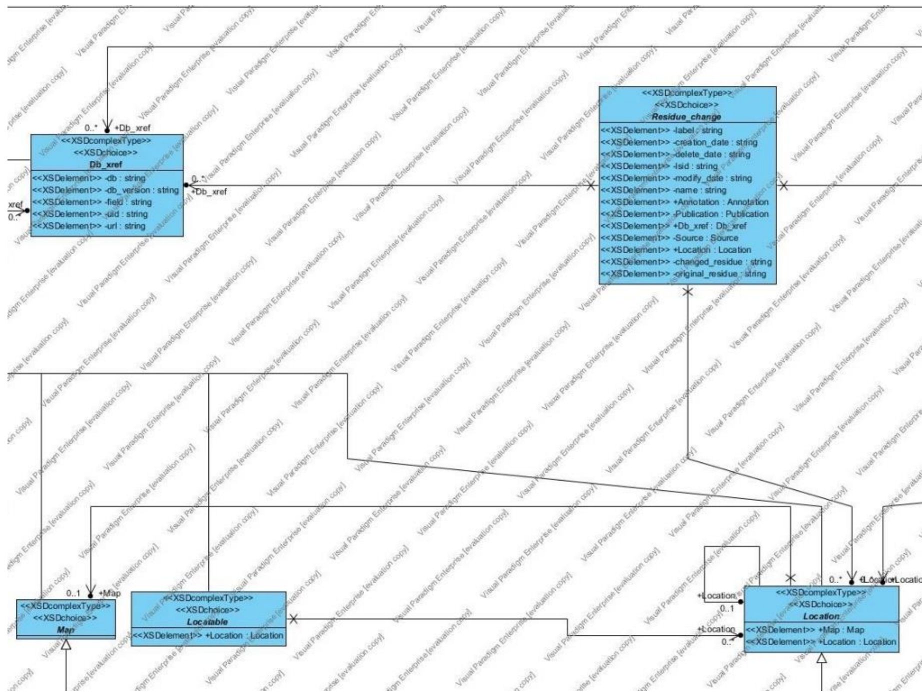
<<XSDelement>> -name : codable-value
<<XSDelement>> -reaction : codable-value
<<XSDelement>> -first-observed : approx-date-time
<<XSDelement>> -allergen-type : codable-value
<<XSDelement>> -allergen-code : codable-value
<<XSDelement>> -treatment-provider : person
<<XSDelement>> -treatment : codable-value
<<XSDelement>> -is-negated : 2001.XMLSchema.boolean

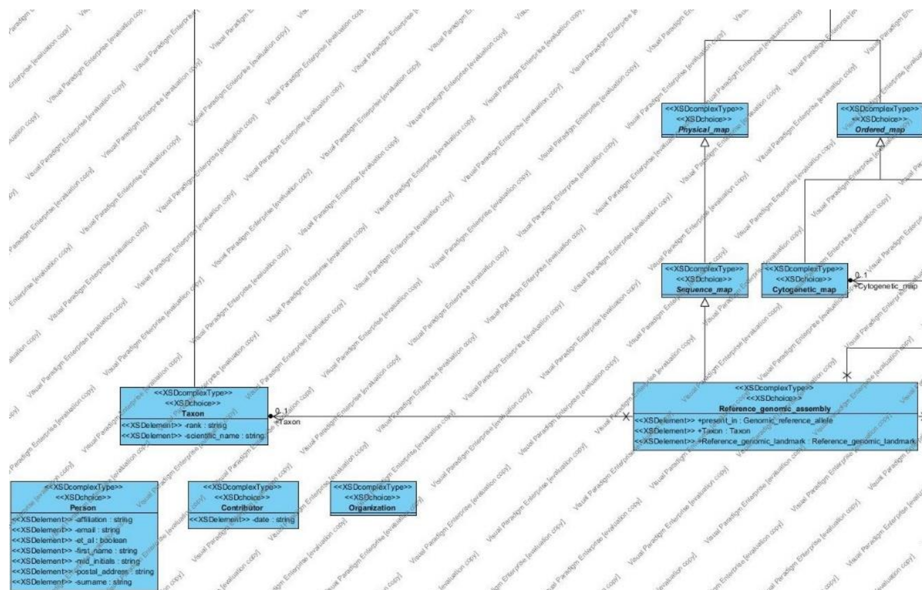
c) SNP

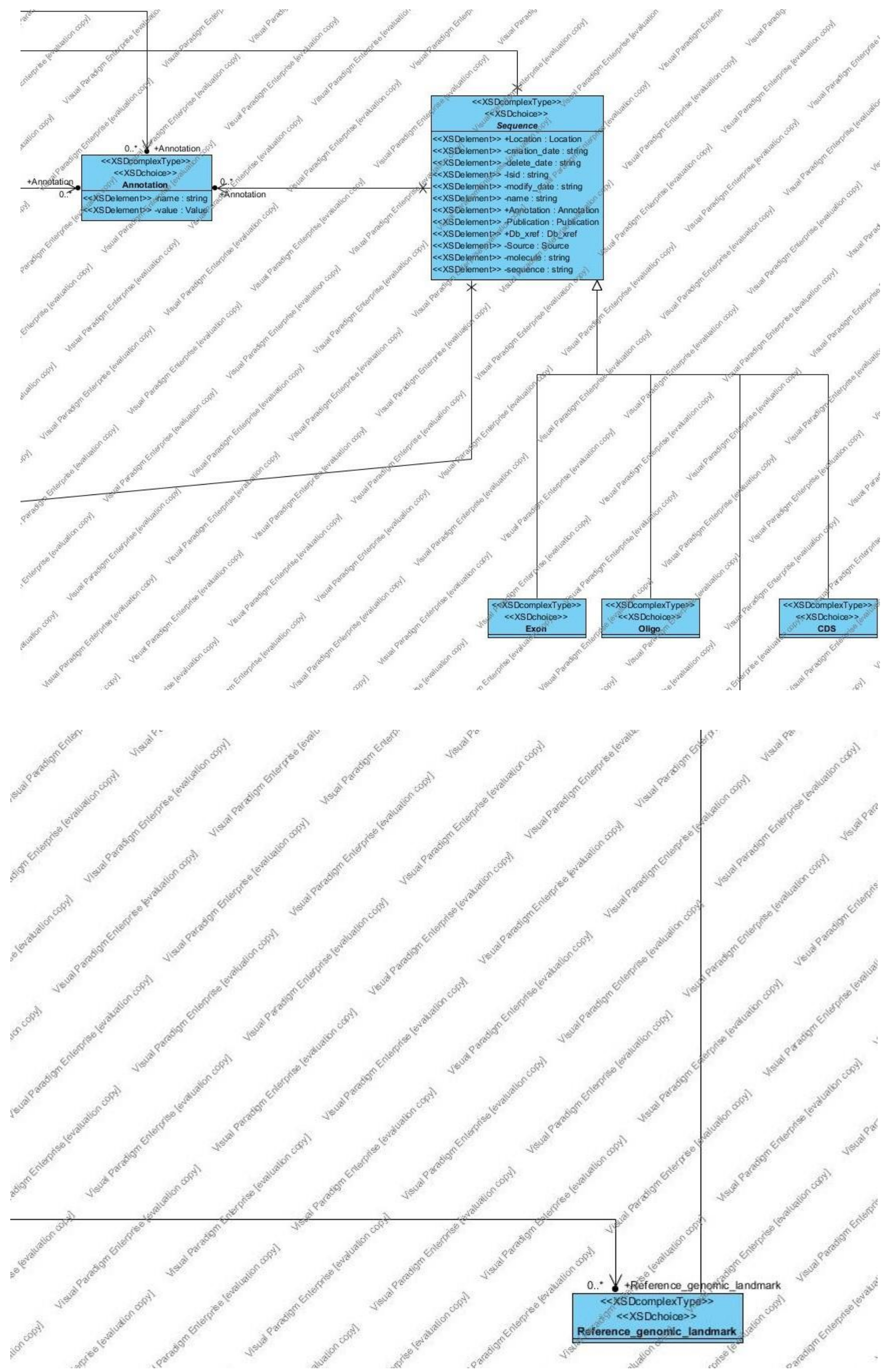




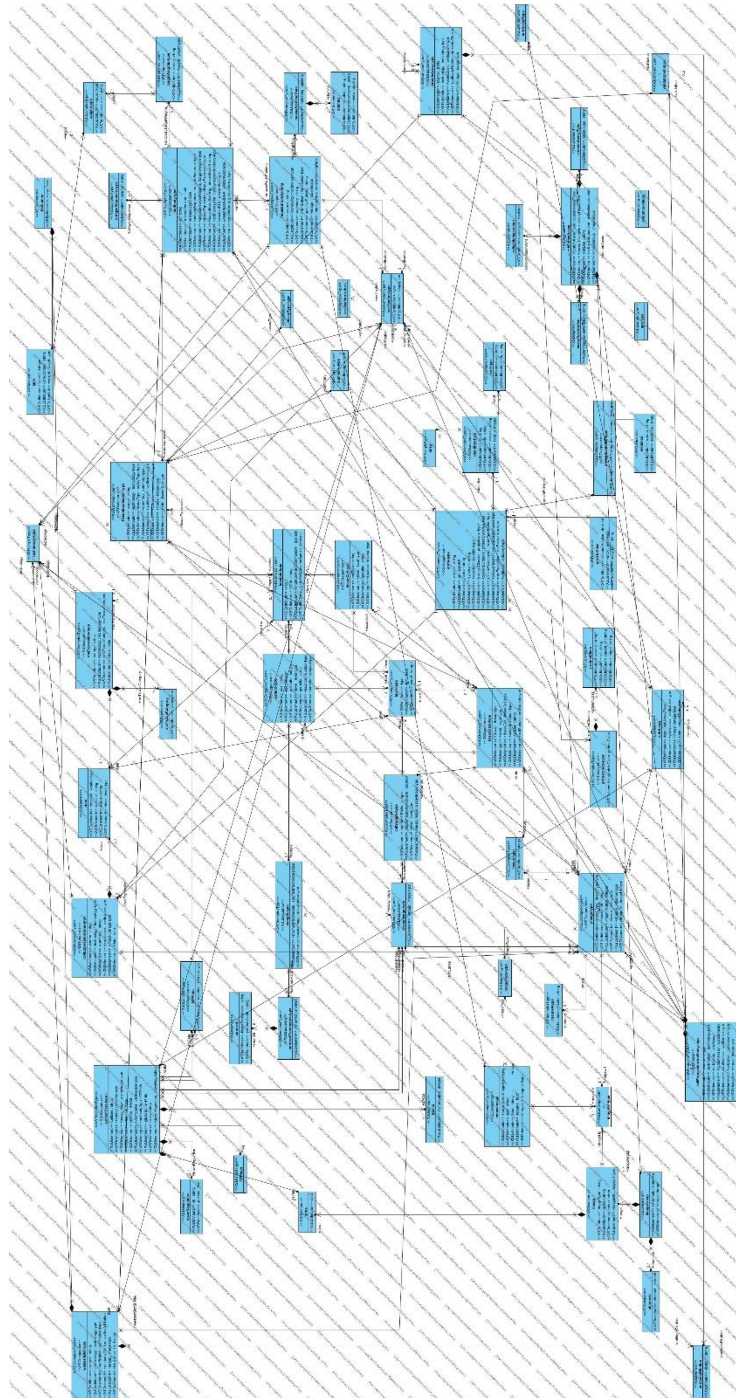
Visual Paradigm Enterprise [evaluation copy]

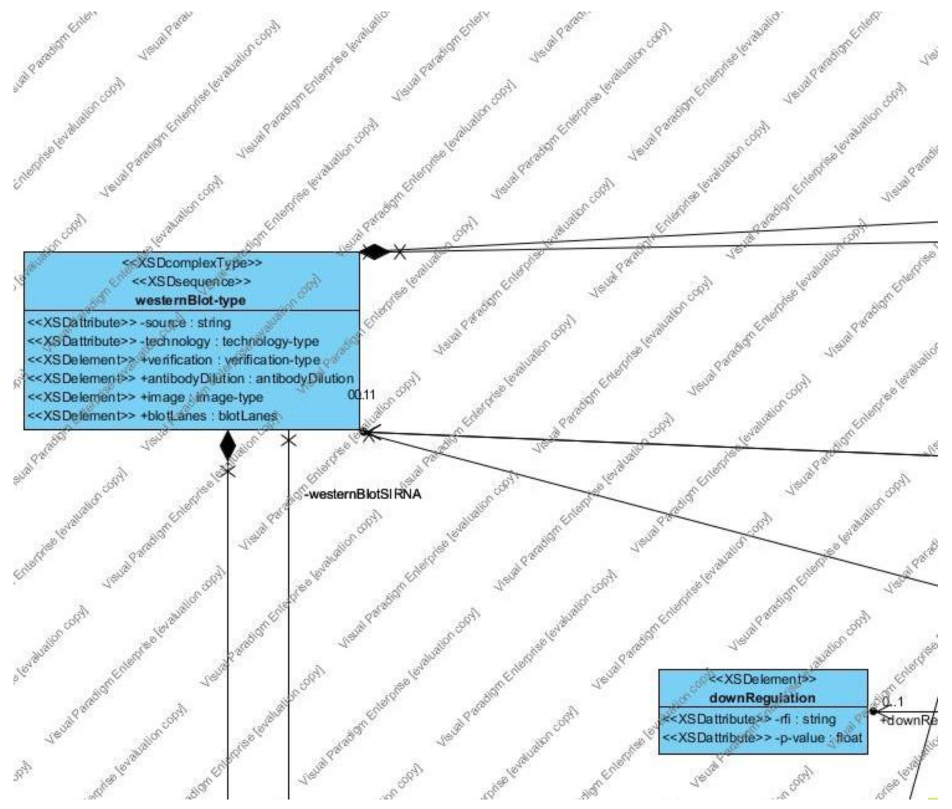


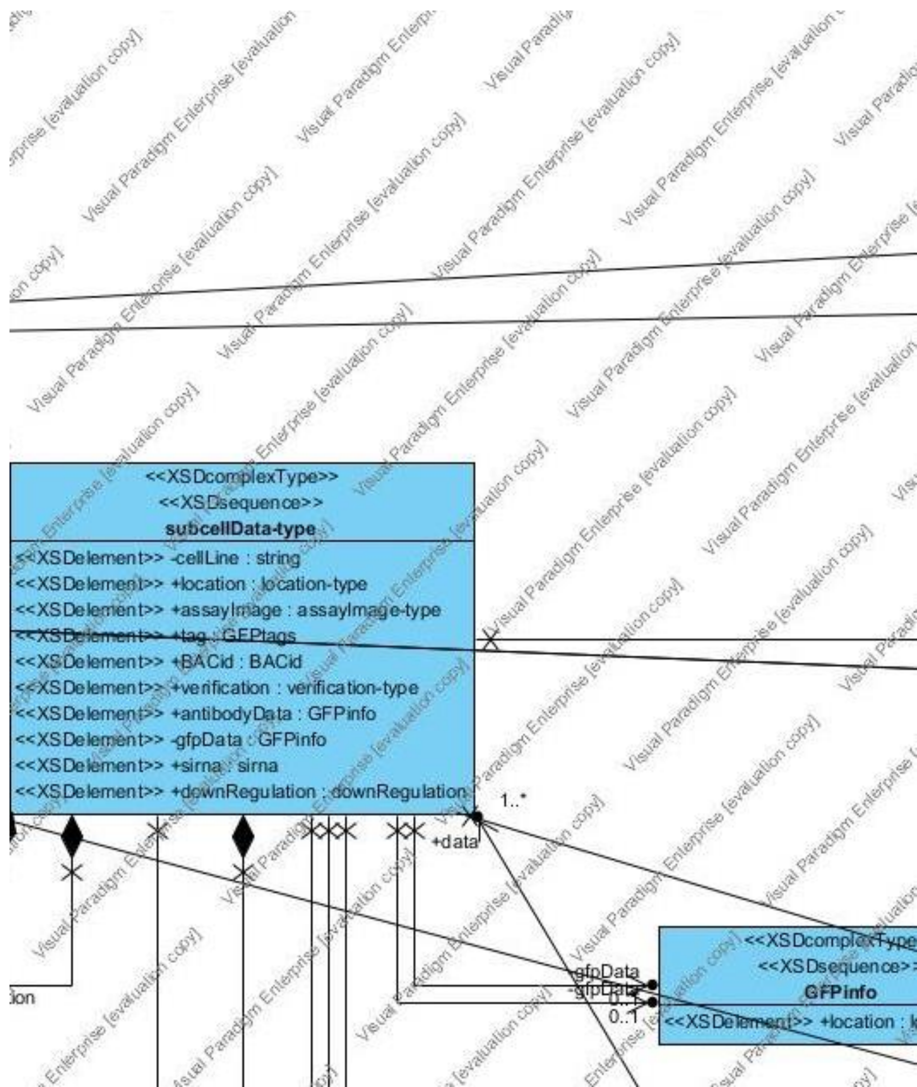



0..*
+Annotation
<<XSDcomplexType>>
<<XSDchoice>>
Annotation
<<XSDelement>> -name : string
<<XSDelement>> -value : Value
+Annotation
0..*
0..*
+Annotation
<<XSDcomplexType>>
<<XSDchoice>>
Sequence
<<XSDelement>> +Location : Location
<<XSDelement>> -creation_date : string
<<XSDelement>> -delete_date : string
<<XSDelement>> -lsid : string
<<XSDelement>> -modify_date : string
<<XSDelement>> -name : string
<<XSDelement>> +Annotation : Annotation
<<XSDelement>> -Publication : Publication
<<XSDelement>> +Db_xref : Db_xref
<<XSDelement>> -Source : Source
<<XSDelement>> -molecule : string
<<XSDelement>> -sequence : string
<<XSDcomplexType>>
<<XSDchoice>>
Exon
<<XSDcomplexType>>
<<XSDchoice>>
Oligo
<<XSDcomplexType>>
<<XSDchoice>>
CDS
0..*
+Reference_genomic_landmark
<<XSDcomplexType>>
<<XSDchoice>>
Reference_genomic_landmark

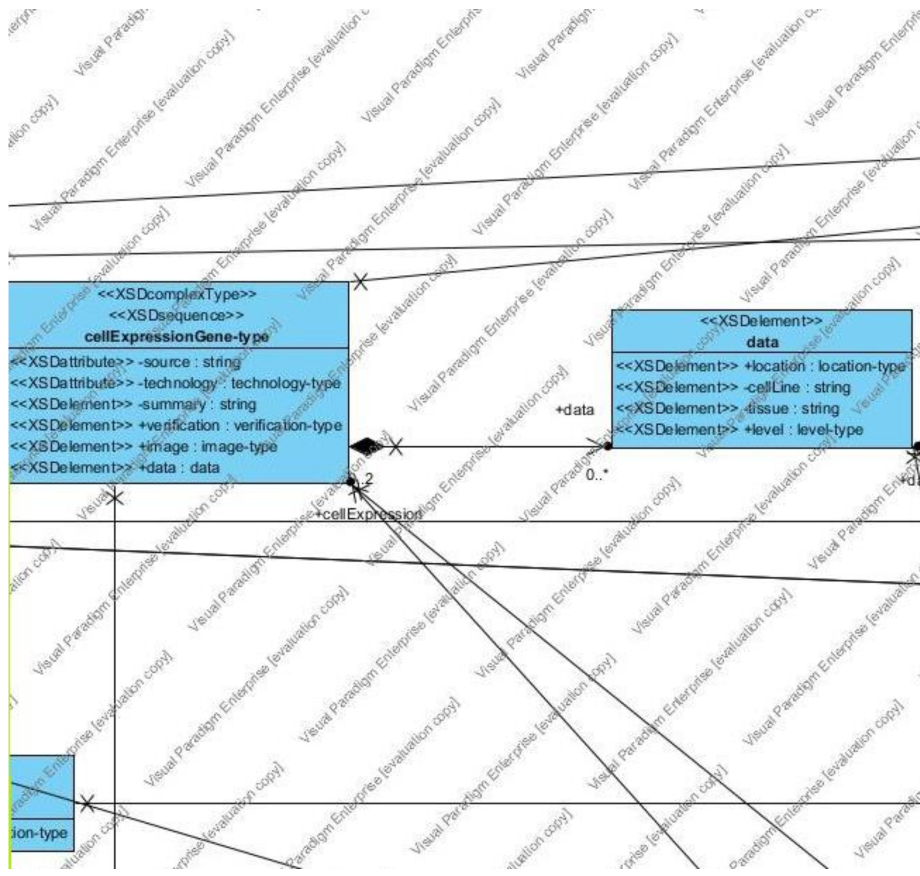
d) HPA

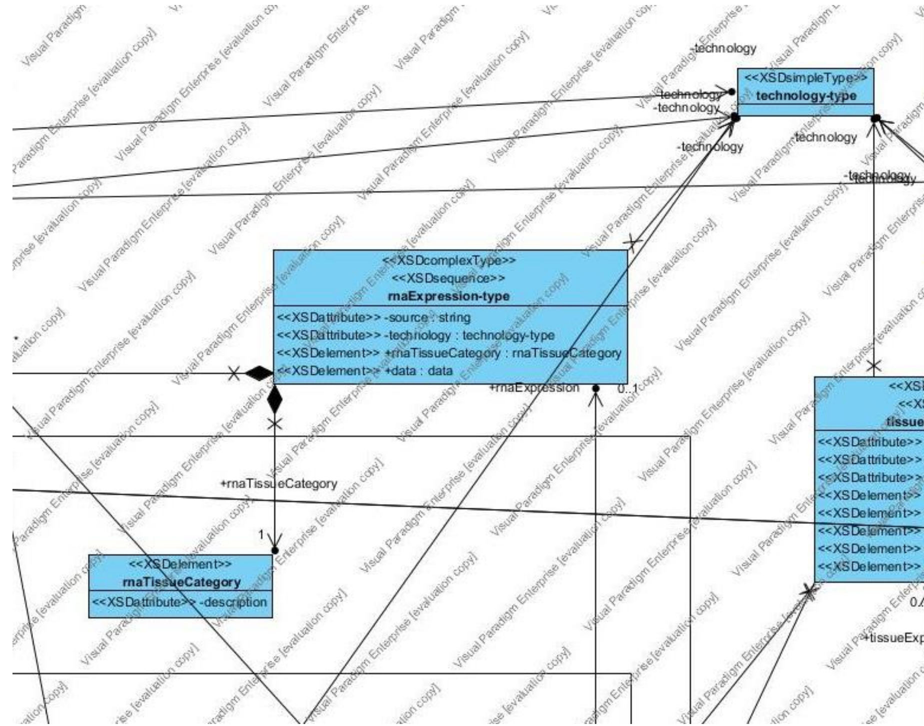


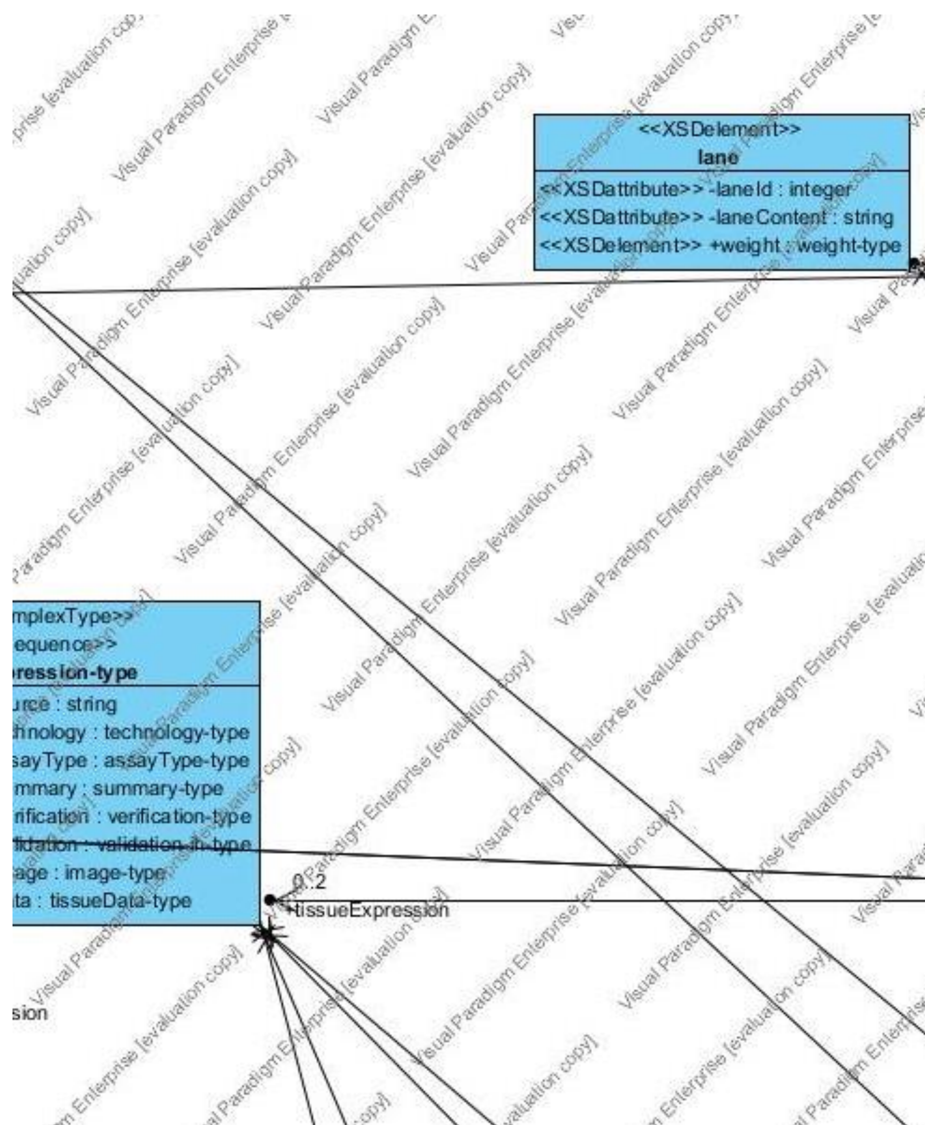

<<XSDcomplexType>>
<<XSDsequence>>
westernBlot-type
<<XSDattribute>> -source : string
<<XSDattribute>> -technology : technology-type
<<XSDelement>> +verification : verification-type
<<XSDelement>> +antibodyDilution : antibodyDilution
<<XSDelement>> +image : image-type
<<XSDelement>> +blotLanes : blotLanes
-westernBlotSIRNA
<<XSDelement>>
downRegulation
<<XSDattribute>> -rfi : string
<<XSDattribute>> -p-value : float

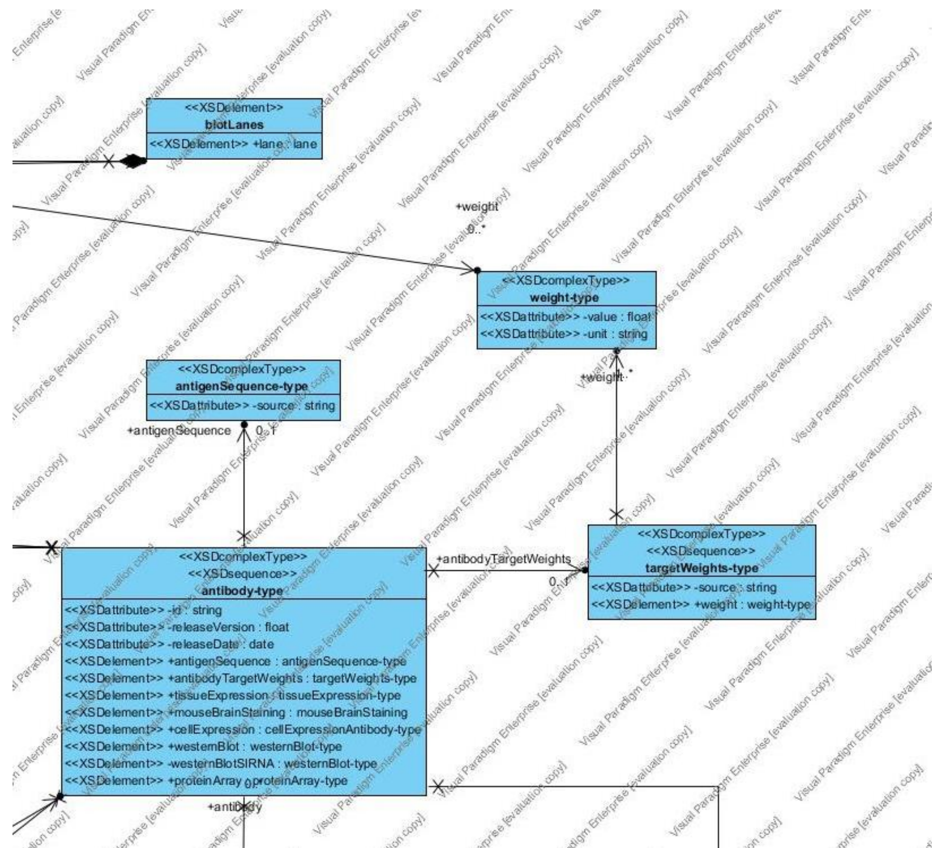


<<XSDcomplexType>>
<<XSDsequence>>
subcellData-type
<<XSDelement>> -cellLine : string
<<XSDelement>> +location : location-type
<<XSDelement>> +assayImage : assayImage-type
<<XSDelement>> +tag : GFPtags
<<XSDelement>> +BACid : BACid
<<XSDelement>> +verification : verification-type
<<XSDelement>> +antibodyData : GFPinfo
<<XSDelement>> -gfpData : GFPinfo
<<XSDelement>> +sirna : sirna
<<XSDelement>> +downRegulation : downRegulation
1..*
+data
-gfpData
0..1
<<XSDelement>> +location
GFPinfo

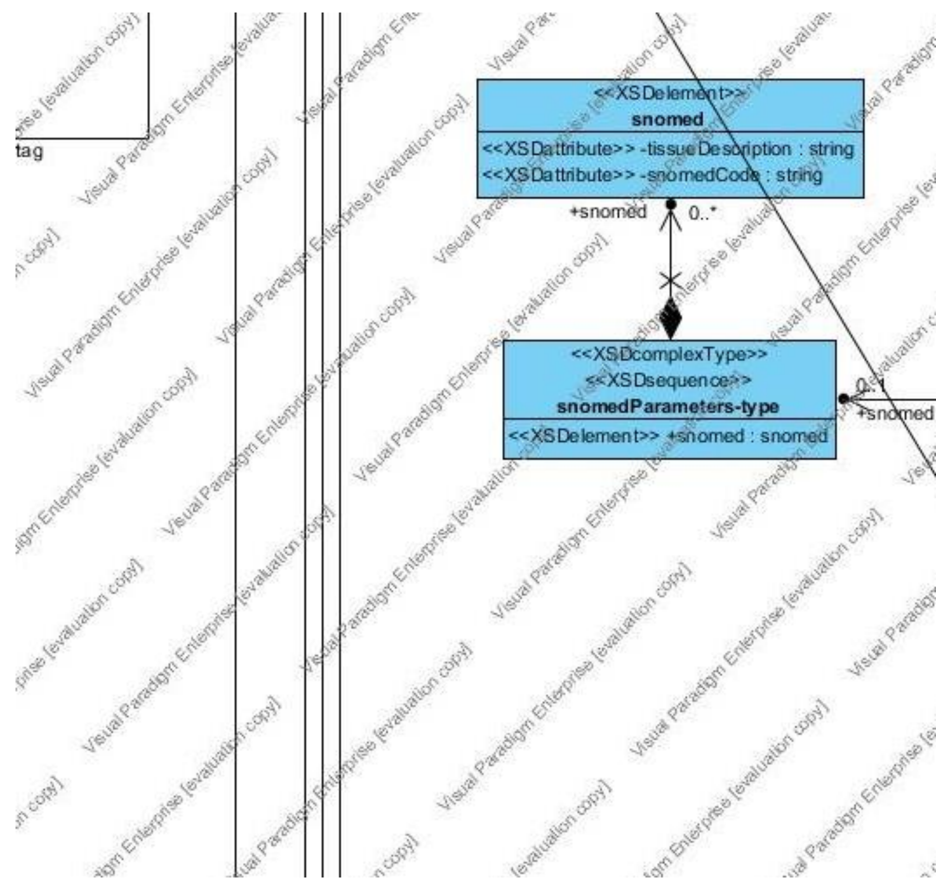

<<XSDcomplexType>>
<<XSDsequence>>
cellExpressionGene-type
<<XSDattribute>> -source : string
<<XSDattribute>> -technology : technology-type
<<XSDelement>> -summary : string
<<XSDelement>> +verification : verification-type
<<XSDelement>> +image : image-type
<<XSDelement>> +data : data
<<XSDelement>>
data
<<XSDelement>> +location : location-type
<<XSDelement>> -cellLine : string
<<XSDelement>> -tissue : string
<<XSDelement>> +level : level-type
+data
0..*
+cellExpression


-technology
<<XSDsimpleType>>
technology-type
-technology
-technology
-technology
-technology
-technology
<<XSDcomplexType>>
<<XSDsequence>>
rnaExpression-type
<<XSDattribute>> -source : string
<<XSDattribute>> -technology : technology-type
<<XSDelement>> +rnaTissueCategory : rnaTissueCategory
<<XSDelement>> +data : data
+rnaExpression
0..1
+rnaTissueCategory
1
<<XSDelement>>
rnaTissueCategory
<<XSDattribute>> -description
+tissueExpr

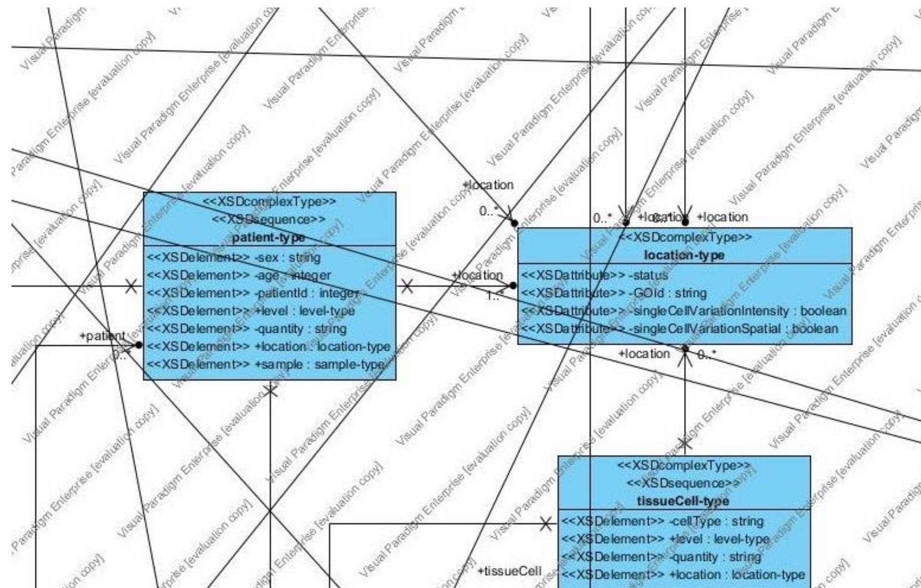

<<XSDelement>>
lane
<<XSDattribute>> -laneId : integer
<<XSDattribute>> -laneContent : string
<<XSDelement>> +weight : weight-type
mplexType>>
equence>>
ression-type
urce : string
hnology : technology-type
sayType : assayType-type
mmary : summary-type
rification : verification-type
lidation : validation-in-type
age : image-type
ta : tissueData-type
0..2
+tissueExpression
sion

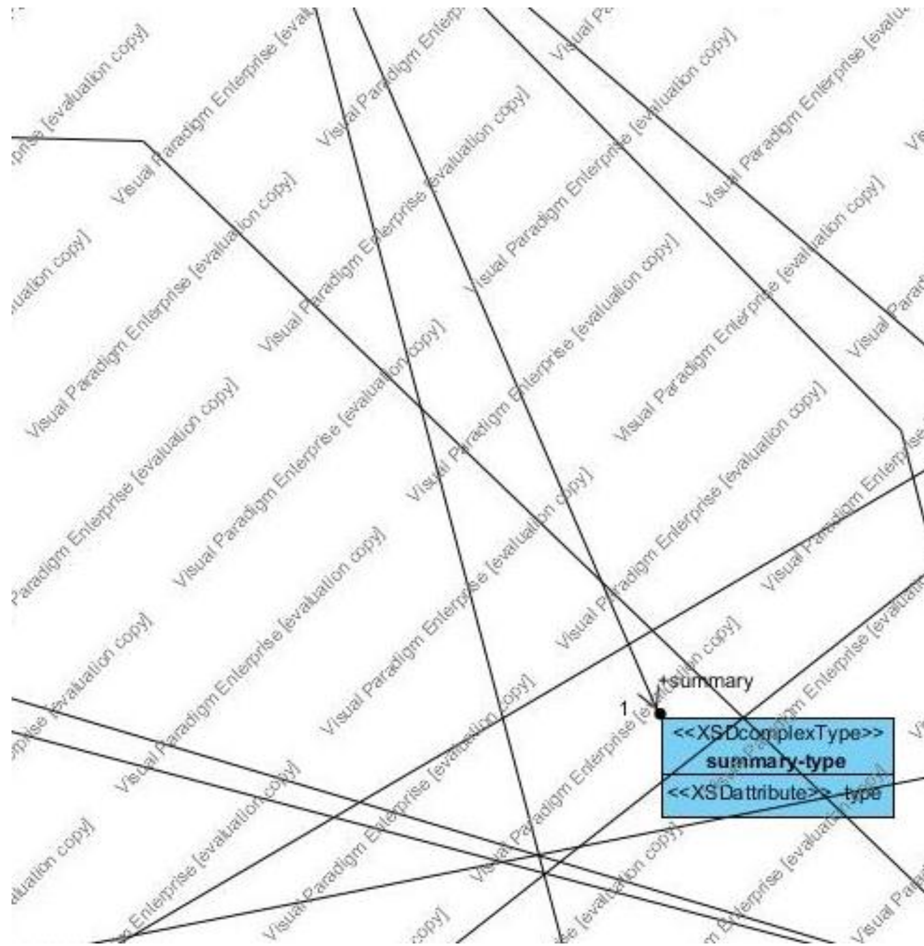

<<XSDelement>>
blotLanes
<<XSDelement>> +lane : lane
+weight
0..*
<<XSDcomplexType>>
weight-type
<<XSDattribute>> -value : float
<<XSDattribute>> -unit : string
+weight
0..*
<<XSDcomplexType>>
antigenSequence-type
<<XSDattribute>> -source : string
+antigenSequence
0..1
<<XSDcomplexType>>
<<XSDsequence>>
antibody-type
<<XSDattribute>> -id : string
<<XSDattribute>> -releaseVersion : float
<<XSDattribute>> -releaseDate : date
<<XSDelement>> +antigenSequence : antigenSequence-type
<<XSDelement>> +antibodyTargetWeights : targetWeights-type
<<XSDelement>> +tissueExpression : tissueExpression-type
<<XSDelement>> +mouseBrainStaining : mouseBrainStaining
<<XSDelement>> +cellExpression : cellExpressionAntibody-type
<<XSDelement>> +westernBlot : westernBlot-type
<<XSDelement>> -westernBlotSIRNA : westernBlot-type
<<XSDelement>> +proteinArray : proteinArray-type
+antibody
+antibodyTargetWeights
0..1
<<XSDcomplexType>>
<<XSDsequence>>
targetWeights-type
<<XSDattribute>> -source : string
<<XSDelement>> +weight : weight-type

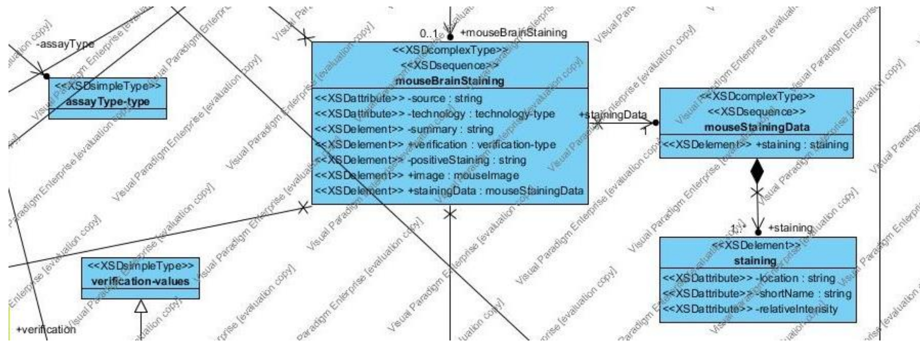
<<XSDsimpleType>>
GFPtags
+sirna
1
<<XSDelement>>
sirna
<<XSDattribute>> -sirna
0..1
+sirna

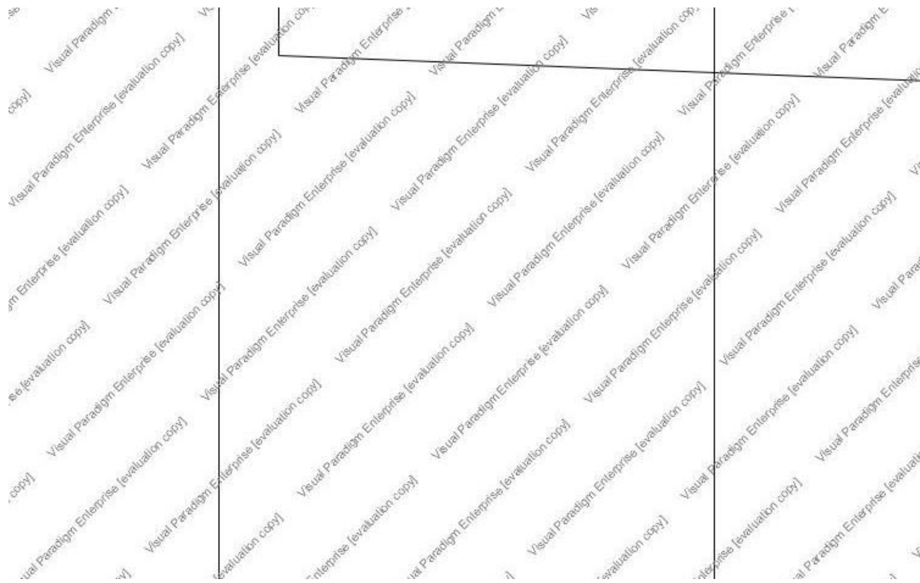

tag
<<XSDelement>>
snomed
<<XSDattribute>> -tissueDescription : string
<<XSDattribute>> -snomedCode : string
+snomed
0..*
<<XSDcomplexType>>
<<XSDsequence>>
snomedParameters-type
<<XSDelement>> +snomed : snomed
0..1
+snomed

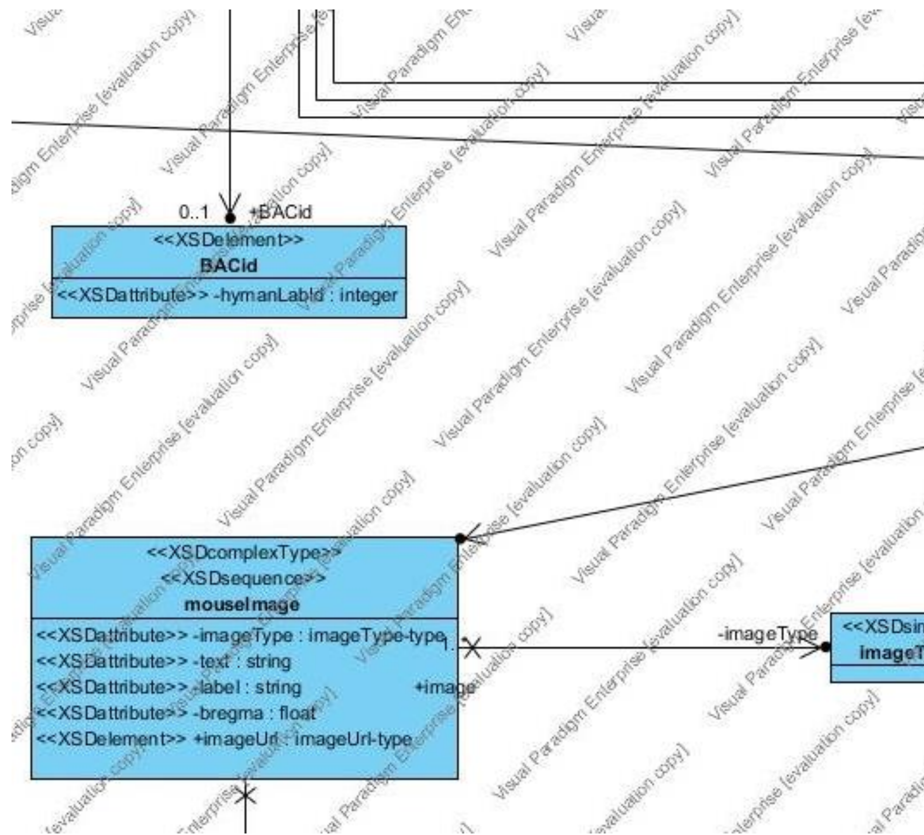
<<XSDcomplexType>>
<<XSDsequence>>
sample-type
<<XSDelement>> +snomedParameters : snomedParameters-type
<<XSDelement>> +assayImage : assayImage-type
1..*
+sample

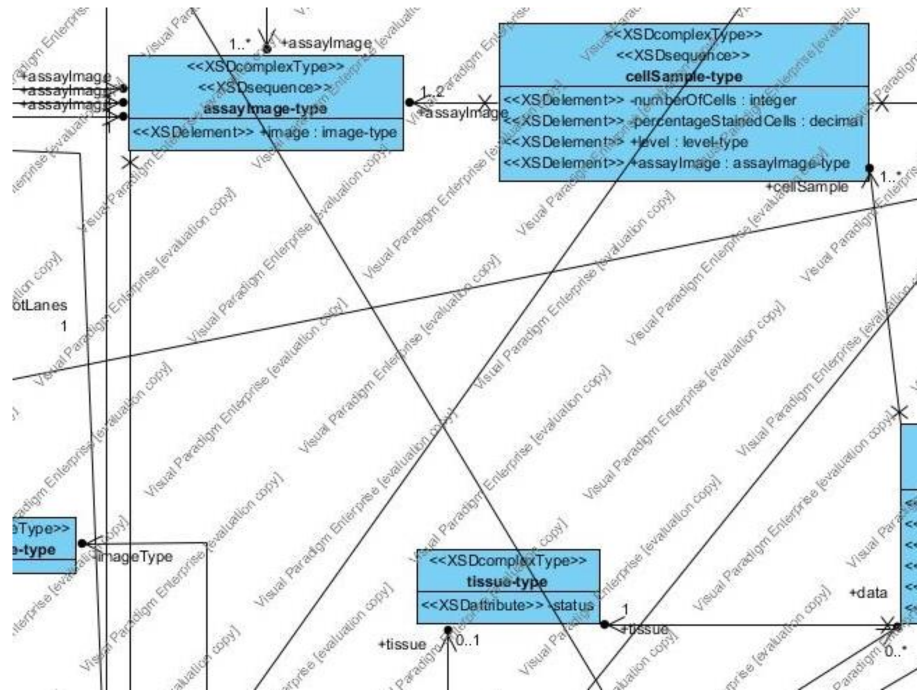

+location
0..*
<<XSDcomplexType>>
<<XSDsequence>>
patient-type
<<XSDelement>> -sex : string
<<XSDelement>> -age : integer
<<XSDelement>> -patientId : integer
<<XSDelement>> +level : level-type
<<XSDelement>> -quantity : string
<<XSDelement>> +location : location-type
<<XSDelement>> +sample : sample-type
+patient
+location
<<XSDcomplexType>>
location-type
<<XSDattribute>> -status
<<XSDattribute>> -GOId : string
<<XSDattribute>> -singleCellVariationIntensity : boolean
<<XSDattribute>> -singleCellVariationSpatial : boolean
+location
0..*
<<XSDcomplexType>>
<<XSDsequence>>
tissueCell-type
<<XSDelement>> -cellType : string
<<XSDelement>> +level : level-type
<<XSDelement>> -quantity : string
<<XSDelement>> +location : location-type
+tissueCell

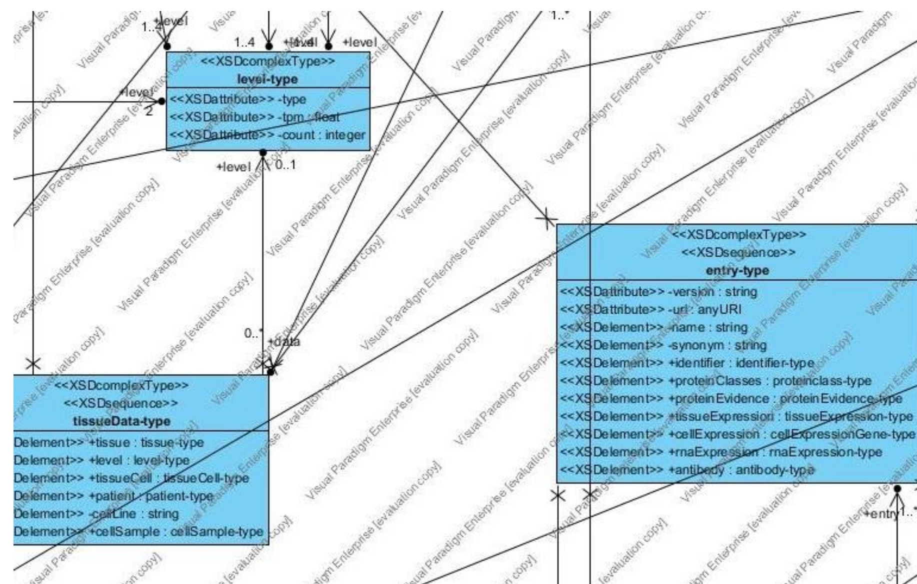

+summary
1
<<XSDcomplexType>>
summary-type
<<XSDattribute>> type

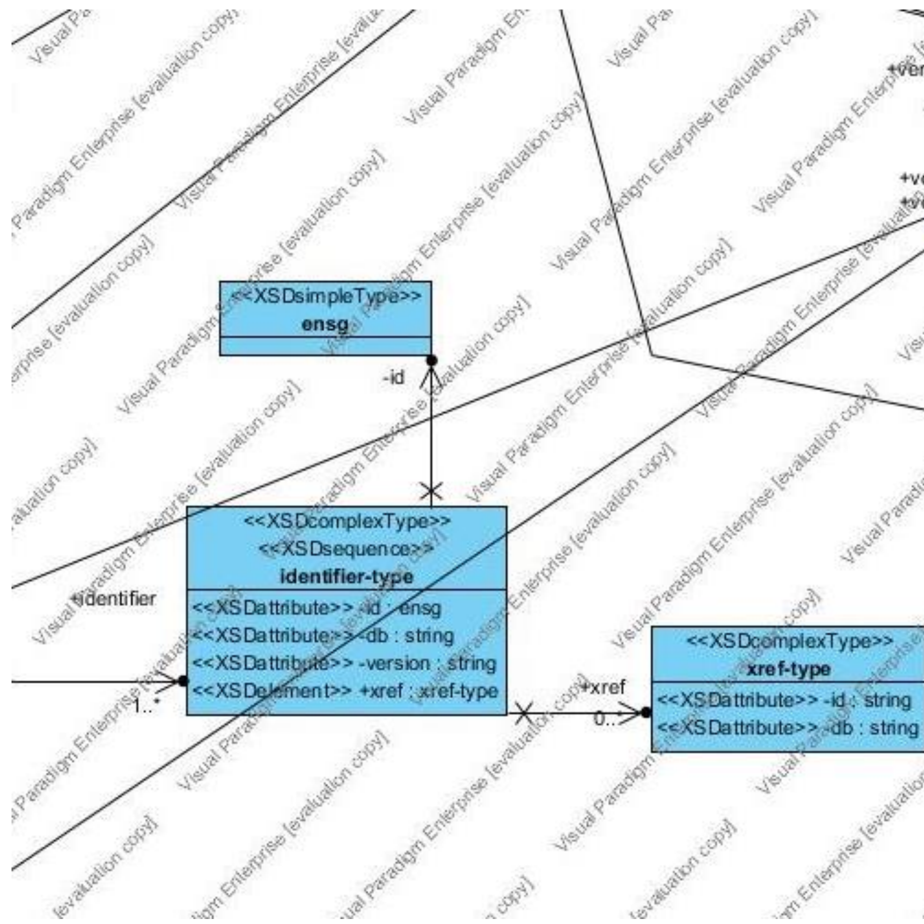

-assayType
<<XSDsimpleType>>
assayType-type
0..1
+mouseBrainStaining
<<XSDcomplexType>>
<<XSDsequence>>
mouseBrainStaining
<<XSDattribute>> -source : string
<<XSDattribute>> -technology : technology-type
<<XSDelement>> -summary : string
<<XSDelement>> +verification : verification-type
<<XSDelement>> -positiveStaining : string
<<XSDelement>> +image : mouseImage
<<XSDelement>> +stainingData : mouseStainingData
+stainingData
1
<<XSDcomplexType>>
<<XSDsequence>>
mouseStainingData
<<XSDelement>> +staining : staining
+staining
<<XSDelement>>
staining
<<XSDattribute>> -location : string
<<XSDattribute>> -shortName : string
<<XSDattribute>> -relativeIntensity
<<XSDsimpleType>>
verification-values
+verification

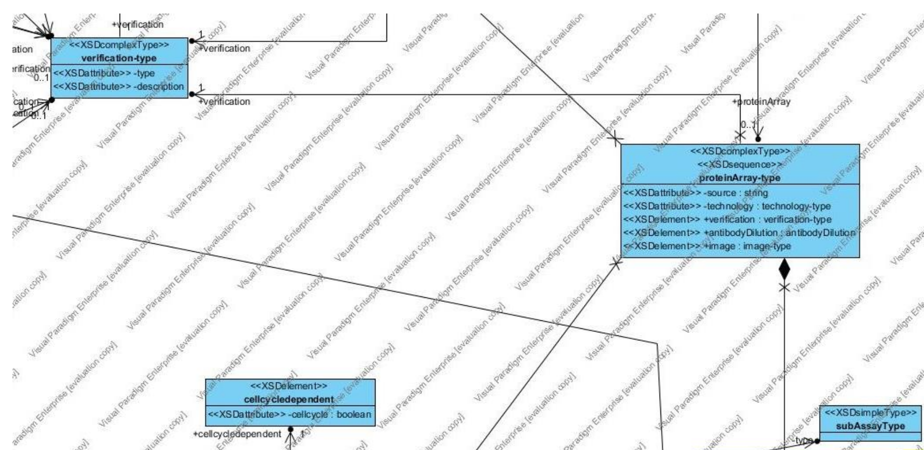


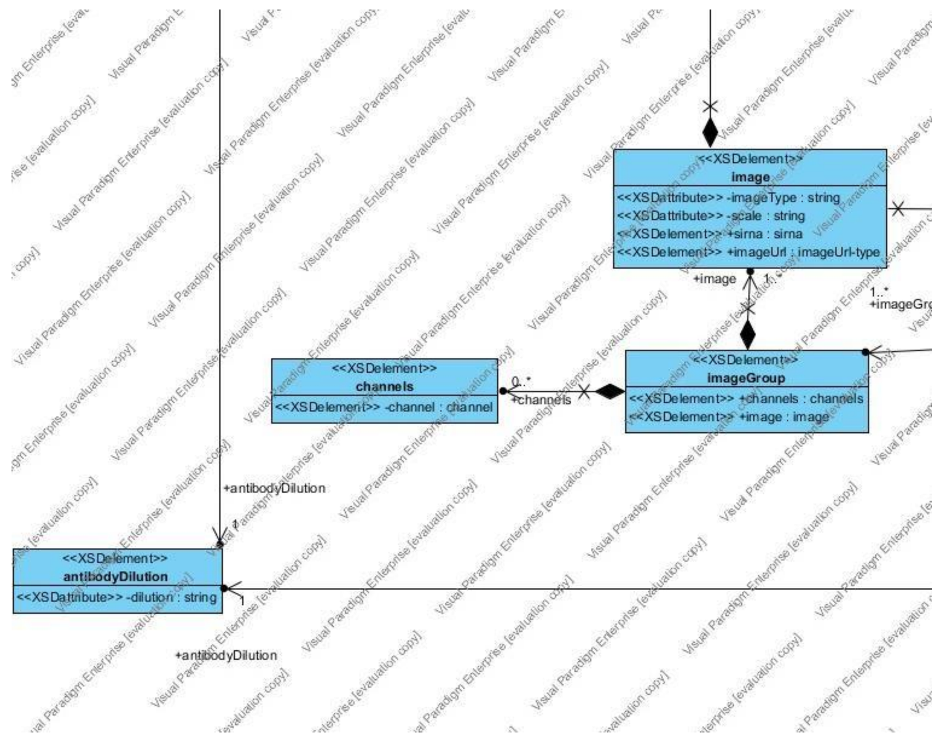

0..1
+BACId
<<XSDelement>>
BACId
<<XSDattribute>> -hymanLabId : integer
<<XSDcomplexType>>
<<XSDsequence>>
mouseImage
<<XSDattribute>> -imageType : imageType-type
<<XSDattribute>> -text : string
<<XSDattribute>> -label : string
<<XSDattribute>> -bregma : float
<<XSDelement>> +imageUrl : imageUrl-type
+image
-imageType

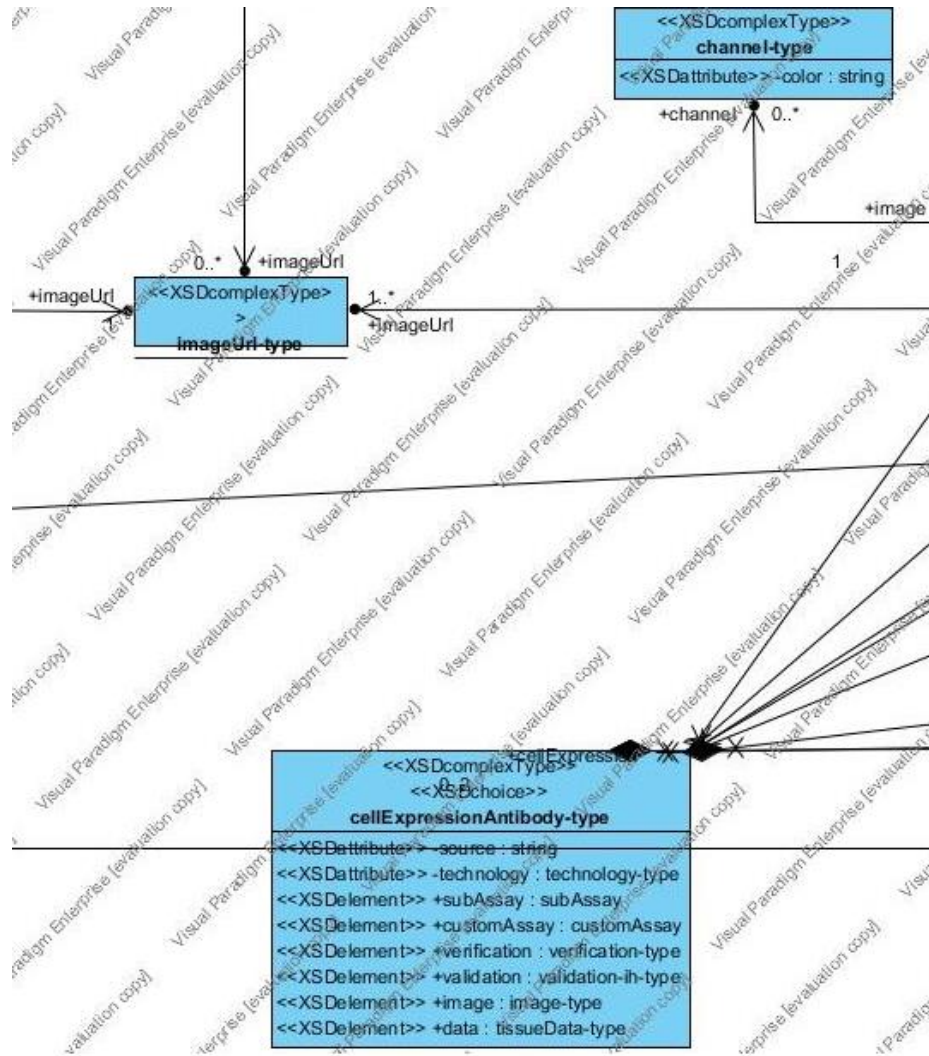

<<XSDcomplexType>>
<<XSDsequence>>
assayImage-type
<<XSDelement>> +image : image-type
<<XSDcomplexType>>
<<XSDsequence>>
cellSample-type
<<XSDelement>> -numberOfCells : integer
<<XSDelement>> -percentageStainedCells : decimal
<<XSDelement>> +level : level-type
<<XSDelement>> +assayImage : assayImage-type
<<XSDcomplexType>>
tissue-type
<<XSDattribute>> -status
+assayImage
+cellSample
+tissue
+data
+imageType

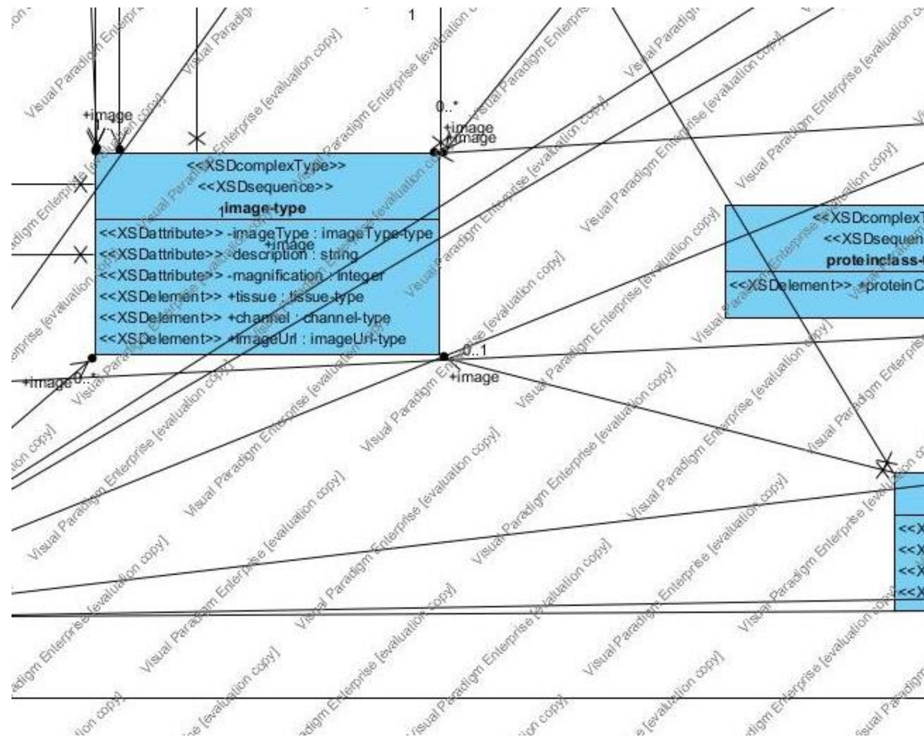

<<XSDcomplexType>>
level-type
<<XSDattribute>> -type
<<XSDattribute>> -tpm : float
<<XSDattribute>> -count : integer
<<XSDcomplexType>>
<<XSDsequence>>
tissueData-type
Delement>> +tissue : tissue-type
Delement>> +level : level-type
Delement>> +tissueCell : tissueCell-type
Delement>> +patient : patient-type
Delement>> -cellLine : string
Delement>> +cellSample : cellSample-type
<<XSDcomplexType>>
<<XSDsequence>>
entry-type
<<XSDattribute>> -version : string
<<XSDattribute>> -url : anyURI
<<XSDelement>> -name : string
<<XSDelement>> -synonym : string
<<XSDelement>> +identifier : identifier-type
<<XSDelement>> +proteinClasses : proteinclass-type
<<XSDelement>> +proteinEvidence : proteinEvidence-type
<<XSDelement>> +tissueExpression : tissueExpression-type
<<XSDelement>> +cellExpression : cellExpressionGene-type
<<XSDelement>> +rnaExpression : rnaExpression-type
<<XSDelement>> +antibody : antibody-type
+level
+data
+entry


<<XSDsimpleType>>
ensg
-id
<<XSDcomplexType>>
<<XSDsequence>>
identifier-type
<<XSDattribute>> -id : ensg
<<XSDattribute>> -db : string
<<XSDattribute>> -version : string
<<XSDelement>> +xref : xref-type
+identifier
1..*
+xref
0..*
<<XSDcomplexType>>
xref-type
<<XSDattribute>> -id : string
<<XSDattribute>> -db : string

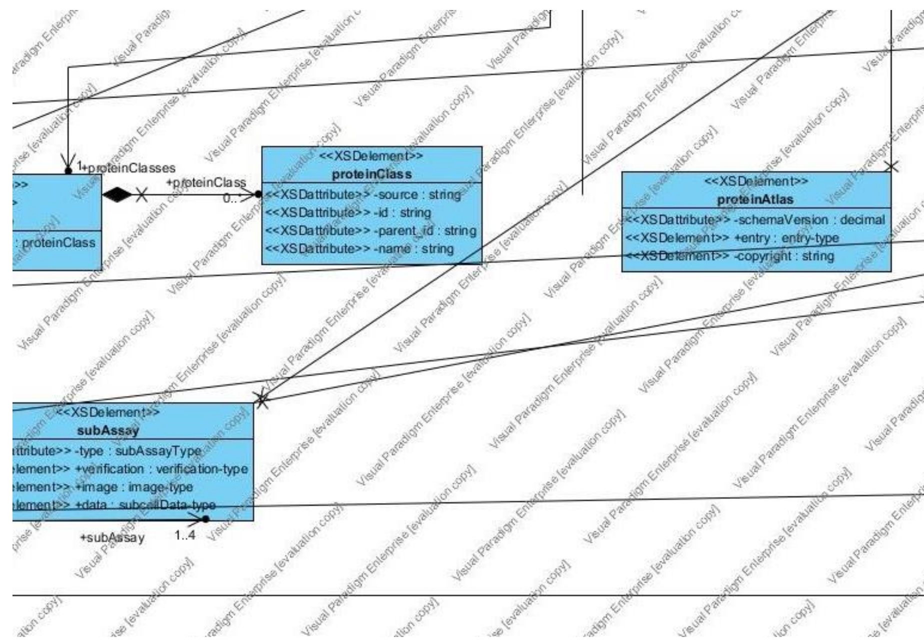

+verification
<<XSDcomplexType>>
verification-type
<<XSDattribute>> -type
<<XSDattribute>> -description
+proteinArray
<<XSDcomplexType>>
<<XSDsequence>>
proteinArray-type
<<XSDattribute>> -source : string
<<XSDattribute>> -technology : technology-type
<<XSDelement>> +verification : verification-type
<<XSDelement>> +antibodyDilution : antibodyDilution
<<XSDelement>> +image : image-type
<<XSDelement>>
cellcycledependent
<<XSDattribute>> -cellcycle : boolean
+cellcycledependent
<<XSDsimpleType>>
subAssayType
+type


<<XSDelement>>
image
<<XSDattribute>> -imageType : string
<<XSDattribute>> -scale : string
<<XSDelement>> +sirna : sirna
<<XSDelement>> +imageUrl : imageUrl-type
+image
1..*
+imageGro
<<XSDelement>>
channels
<<XSDelement>> -channel : channel
0..*
+channels
<<XSDelement>>
imageGroup
<<XSDelement>> +channels : channels
<<XSDelement>> +image : image
+antibodyDilution
1
<<XSDelement>>
antibodyDilution
<<XSDattribute>> -dilution : string
1
+antibodyDilution


<<XSDcomplexType>>
channel-type
<<XSDattribute>> -color : string
+channel
0..*
+image
0..*
+imageUrl
+imageUrl
1
<<XSDcomplexType>>
imageUrl-type
1..*
+imageUrl
1
+cellExpression
0..1
<<XSDcomplexType>>
<<XSDchoice>>
cellExpressionAntibody-type
<<XSDattribute>> -source : string
<<XSDattribute>> -technology : technology-type
<<XSDelement>> +subAssay : subAssay
<<XSDelement>> +customAssay : customAssay
<<XSDelement>> +verification : verification-type
<<XSDelement>> +validation : validation-ih-type
<<XSDelement>> +image : image-type
<<XSDelement>> +data : tissueData-type

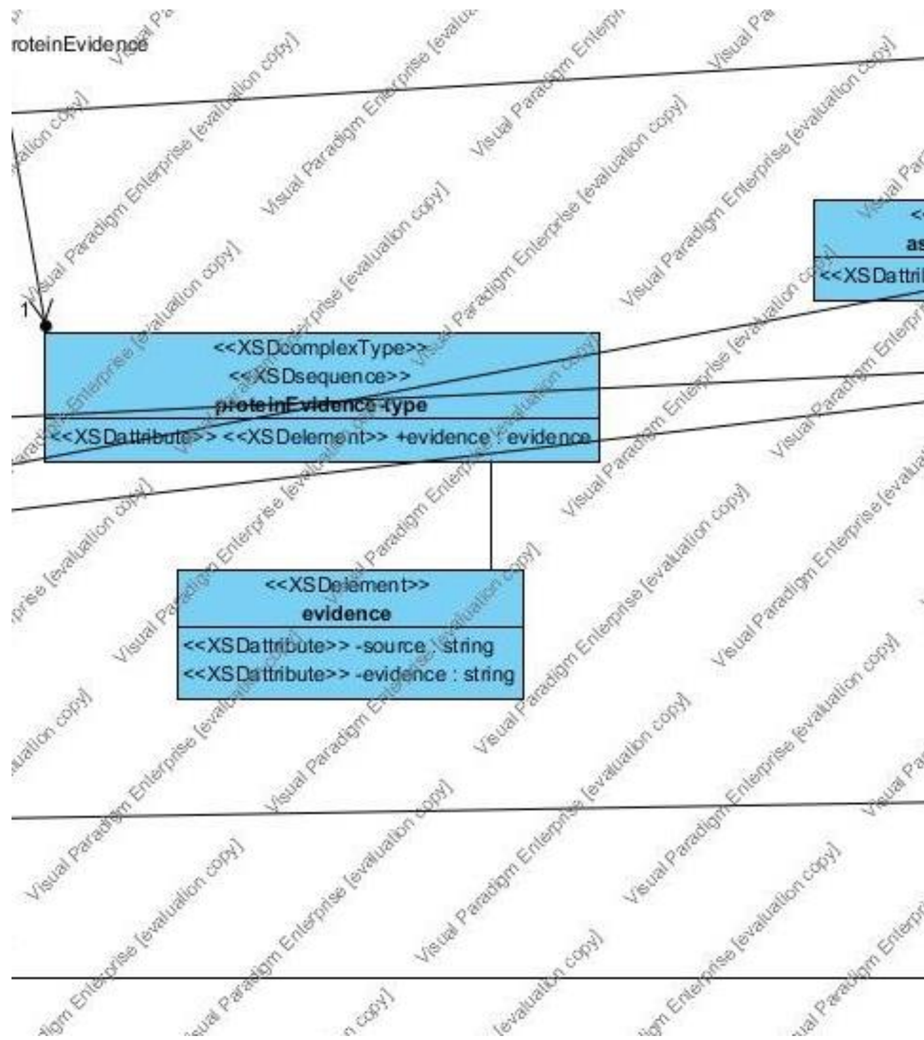


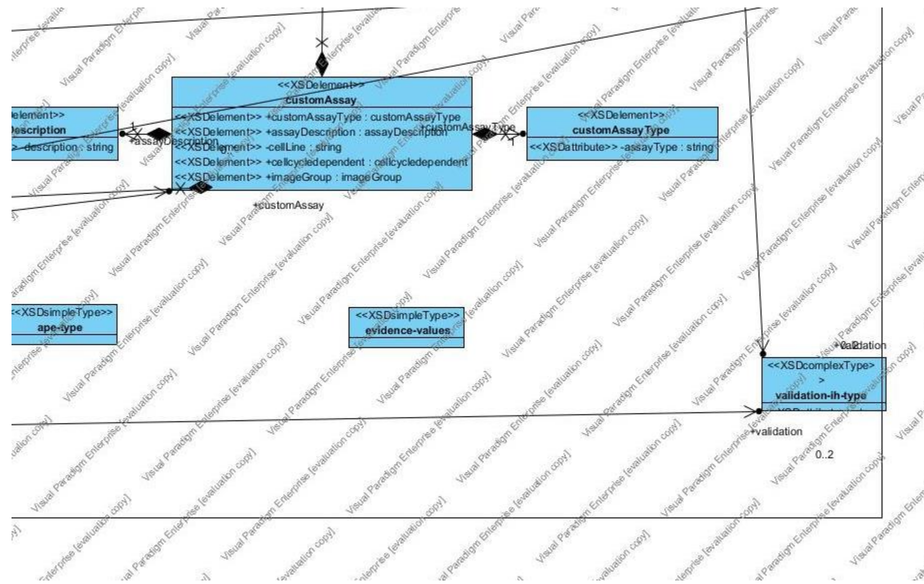
<<XSDcomplexType>>
<<XSDsequence>>
image-type
<<XSDattribute>> -imageType : imageType-type
<<XSDattribute>> -description : string
<<XSDattribute>> -magnification : integer
<<XSDelement>> +tissue : tissue-type
<<XSDelement>> +channel : channel-type
<<XSDelement>> +imageUrl : imageUrl-type
+image
0..*
0..1



<<XSDelement>>
proteinClass
<<XSDattribute>> -source : string
<<XSDattribute>> -id : string
<<XSDattribute>> -parent_id : string
<<XSDattribute>> -name : string
+proteinClasses
+proteinClass
<<XSDelement>>
proteinAtlas
<<XSDattribute>> -schemaVersion : decimal
<<XSDelement>> +entry : entry-type
<<XSDelement>> -copyright : string
<<XSDelement>>
subAssay
+subAssay
1..4

JOHANNES KEPLER UNIVERSITY LINZ





11. Appendix B: Program Code

a) Python code to create the INSERT statements

```
#!/usr/bin/env python
# Author: Anna Saibold
import sys
import re
infile = open("Alternative1.xml", 'r')  # read
entryFile = open("INSERT_GeneEntry.sql", 'a')
aliasFile = open("INSERT_GeneAlias.sql", 'a')
sentenceFile = open("INSERT_Sentence.sql", 'a')
statementFile = open("INSERT_Statement.sql", 'a')
rolesFile = open("INSERT_Roles.sql", 'a')
proleFile = open("INSERT_PrimaryRole.sql", 'a')
oroleFile = open("INSERT_OtherRole.sql", 'a')
evidenceFile = open("INSERT_EvidenceCode.sql", 'a')
commentFile = open("INSERT_Comments.sql", 'a')
wholeF = infile.read()
#print(wholeF)
entry=wholeF.split("</GeneEntry>")
for e in range(0, len(entry)-1):
    #GeneEntry
    HUGO = re.findall(" HUGOGeneSymbol=\"(.*?)\">", entry[e])
    gStatus = re.findall("<geneStatus>(.*?)</geneStatus>", entry[e])
    hgnc = re.findall("<hgncID>(.*?)</hgncID>", entry[e])
    if not hgnc:
        hgnc.append("")
    locuslink  = re.findall("<locusLinkID>(.*?)</locusLinkID>", entry[e])
    if not locuslink:
        locuslink.append("")
    genbank = re.findall("<genbankAccession>(.*?)</genbankAccession>", entry[e])
    if not genbank:
        genbank.append("")
    refseq = re.findall("<refSeqID>(.*?)</refSeqID>", entry[e])
    if not refseq:
        refseq.append("")
    uniprot = re.findall("<uniProtID>(.*?)</uniProtID>", entry[e])
    if not uniprot:
        uniprot.append("")
    entryFile.write("INSERT INTO GeneEntry (HUGOGeneSymbol, geneStatus, hgncID, lo-
cusLinkID, genbankAccession, refSeqID, uniProtID) VALUES
(\'"+str(HUGO[0])+"\',\'"+str(gStatus[0])+"\',"+str(hgnc[0])+","+str(locuslink[0])+",\'"+
str(genbank[0])+"\',\'"+str(refseq[0])+"\',\'"+str(uniprot[0])+"\');\n")
    #GeneAlias
    gaID = re.findall(" geneAliasID=\"(.*?)\">", entry[e])
    GeneA = re.findall("<GeneAlias.*>(.*?)</GeneAlias>", entry[e])
    for id in range(0, len(gaID)):
        aliasFile.write("INSERT INTO GeneAlias (geneAliasID, geneAlias, HUGOGeneSymbol)
VALUES ("+str(gaID[id])+",\'"+str(GeneA[id])+"\',\'"+str(HUGO[0])+"\');\n")
    #Sentence
    sentenceID = re.findall(" sentenceID=\"(.*?)\">", entry[e])
    sentence = re.split("</Sentence>", entry[e])
    for s in range(0, len(sentenceID)):
        sStatus = re.findall("<sentenceStatus>(.*?)</sentenceStatus>", sentence[s])
        pubmed = re.findall("<pubMedID>(.*?)</pubMedID>", sentence[s])
        if not pubmed:
            pubmed.append("")
        organism = re.findall("<organism>(.*?)</organism>", sentence[s])
        nIndicator = re.findall("<negationIndicator>(.*?)</negationIndicator>", sen-
tence[s])
        cIndicator = re.findall("<cellLineIndicator>(.*?)</cellLineIndicator>", sen-
tence[s])
```

```
        mGeneT = re.findall("<matchedGeneTerm>(.*?)</matchedGeneTerm>", sentence[s])
        NCIGeneConceptCode = re.findall("<NCIGeneConceptCode>(.*?)</NCIGeneConceptCode>",
sentence[s])
        if not NCIGeneConceptCode:
            NCIGeneConceptCode.append("")
        mDiseaseT = re.findall("<matchedDiseaseTerm>(.*?)</matchedDiseaseTerm>", sen-
tence[s])
        NCIDiseaseConceptCode =
re.findall("<NCIDiseaseConceptCode>(.*?)</NCIDiseaseConceptCode>", sentence[s])
        if not NCIDiseaseConceptCode:
            NCIDiseaseConceptCode.append("")
        sentenceFile.write("INSERT INTO Sentence (sentenceID, sentenceStatus, pubMedID,
organism, negationIndicator, cellLineIndicator, matchedGeneTerm, NCIGeneConceptCode,
matchedDiseaseTerm, NCIDiseaseConceptCode, HUGOGeneSymbol) "
                            "VALUES
("+str(sentenceID[s])+",\'"+str(sStatus[0])+"\',"+str(pubmed[0])+",\'"+str(organism[0])+"
\',\'"+str(nIndicator[0])+"\',\'"+str(cIndicator[0])+"\',\'"+str(mGeneT[0])+"\',\'"+str(N
CIGene-
Con-
ceptCode[0])+"\',\'"+str(mDiseaseT[0])+"\',\'"+str(NCIDiseaseConceptCode[0])+"\',\'"+str(
HUGO[0])+"\');\n")
        #Statement
        statemID = re.findall(" statementID=\"(.*?)\">", sentence[s])
        statem = re.findall("<Statement.*>(.*?)</Statement>", sentence[s])
        for id in range(0, len(statemID)):
            statementFile.write("INSERT INTO Statement (statementID, statement, senten-
ceID) VALUES ("+str(statemID[id])+",\'"+str(statem[id])+"\',"+str(sentenceID[s])+");\n")
        #Roles
        rolesID = re.findall(" rolesID=\"(.*?)\">", sentence[s])
        role = re.split("</Roles>", sentence[s])
        for r in range(0, len(rolesID)):
            rolesFile.write("INSERT INTO Roles (rolesID, sentenceID) VALUES (" +
str(rolesID[r]) + "," + str(sentenceID[s]) + ");\n")
            #PrimaryRole
            prolesID = re.findall(" primaryRoleID=\"(.*?)\">", role[r])
            pRole = re.findall("<PrimaryNCIRoleCode.*>(.*?)</PrimaryNCIRoleCode>",
role[r])
            for id in range(0, len(prolesID)):
                proleFile.write("INSERT INTO PrimaryNCIRoleCode (primaryRoleID, prima-
ryNCIRoleCode, rolesID) VALUES (" + str(prolesID[id]) + ",\'" +str(pRole[id])+"\',"+
str(rolesID[r]) + ");\n")
            #OtherRole
            orolesID = re.findall(" otherRoleID=\"(.*?)\">", sentence[s])
            oRole = re.findall("<OtherRole.*>(.*?)</OtherRole>", sentence[s])
            for id in range(0, len(orolesID)):
                oroleFile.write("INSERT INTO OtherRole (otherRoleID, otherRole, rolesID)
VALUES (" + str(orolesID[id]) + ",\'" + str(oRole[id]) + "\'," + str(rolesID[r]) +
");\n")
        #EvidenceCode
        evidenceID = re.findall(" evidenceID=\"(.*?)\">", sentence[s])
        evi = re.findall("<EvidenceCode.*>(.*?)</EvidenceCode>", sentence[s])
        for id in range(0, len(evidenceID)):
            evidenceFile.write("INSERT INTO EvidenceCode (evidenceID, evidenceCode, sen-
tenceID) VALUES (" + str(evidenceID[id]) + ",\'" + str(evi[id]) + "\'," +
str(sentenceID[s]) + ");\n")
         #Comments
        commentsID = re.findall(" commentsID=\"(.*?)\">", sentence[s])
        Comm = re.findall("<Comments.*>(.*?)</Comments>", sentence[s])
        for id in range(0, len(commentsID)):
            commentFile.write("INSERT INTO Comments (commentsID, comments, sentenceID)
VALUES (" + str(commentsID[id]) + ",\'" + str(Comm[id]) + "\'," + str(sentenceID[s]) +
");\n")
```

b) UniProt schema registration

```
CREATE OR REPLACE DIRECTORY XML_FILES AS '/home/teaching/ex4_data_cgd';
BEGIN
  DBMS_XMLSCHEMA.registerSchema(
  SCHEMAURL => 'www.uniprot.org/support/docs/uniprot.xsd',
  SCHEMADOC => bfilename('XML_FILES','uniprot.xsd'),
  CSID => nls_charset_id('AL32UTF8'));
END;
/
```

c) UniProt as columns and insertion of UniProt data

```
CREATE TABLE uniprot_as_column(Version VARCHAR2(10), xml_doc XMLType)
XMLTYPE COLUMN xml_doc
STORE AS CLOB
XMLSCHEMA "www.uniprot.org/support/docs/uniprot.xsd"
ELEMENT "uniprot";

CREATE OR REPLACE DIRECTORY XML_FILES AS '/home/teaching/ex4_data_cgd';
INSERT INTO uniprot_as_column (Version, xml_doc) VALUES(
  '17-04-2018',
  XMLType(BFILENAME('XML_FILES', 'uniprot_sprot_human5.xml'),
  NLS_CHARSET_ID('AL32UTF8')));
INSERT INTO uniprot_as_column (Version, xml_doc) VALUES(
  '18-04-2018',
  XMLType(BFILENAME('XML_FILES', 'uniprot_sprot_human6.xml'),
  NLS_CHARSET_ID('AL32UTF8')));
INSERT INTO uniprot_as_column (Version, xml_doc) VALUES(
  '19-04-2018',
  XMLType(BFILENAME('XML_FILES', 'uniprot_sprot_mouse5.xml'),
  NLS_CHARSET_ID('AL32UTF8')));
INSERT INTO uniprot_as_column (Version, xml_doc) VALUES(
  '20-04-2018',
  XMLType(BFILENAME('XML_FILES', 'uniprot_sprot_rat5.xml'),
  NLS_CHARSET_ID('AL32UTF8')));
```

d) Creation of SwissProt views

```
create or replace VIEW SP_PROTEIN AS
select uac.VERSION as SP_Version, x_ml.*
from uniprot_as_column uac, XMLTable(
xmlnamespaces
(DEFAULT 'http://uniprot.org/uniprot'),
'for $i in /uniprot/entry return $i'
passing uac.XML_DOC
columns
Id varchar2(12) path 'name',
CreateDate date path '@created',
ModifiedDate date path '@modified',
Version int path '@version',
OrganismName varchar2(30) path 'organism/name[@type="common"]',
SequenceLength int path 'sequence/@length',
Mass int path 'sequence/@mass',
Sequence varchar2(2470) path 'sequence'
) x_ml;

create or replace VIEW SP_DBREFERENCE AS
select uac.VERSION as SP_Version, x_ml.*
from UNIPROT_AS_COLUMN uac, XMLTable(
xmlnamespaces
(DEFAULT 'http://uniprot.org/uniprot'),
'for $i in /uniprot/entry,
```

```
  $j in $i/dbReference
  return element a{
    $i/name,
    element dbaccession {$j/@id/string()},
    element db {$j/@type/string()}
  }'
passing uac.XML_DOC
columns
Id varchar2(20) path 'name',
DBAccession varchar2(30) path 'dbaccession',
db varchar2(30) path 'db'
) x_ml;

create or replace VIEW SP_KEYWORDS AS
select uac.VERSION as SP_Version, x_ml.*
from UNIPROT_AS_COLUMN uac, XMLTable(
xmlnamespaces
(DEFAULT 'http://uniprot.org/uniprot'),
'for $i in /uniprot/entry,
$j in $i/keyword
return element a{
  $i/name,
  element keyword {
    $j/text()
  }
}'
passing uac.XML_DOC
columns
Id varchar2(20) path 'name',
Keyword varchar2(30) path 'keyword'
) x_ml;

create or replace VIEW SP_ACCESSION AS
select uac.VERSION as SP_Version, x_ml.*
from UNIPROT_AS_COLUMN uac, XMLTable(
xmlnamespaces
(DEFAULT 'http://uniprot.org/uniprot'),
'for $i in /uniprot/entry,
$j in $i/accession
return element a{
  $i/name,
  element accession {
    $j/text()
  }
}'
passing uac.XML_DOC
columns
Id varchar2(20) path 'name',
Accession varchar2(10) path 'accession'
) x_ml;

create or replace VIEW SP_GENENAMES AS
select uac.VERSION as SP_Version, x_ml.*
from UNIPROT_AS_COLUMN uac, XMLTable(
xmlnamespaces
(DEFAULT 'http://uniprot.org/uniprot'),
'for $i in /uniprot/entry,
$j in $i/gene/name
return element a{
  $i/name,
  element nametype {
    $j/@type/string()
  },
  element genename{
    $j/text()
  }
```

JOHANNES KEPLER
UNIVERSITY LINZ

```
}'
passing uac.XML_DOC
columns
Id varchar2(20) path 'name',
NameType varchar2(20) path 'nametype',
Name varchar2(10) path 'genename'
) x_ml;
```

e) Creation of the external Prosite table

```
CREATE OR REPLACE DIRECTORY ex5_data_cc AS '/home/teaching/ex5_data_cc';
drop table PROSITE;
create table PROSITE (
  AID   char(5),
  LineCode   char(2),
  Value  char(153)
)
organization external (
   type       oracle_loader
   default directory ex5_data_cc
   access parameters (
     records delimited by "\n"
     fields (
         AID   position(1:5) char(5),
         LineCode  position(7:8) char(2),
         Value position(12:164) char(153)
    )
  )
  location ('prosite.dat')
)
reject limit unlimited;
```

f) Creation of the Prosite views

```
create or replace view Prosite_Pattern (AID, Pattern) as
select AID, value from prosite where
linecode = 'PA';

create or replace view Prosite_Entry_ID as
select AID, value from PROSITE where
LineCode = 'ID';

create or replace view Prosite_Entry_AC as
select AID, value from PROSITE where
LineCode = 'AC';

create or replace view Prosite_Entry_DT as
select AID, value from PROSITE where
LineCode = 'DT';

create or replace view Prosite_Entry_DE as
select AID, value from PROSITE where
LineCode = 'DE';

create or replace view Prosite_Entry_DO as
select AID, value from PROSITE where
LineCode = 'DO';

create or replace view PROSITE_ENTRY as
select Prosite_Entry_ID.AID as AID, Prosite_Entry_ID.value as Identification,
Prosite_Entry_AC.value as Accession, Prosite_Entry_DT.value as Dates, Pro-
site_Entry_DE.value as Description,
Prosite_Entry_DO.value as Documentation from
Prosite_Entry_ID, Prosite_Entry_AC, Prosite_Entry_DT,
```

```
Prosite_Entry_DE, Prosite_Entry_DO where
Prosite_Entry_ID.AID = Prosite_Entry_AC.AID and
Prosite_Entry_ID.AID = Prosite_Entry_DT.AID and
Prosite_Entry_ID.AID = Prosite_Entry_DE.AID and
Prosite_Entry_ID.AID = Prosite_Entry_DO.AID;

create or replace view Prosite_DBRef_SUB1 as
select AID, REGEXP_SUBSTR(value, '[[:alnum:]]{6}') as SwissProt_AC,
REGEXP_SUBSTR(value, '[[:alnum:]]{0,5}_[[:alnum:]]{0,5}') as SwissProt_ID,
substr(REGEXP_SUBSTR(value, '[[:alnum:]];'),1,1) as Flag
from PROSITE where
regexp_instr(value,'[[:alnum:]]{6}') != 0 and
regexp_instr(value,'[[:alnum:]]{0,5}_[[:alnum:]]{0,5}')!=0 and
regexp_instr(value,'[[:alnum:]];') != 0 and
LineCode = 'DR';

create or replace view Prosite_DBRef_SUB2 as
select AID, REGEXP_SUBSTR(value, '[[:alnum:]]{6}',1,2) as SwissProt_AC,
REGEXP_SUBSTR(value, '[[:alnum:]]{0,5}_[[:alnum:]]{0,5}',1,2) as SwissProt_ID,
substr(REGEXP_SUBSTR(value, '[[:alnum:]];',1,2),1,1) as Flag
from PROSITE where
regexp_instr(value,'[[:alnum:]]{6}',1,2) != 0 and
regexp_instr(value,'[[:alnum:]]{0,5}_[[:alnum:]]{0,5}' ,1,2)!=0 and
regexp_instr(value,'[[:alnum:]];',1,2) != 0 and
LineCode = 'DR';

create or replace view Prosite_DBRef_SUB3 as
select AID, REGEXP_SUBSTR(value, '[[:alnum:]]{6}',1,3) as SwissProt_AC,
REGEXP_SUBSTR(value, '[[:alnum:]]{0,5}_[[:alnum:]]{0,5}',1,3) as SwissProt_ID,
substr(REGEXP_SUBSTR(value, '[[:alnum:]];',1,3),1,1) as Flag
from PROSITE where
regexp_instr(value,'[[:alnum:]]{6}',1,3) != 0 and
regexp_instr(value,'[[:alnum:]]{0,5}_[[:alnum:]]{0,5}',1,3)!=0 and
regexp_instr(value,'[[:alnum:]];',1,3) != 0 and
LineCode = 'DR';

create or replace view Prosite_DatabaseReference as
select * from Prosite_DBRef_SUB1
union
select * from Prosite_DBRef_SUB2
union
select * from Prosite_DBRef_SUB3;
```

g) Creation of table terms

```
CREATE TABLE term (
id NUMBER(11) NOT NULL UNIQUE,
name VARCHAR2(255) NOT NULL,
term_type VARCHAR2(55) NOT NULL,
acc VARCHAR2(255) NOT NULL,
is_obsolete NUMBER(11) NOT NULL,
is_root NUMBER(11) NOT NULL,
is_relation NUMBER(11) NOT NULL
);
```

h) Creation of table term2term

```
CREATE TABLE term2term (
id NUMBER(11),
relationship_type_id NUMBER(11) NOT NULL,
term1_id NUMBER(11) NOT NULL,
term2_id NUMBER(11) NOT NULL,
is_complete NUMBER(11) NOT NULL,
CONSTRAINT PK_term2term PRIMARY KEY (id),
```

```
CONSTRAINT FK_type FOREIGN KEY (relationship_type_id) REFERENCES term(id),
CONSTRAINT FK_term1 FOREIGN KEY (term1_id) REFERENCES term(id),
CONSTRAINT FK_term2 FOREIGN KEY (term2_id) REFERENCES term(id)
);
```