TÉCNICO LISBOA

# Reusable Framework for Digital Humanities

A Case Study with the *LdoD Archive*

**Miguel Ângelo dos Santos Cruz**

Thesis to obtain the Master of Science Degree in

## Information Systems and Computer Engineering

Supervisors: Prof. António Manuel Ferreira Rito da Silva
Prof. Manuel José de Freitas Portela

### Examination Committee

Chairperson: Prof. Francisco João Duarte Cordeiro Correia dos Santos
Supervisor: Prof. António Manuel Ferreira Rito da Silva
Member of the Committee: Prof. Rui Filipe Lima Maranhão de Abreu

**October 2018**

# Acknowledgments

I would like to thank my whole family for their friendship, encouragement and caring over all these years, for always being there for me, in special to my parents that have made so much efforts for this to be possible.

I express all my gratitude to my girlfriend that always supported me through this last year and never let me give up, that had always helped me keep focused and always listened to my worries.

I would also like to acknowledge my dissertation supervisors Prof. António Rito Silva and Prof. Manuel Portela for their insight, support and sharing of knowledge that has made this Thesis possible.

Last but not least, to all my friends and colleagues that helped me grow as a person and were always there for me during the good and bad times through this last year, Thank you.

To each and every one of you – Thank you.

# Resumo

Esta tese estuda vários repositórios digitais a fim de definir um conjunto de módulos que podem ser compostos para facilitar a implementação de arquivos digitais. O objetivo é atingido analisando o Arquivo LdoD, uma ferramenta existente para o Livro do Desassossego de Fernando Pessoa, em função de um modelo de funcionalidades e mostrar que pode ser usado para reproduzir outras duas ferramenas existentes, EVT e LombardPress. O domínio do Arquivo LdoD foi refatorizado tendo por base o modelo de funcionalidades, o que resultou na necessidade de resolver dependencias entre classes de módulos diferentes. As técnicas usadas para dividir o domínio são explicadas ao pormenor. Por outro lado, foi desenvolvida uma interface de utilizador dinâmica para permitir o desenvolvimento incremental, tornando a transição entre interfaces de utilizador mais fácil. A aplicação final é a composição de módulos que implementam diferentes funcionalidades.

# Palavras Chave

Humanidades Digitais, TEI, Arquivo Digital, O Livro do Desassossego, Arquivo LdoD, Engenharia de Software

# Abstract

This thesis examines several digital repositories in order to define a set of modules which can be composed to easily implement digital repositories. The goal is achieved by analysing the *LdoD Archive*, an existing archive for Fernando Pessoa's *Book of Disquiet*, in terms of a feature model and show how it can be used to simulate two other well known digital humanities systems, EVT and LombardPress. The domain of the *LdoD Archive* was refactored in terms of the identified feature model, which resulted in the need to solve dependencies between classes that belong to different modules. The techniques used to divide the domain are explained in depth. On the other hand, a dynamic user interface was developed to allow incremental development, making the transition between user interfaces easier. The final core application is a composition of modules that implement different features.

# Keywords

Digital Humanities, TEI, Digital Archive, The Book of the Disquiet, Fernando Pessoa, LdoD Archive, Software Architecture

# Contents


**1 Introduction** **1**

**2 Related Work** **5**

2.1 Digital Humanities . . . . . 5

2.2 Digital Humanities (DH) Tool Support . . . . . 8

2.3 *LdoD Archive* . . . . . 9

2.4 Fénix Framework . . . . . 13

2.5 Existing Tools . . . . . 14

2.6 Analysis . . . . . 15

2.7 Edition Visualization Technology (EVT) . . . . . 17

2.8 Juxta Commons . . . . . 19

2.9 Lombard Press . . . . . 21

2.10 Comparison with *LdoD Archive* . . . . . 22

**3 Initial *LdoD Archive* Architecture** **25**

3.1 Features . . . . . 25

3.2 Packages . . . . . 27

3.3 Domain . . . . . 28

**4 The new architecure** **29**

4.1 Features Decomposition . . . . . 29

4.2 Packages Decomposition . . . . . 30

4.3 Domain Decomposition . . . . . 31

4.3.1 Relations between classes of different modules . . . . . 31

4.3.2 *LdoD Archive* case . . . . . 33

4.3.2.A Relations among classes from different modules . . . . . 33

4.3.2.B Delete Object . . . . . 37

4.3.2.C Further decompositions . . . . . 41

**5 Presentation of *LdoD Archive*** **43**

**6 Validation** **55**
6.1 `edition-user` module . . . 56
6.2 `edition-annotation` module . . . 56
6.3 `edition-aware` module . . . 56
6.4 `edition-search` module . . . 57
6.5 Deleters configuration . . . 57
6.6 Composing the imports and exports . . . 58
**7 Conclusion** **59**
7.1 Conclusions . . . 59
7.2 System Limitations and Future Work . . . 60
**A Diagrams** **68**

# List of Figures


2.1 Model of the genetic and social interpretations of LdoD . . . . . . . . . . . . . . . . . . . . 11
2.2 Model of the virtual interpretations of LdoD . . . . . . . . . . . . . . . . . . . . . . . . . . . 12
2.3 *LdoD Archive* Software Architecture . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 13

3.1 Component and Connector Diagram of *LdoD Archive* . . . . . . . . . . . . . . . . . . . . 26
3.2 Package Diagram of *LdoD Archive* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 27

4.1 Decomposition of the packages into the two modules . . . . . . . . . . . . . . . . . . . . . 30
4.2 Uses relation across modules . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 Uses relation across modules with Fenix-Framework . . . . . . . . . . . . . . . . . . . . . 32
4.4 How classes are recompiled with Fenix-Framework . . . . . . . . . . . . . . . . . . . . . . 32
4.5 Relation between the class `FragInter` and its subclasses . . . . . . . . . . . . . . . . . . 33
4.6 Relations of inheritance and use between classes . . . . . . . . . . . . . . . . . . . . . . 36

5.1 Division between ReactJS elements and JSP content . . . . . . . . . . . . . . . . . . . . 44
5.2 Menu element About . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 46

A.1 Feature Diagram of *LdoD Archive* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 69
A.2 Domain Diagram of *LdoD Archive* . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
A.3 Feature Diagram comparing the three tools . . . . . . . . . . . . . . . . . . . . . . . . . . 71
A.4 Decomposition of the domain into the two modules . . . . . . . . . . . . . . . . . . . . . . 72
A.5 Proposed decomposition of the domain according to the features . . . . . . . . . . . . . . 73
A.6 Decomposition of the packages into the nine modules . . . . . . . . . . . . . . . . . . . . 74

# List of Tables

2.1 Comparison summary of studied tools. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

# Listings

4.1 Methods of ExpertEditionInter used by VirtualEditionInter . . . . . . . . . . . . . . . 34
4.2 Method getLastUsed in the VirtualEditionInter class before the changes . . . . . . . 35
4.3 Method getLastUsed and its auxiliary method in the VirtualEditionInter class after the changes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
4.4 VirtualEditionController . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 36
4.5 Method remove of the class Fragment before refactoring . . . . . . . . . . . . . . . . . . . 37
4.6 Method remove of the class Fragment after refactoring . . . . . . . . . . . . . . . . . . . 38
4.7 FragmentDeleter class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39
4.8 FragmentDeleterVirtual class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.9 DeletersConfig class . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
5.1 Code to generate each menu element . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
5.2 Code to generate the About menu element . . . . . . . . . . . . . . . . . . . . . . . . . 46
5.3 Base code of the UpdatableIntlProvider . . . . . . . . . . . . . . . . . . . . . . . . . . 47
5.4 updateLocale of the UpdatableIntlProvider element . . . . . . . . . . . . . . . . . . . 48
5.5 Passing updateLocale function to the context . . . . . . . . . . . . . . . . . . . . . . . 49
5.6 Base element of the ReactJS application . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.7 Code to handle the href inside the legacy HTML . . . . . . . . . . . . . . . . . . . . . . 50
5.8 render function of LegacyPage . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.9 Fetch of the HTML from the Spring Application . . . . . . . . . . . . . . . . . . . . . . 51

# Acronyms

**DH** Digital Humanities

**EVT** Edition Visualization Technology

**SCTA** Sentence Commentary Text Archive

**QA** Quality Assurance

**DDG** Data Dictionary Generator

**TILE** Text-Image Linking Environment

**DML** Domain Modeling Language

# 1 Introduction

*LdoD Archive* is a Digital Archive focused on ”The Book of Disquiet” by Fernando Pessoa and intends to give a customized view of the book by providing features that deliver a unique experience to the users. With *LdoD Archive* it is possible to read the book as organised by four experts, read the original version directly from the original format, reorder the fragments at will or even read the documents ordered according to multiple criteria.

Digital Humanities (DH) is a field of the Humanities' research that aims to develop tools that support the work done by the various disciplines of Humanities. The definition of DH continues to be debated and discussed as witnessed in several books published in the last decade. There is a cooperation between different fields to optimise the work done in DH and the final work is the result of bringing together the traditional humanities' disciplines and redefining them in the context of developing computational tools and resources.

One field of work within the DH context are the Digital Archives, in specific, the literary ones, that provide literary works to their users. Being generally accessed through a website, it gives a much larger number of potential users than a physical archive, by being accessible using a common web browser. This allows users to access the literary works from anywhere in the world, breaking the geographical

restrictions.

Among the available Digital Archives found online there were two that were studied more in depth because of their similarity with the *LdoD Archive*, Edition Visualization Technology (EVT) and Lombard-Press. They are Digital Archives that use literary works encoded using the TEI specification and present them to the user along with other features, such as information about the texts themselves, a digital copy of the original material document or comparison between multiple versions of the same text. These are features that are also found in the *LdoD Archive* among many others. These tools helped to understand what features must be available in a Digital Archive and how can these features be implemented.

Using the LdoD Archive as the base of the work, the ultimate goal of this dissertation is to build a software product line for the DH that produces a Digital Archive that meets the requirements of as many Digital Archives as possible. It requires that an extensive set of features is available to compose the different projects, but also, and not less important, not to provide features that are unnecessary or irrelevant for the context of the project.

Currently, *LdoD Archive* is a Spring Application implemented using only one module separated into multiple packages. There is a unique domain that is shared by the entire application. The goal is to divide the current single module into multiple modules. Starting from a base module that contains the base feature of every Digital Archive, a set of modules should be created, such that each module is responsible for the implementation of a single feature with as few dependencies between modules as possible. This way, it is possible to create a new application that works as a composition of the multiple modules, according to the needed features for the project that is being implemented.

To achieve this goal it is necessary to divide the currently implemented domain into smaller, independent domains that only have the necessary classes to implement the feature that such module implements. There are cases where classes that need to be placed into different modules have crossed dependencies. In these cases it is necessary to understand if those dependencies are really necessary and how can they be solved. To solve those dependencies a set of strategies were developed in order to create the different modules. There were cases where a method from a feature received objects that it would then use to retrieve the intended type of object. In this case, instead of calling the method with the wrong type of object, the right one was obtained before calling the method. There are also cases where the objects had relations that were established outside its module and it is not possible to use that relation within the context of the module that is used. A more particular case of this problem happens when an object needed to be deleted, where Fenix-Framework demands that before deleting an object it is necessary to remove all the relations in which the object exists. In these cases it was necessary to create a set of classes that were responsible for deleting the objects and all their relations without introducing dependencies between modules.

It was also created a new user interface, based in ReactJS, a JavaScript library, that allows an in-

cremental development, where the previous user interface can be used in combination with the new interface while the implementation of the new interface is in development. The new user interface keeps the same look and feel from the previous interface, with the difference that it should rely on REST endpoints instead of JSPs, which allows projects to reuse the default interface, implement a new interface that suits the context better or even have multiple interfaces for the same project.

The final result is a set of modules that are responsible for different features that can be combined to obtain a tool that meets the requirements for a given project. The obtained tool is a composition of these modules that is the configured to the needs of the project. Although the work was not completed to what was considered the ideal state, the methodologies used to begin the work were well documented and can be applied to the remaining divisions that were not done.

In this work, the modules that were created allowed to compose the tools that were studied, EVT and LombardPress. Although the tools can be reproduced, the ideal would be to create smaller modules, once there were still features that were inside the same module that should be optional. For every feature that was identified as optional it should be possible to remove it without removing any other, unless there is a feature that depends on the first one. In that case the one that depends on the other should also be removed.

# 2

# Related Work

## 2.1 Digital Humanities

Humanities' research is based on reasoning over sources, which may be textual, material or intangible. This investigation can be improved by using tools that facilitate part of the work. The creation of these tools is an important part of research in DH, whose definition is still being debated since it is a field which is currently under an intense process of growth and transformation. The emergent, heterogeneous and debatable nature of DH is reflected in several major books published over the last five years. [1–5]

The current DH was developed out of humanities computing and started working together with other fields, such as social computing and media studies. The working materials for DH are both digitized, i.e. former physical material that was brought to digital format, and created directly in digital format. DH brings together methodologies from traditional humanities disciplines, e.g. history, philosophy, linguistics, literature, art, archaeology, music, cultural studies, and social sciences and redefines them in the context of developing computational tools and resources, e.g. Hypertext, data visualization, information retrieval and data mining. [6]

Although it is hard to classify this field there are five common values across many of the existing

projects of DH. They are an interesting way of getting to understand the field of DH itself, in both form and content, theory and practice. [7]

- Critical and Theoretical - DH scholarship is based on the humanistic theory and critical tradition, like many scholarly practices but it is often also based on a humanistic self-criticism, including the criticism of the very tools, technologies, and platforms that enable its own practices and publications.

- Iterative and Experimental - DH allows the iterative versioning of the digital projects, which promotes experimentation, risk-taking, redefinition, and sometimes failure. It is important that the normalization of practices, the standardization of methodologies and the definition of evaluation metrics are not rushed.

- Collaborative and Distributed - Since DH texts often have multiple authors with multiple versions, subtle and robust collaborations are the foundation of many DH projects which involves distributed networks of expertise including scholars, students, programmers, technologists, librarians, designers, and more.

- Multimodal and Performative - The DH scholarship are not confined by the strictures and structures of print, embracing many modes, such as text, audio or video. With the presence of several formats it becomes easier to express the author's point to the readers, facilitating the interpretation of the text.

- Open and Accessible - Although not all are open-source, most of the DH projects are open and publicly accessible, with pre- and post-publication peer analysis which allows for the production of work with multiple feedback leading to an improved final result.

Alongside the central role of theory there are methodologies that guide DH work. These methodologies should not be mistaken with DH technologies. The most effective way to understand DH is to explore through its many methodologies. Although it is hard to enumerate all the methodologies the book "Digital_Humanities" [8] contains a whole chapter dedicated to specify as far as possible the DH methodologies. The following list is based on the methods described on pages 32-60.

- Enhanced Critical Curation (page 32-34) - The process by which media (traditionally texts, but now expanded to all digital forms) is organized and displayed to best convey the information to its readers supported by digitized or digitally produced source materials.

- Augmented Editions and Fluid Textuality (page 35-36) - Digital critical editions that are marked-up and encoded texts, generally developed through crowd-sourced contributions and always open to revision or annotation.

- Scale: The Law of Large Numbers (page 37-39) - With the continuous growing of the quantity of information to be stored and accessed there is an urge to find digitized and digitally produced culture materials through computational means.
- Distant/Close, Macro/Micro, Surface/Depth (page 39-40) - In opposition to close reading, distant reading allows to go through digital data bases to find ”trends, patterns and relationships” instead of focusing in a work or few works at a time.
- Cultural Analytics, Aggregation, and Data-Mining (page 40-42) - With the evolution of the technology it became possible ”to dissect large-scale cultural data sets” and extract information from them through data-mining. With such information it is possible to display materials in different forms, including interactive and narrativized visualizations.
- Visualization and Data Design (page 42-45) - Information constructed from the visualization of data, virtual/spatial representations, geo-referencing and mapping, simulated environments, and other designs.
- Locative Investigation and Thick Mapping (page 45-47) - Association between data and cultures or geographic locations with the connection of real, virtual and interpretative sites.
- The Animated Archive (page 47-49) - Allows the static archives from the past to be represented digitally, organizing and accessing physical spaces using virtual means.
- Distributed Knowledge Production and Performative Access (page 49-51) - Alternatively to the traditional idea of the ”author”, nowadays it is possible to have collaborative teams from the different disciplines and countries contributing to the same project, without ever meeting personally.
- Humanities Gaming (page 51-52) - Brings the humanist themes to games using virtual learning environments with interactive narratives.
- Code, Software, and Platform Studies (page 53-54) - The study of code as text, what it does and does not allow to do, the cultural context where the code is developed, the structure of the code and the relation between the software and the hardware.
- Database Documentaries (page 54-55) - Modular and combinatoric sets of information gathered that can be built of out of a wide range of media like, video, sound, static image, text and even World Wide Web content. This differs from the typical documentary by allowing to generate different narratives over the same content, guiding the viewer through the most relevant information given the case.

- Repurposable Content and Remix Culture (page 55-57) - All data available digitally can be read, written, and rewritten, so all data is subject of sample, migration, translation, remix and other ways of reuse.

- Pervasive Infrastructure (page 57-58) - With the actual diversity of devices to access digital information and the spread of cloud computing and storage use data is reaching more people and is less susceptible to loss.

- Ubiquitous Scholarship (page 58-60) - The distance from the print publication only era is growing and nowadays a whole new set of publishing forms is expanding, becoming increasingly ubiquitous and open.

## 2.2 DH Tool Support

Research in the Humanities is supported by creating tools and tailoring computer technologies to the specific needs of humanities investigators, who have particular methodologies of work, different from the common computer applications that already exist.

Nowadays there is a large set of tools which make the most of recent technologies, which means that it is possible for these tools to be accessible from almost any device, depending on the tool, from small devices such as a mobile phone to large virtual reality labs. These tools also vary in complexity, from tools that use complex programming languages and databases to more simple tools for simpler tasks.

Typically, Humanities researchers, the main stakeholders of these tools, are not advanced computer users, and this must be taken into account when developing a tool that is going to be used by them, from the documentation to the interface. This is the only way to ensure the tools are going to be used and will, effectively, improve the work experience of their users.

There are many projects of Digital Humanities that aim to deliver the best literature works of history, both for scholars and general readers. Some of these projects intend to provide a platform where everybody can access works that would not be available for the general public from texts of ancient literature to contemporary writers. These projects have generally been called Digital Archives.

All Digital Archives have different features depending on the purpose of the project. There are projects whose aim is to help the work of scholars, providing tools to analyse the texts or transcribing the originals into digital format. On the other hand, there are projects that aim to deliver the literary works to the general public, allowing everybody to access a large number of works that would not be possible to access by other means.

For this to be possible it is necessary to do a tremendous work and there are a huge set of decisions that need to be made. To begin with, if the original source is not digital, it is necessary to transform

it to digital format. This conversion may include or not the transcription of the texts to digital format, i.e. the texts can be simply scanned or completely transcribed. When it comes to transcribing texts it is necessary to establish what is the encoding that is going to be used, TEI being one of the standard options.

TEI [9] is a consortium which collectively develops and maintains a standard for the representation of texts in digital format. The consortium is composed of academic institutions, research projects, and individual scholars from around the world that intended to specify the encoding methods for machine-readable texts.

After defining what is the encoding used in the archive it is necessary to choose all the architecture that will keep the system running. At this point it is necessary to define how the information is going to be loaded, stored, accessed and presented.

The loading of the information may vary depending on the encoding of the texts. There are texts encoded specifically for a project facilitating the loading by having a known and regular specification. In cases where the objective is to reuse previously encoded texts that do not have any special treatment besides the regular encoding the loading of the information may be more complicated.

After defining how the information is loaded it is necessary how it is going to be stored. The information can be stored as it is, i.e. storing the files as they are, without any treatment, or the information can be processed, e.g. storing the information as a object-oriented database, the method implemented in the *LdoD Archive*.

With the information stored it remains to decide how the information is going to be accessed and presented. When the source files use TEI encoding there are two different ways of generating the visualization of the texts. If the files are stored as they are in the database then the presentation is defined by applying XSLT transformations to the file, if it is stored in a object-oriented database then the presentation is generated like any information system data is displayed, through the specification of the behaviour of each of the objects in a traditional programming language, this being the case of *LdoD Archive*.

When displaying the information it is necessary to establish what are the features that the archive is going to provide. The most commonly and easily provided are the simple presentation of the texts without any interaction, i.e. a static interface.

## 2.3 *LdoD Archive*

Fernando Pessoa's *The Book of Disquiet* is an unfinished book written between 1913 and 1935 that was never published, so the texts are simply typed or hand-written on loose sheets and scattered across various notebooks. Although the author left some information about how the book should be organized

there is no final authorial version of the book. This situation led to several interpretations of what should be the organization of the book and different editions started to be published, the first, by Jacinto do Prado Coelho [10], appeared only in 1982 and three more editions were published until now.

The particular situation of the book was the reason for the creation of a digital repository that would allow to read and analyse the book as it was put together by the different editions but also the original texts. The objective was to bring together three dimensions of the book, genetic, i.e. the composition of the book by the author, social, i.e. the construction of the book by its editors, and virtual, i.e. the reconstruction of the book by its readers, using the tools provided by information systems.

The *LdoD Archive* was created, according to a different model from other existing digital archives. It allows not only to read and analyse the multiple editions of the book, but also enabled the collaboration and social interaction among users and marking material changes at code level. These features of networked computational media intended to redesign the digital archive beyond the usual concepts and forms of implementation, in ways that enabled it to be used by researchers, teachers, and students, but also by general readers.

The *LdoD Archive* allow users to virtualize the book according to four functions: reader-function, editor-function, book-function and author-function.

The reader-function's main objective is to support a contextualized reading of the book by allowing to visualize the variations of the texts according to different witnesses. Each of these witnesses can be authorial revisions, i.e. different versions of the text done by the author during his life, and editorial variations, i.e. work done by interpretation after the author's death. The features associated with reader-function enable the simultaneous visualization of facsimiles, and their textual transcriptions.

The editor-function's main objective is to support the interpretation of the book due to its unfinished state, done by the manipulation of meta-textual information in the fragments of the book as a whole. Its main features are static encoding for the genetic and social editions and dynamic encoding of the virtual editions.

The book-function's main purpose is to construct the book based on pre-defined information, allowing users to generate a new edition of the book based on the textual and meta-textual information of the archive. The major feature of this function is instantiated through the search considering the previously mentioned textual and meta-textual information, but also navigating through fragments, beginning at the named *context-fragment* and navigating to a variant of the same fragment or another fragment chosen by its distance to the current fragment calculated using various types of metadata. The generation of a virtual edition produces an hierarchical list of contents of fragments from the book which are then navigable across the various fragments.

The author-function's main objective is to extend the book with new texts based on the original fragments. For a text to be considered an extension from the original fragments, the references must be

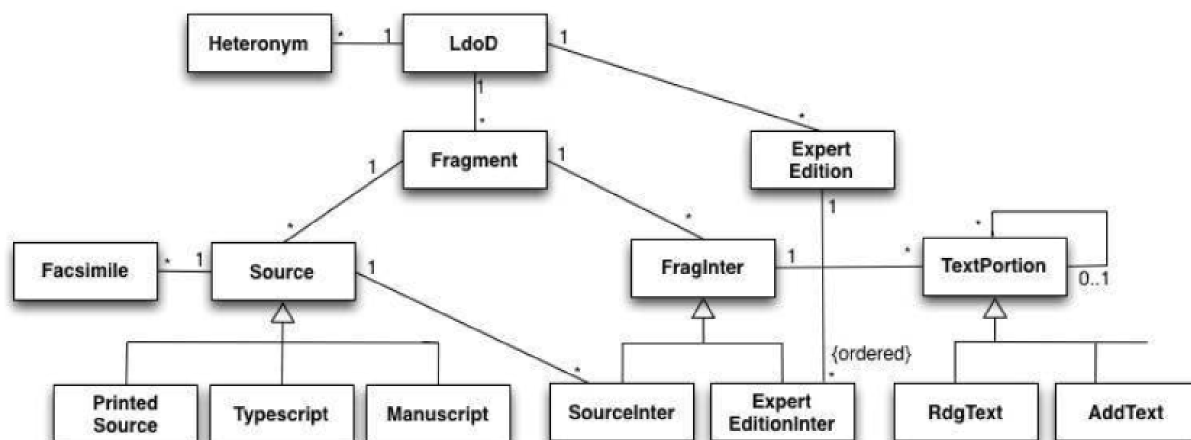


**Figure 2.1:** Model of the genetic and social interpretations of LdoD

kept to the sentences and sequences of words of the original fragment. After doing the extension, the extended fragment becomes part of the virtual edition and can then be included in another virtual editions. This function provides a set of functionalities, copying a fragment for expansion and modification, creating a new fragment based of a set of seed-fragments, and allowing the two first functionalities to work together.

Each of the four functions are performed according to three dimensions, genetic, social and virtual. The genetic dimension is about the creation of the book by Fernando Pessoa with special relevance for the reader-function and book-function. The social dimension is identical to the genetic dimension but instead of the author focused task, it is about the four critical editions. The virtual dimension includes the ability of the users to become editors and authors of the book, with all editorial and authorial actions being made in the context of the virtual edition which is responsible for storing the extended fragments and the new texts. The virtual dimension is also expressed through the creation of shared virtual editions which are the result of the collaboration among multiple users, one of the aspects that differentiate *LdoD Archive* from other digital archives.

While it was relatively easy to address the specification of TEI for the genetic and social dimensions since TEI can express authorial witness and their expert interpretations through the concept of edition, the virtual dimension is more complex since there is the need to create a non-predefined number of virtual editions.

For the genetic and social interpretations, as seen in **Figure 2.1** of the book the root concept is the book itself, which contains a set of Fragments, a set of Heteronyms and Expert Editions. Each of the Fragments have a set of Sources and a set of FragInter, that are Interpretations of the Fragment. These Interpretations can be Authorial or Editorial, SourceInter and ExpertEditionInter respectively. The Authorial Interpretations can be a PrintedSource, Typescript or Manuscript. Editorial Interpretations are contained and ordered in the context of Expert Editions. Each of the Sources has a set of Facsimiles.

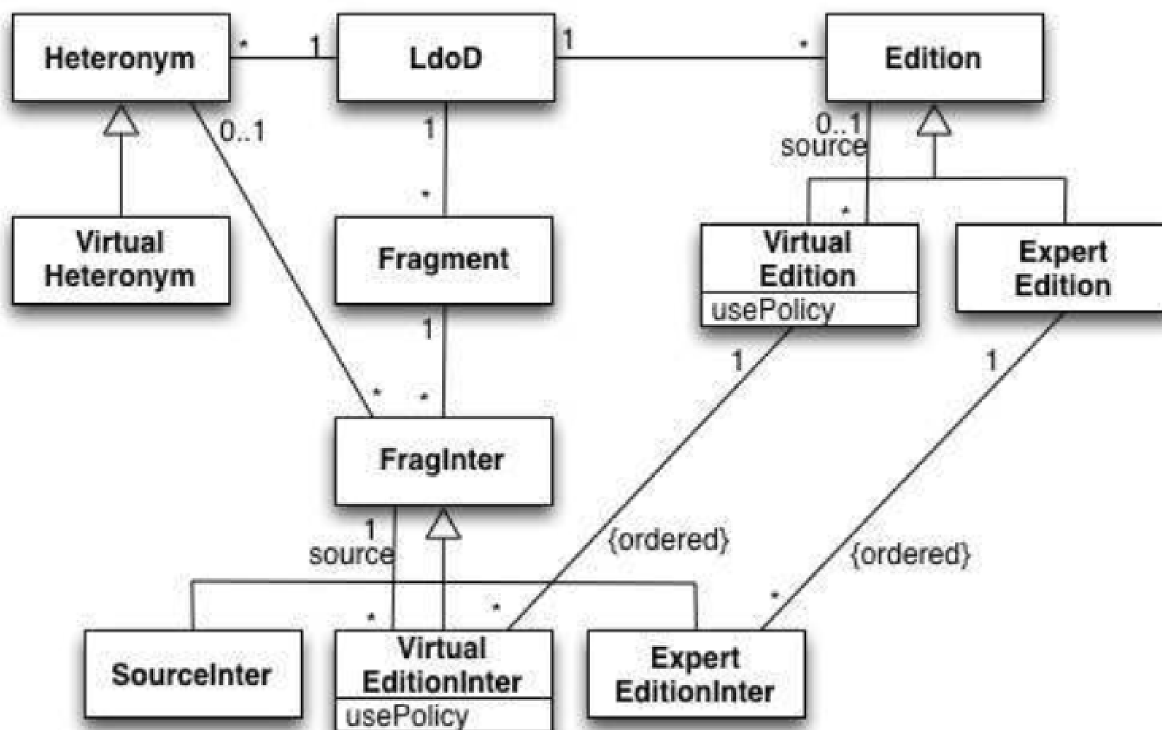


**Figure 2.2:** Model of the virtual interpretations of LdoD

The model to support virtual interpretations was an enrichment of the model for the genetic and social interpretations. As seen in **Figure 2.2**, three new concepts were added to the previous model, VirtualEdition, representing the virtual editions, VirtualEditionInter, representing the interpretations of the fragments in the context of the virtual editions and finally VirtualHeteronym, representing the new heteronyms that may be created by the end users. The principle of the separation of expert and virtual editions is implemented by these new concepts. To support the coexistence of expert and virtual editions an optional association was defined between a VirtualEdition and an Edition and a mandatory association between a VirtualEditionInter and a FragInter.

Regarding the project's architecture, *LdoD Archive* is distinct from most of the other digital scholarly archives by not being static, i.e. the construction is separated from its use. Most of those archives are supported by XLST transformations which do not allow for the provision of Web2.0 functionality to the archive. In the LdoD Archive, the encoding in TEI by experts must be assured while allowing dynamic user interactions with the platform

Instead of supporting the presentation of the texts through XLST transformations of the files from the repository, *LdoD Archive* provides an importer component (:LdoD TEI File Importer) that loads the files into an object-oriented database (:LdoD Object-Oriented Repository). The interaction between the users and the object-oriented database is provided by a web-client server interface (:LdoD Application Server). All the changes performed by the users are stored in the same database. In order to preserve this

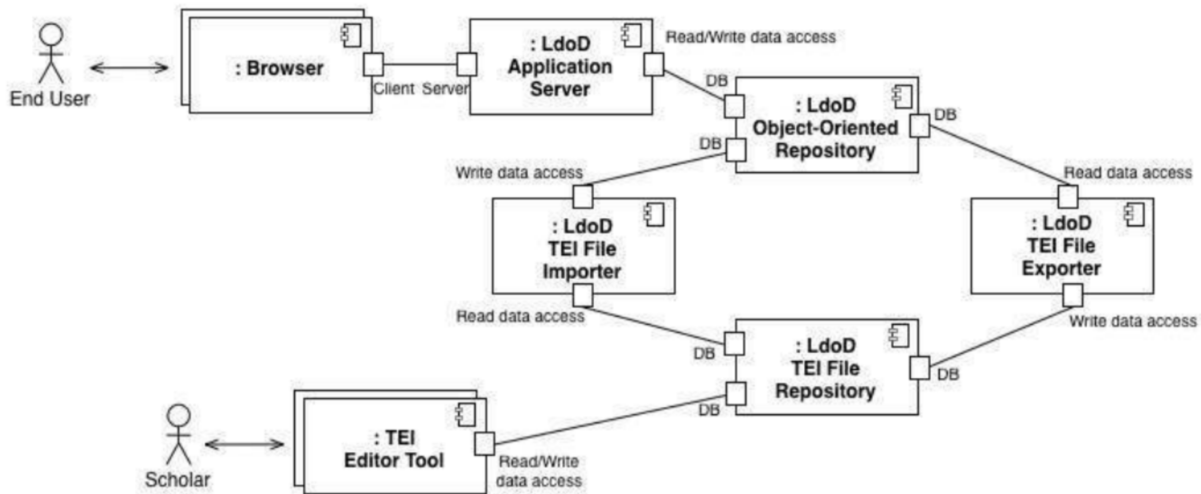


**Figure 2.3:** *LdoD Archive* Software Architecture

changes in TEI-encoded files there is an export component (:LdoD TEI File Exporter) that regenerates the TEI-encoded files from the object-oriented repository, . This component allows selective generation, i.e. it is possible to select which parts of the repository to generate. See **Figure** 2.3.

The key point of the *LdoD Archive* architecture is the representation of the *LdoD Archive* through an object domain model. LdoD encoded in TEI is transformed to the object model allowing the visualization and edition of this object model through a web user interface, and subsequently it is possible to generate TEI files from the object model. With this approach (1) the archive's experts can continue producing TEI encoded content, (2) users can create virtual editions and fragments' extensions through the web user interface, (3) the semantic consistency is assured by the constant checking of the consistency of user's operations, (4) interoperability can be supported by exporting the regenerated TEI files, (5) TEI files can be regenerated on demand.

The Archive is implemented in JAVA, using the Spring Boot framework for server-side behaviour, Bootstrap and JQuery for client-side implementation and all the transactional and persistent domain model is supported by Fénix Framework.

## 2.4 Fénix Framework

Fénix Framework [11] is a project by FenixEDU [12] whose a primary goal is to support the development of JAVA-based applications that require transactional and persistent domain model by providing a programming model that manages the database. This programming model totally hides its existence from the programmer instead of being required to create and keep a relational database apart from the code itself with intuitive methods of accessing the database from inside the code.

When using Fénix Framework programmers must specify the application's domain model using a language created exclusively for this purpose, Domain Modeling Language (DML), while all the development made in plain JAVA. Currently, all the necessary DML for *LdoD Archive* to work was written manually, specifically for the data that is being handled, which requires human intervention every time a change occurs on the requirements of the persistent data model. This makes the process of adapting the current project to other contexts too slow.

## 2.5 Existing Tools

From the universe of all DH tools, our investigation was done only within the ones directly related with the TEI specification. Some of the tools do not manipulate TEI directly but must have some kind of support for TEI files, like using TEI as input for information, or supporting TEI as one of the formats for saving your work's progress. It is possible to separate all the tools related with TEI into twelve, nonexclusive, i.e. each tool can be from more than one category simultaneously.

- Administrative - tools that are used to manage documents, store, catalogue and search, within an organization.
- Analysis - tools focused on the analysis of texts which help users with their manual tasks such as annotating, text revision or comparing, also possibly done automatically, or tools for automatic analysis, like text statistics.
- Conversion and pre-processing - tools that convert to and from TEI using XSLT stylesheets, add or edit tags within the TEI file.
- Development environments - Tools for writing scripts and stylesheets which are used for the manipulation of TEI resources.
- Documentation access - Tools to facilitate the access to TEI documentation given a small set of keywords.
- Header Creation - Tools for creating TEI headers by analysing the content of the file.
- Publishing and Delivery - Tools for publishing the final encoded texts, which may have analysis and image support, providing all the necessary support for having a functional interface.
- Querying - Tools for querying a textual or XML database.
- TEI Encoding - Text editors that have support for TEI editing, which may be explicitly related to TEI or simply general XML support. These tools provide syntax highlight and auto-complete but more

TEI friendly tools also offer the possibility to show document structure, elements' content model, XSLT debugger and tagless editing modes.

- Testing and Quality Assurance (QA) - Tools that check the validation, well-formedness and plausibility of the markup before using the document for other purposes.

At the moment, *LdoD Archive* is included in the analysis category, by providing the possibility to annotate texts, publishing and delivery category, by providing an interface for the users to explore all the book's fragments. There are other categories that were used in the process of bringing the Archive to its current state, like TEI Encoding for preparing the TEI files using the given specification, and Testing and QA tools for checking the correctness of the files written by the specialists.

## 2.6 Analysis

There was an extensive set of tools, related with TEI, studied at the beginning of the work [13, 14]. Most of those tools were discarded mainly because their focus was not within the scope of *LdoD Archive*.

A considerable set of tools were simple XML editors, some of them with more advanced features, but not relevant for our project because all the edition was done previously in external specialized tools and the final specifications are parsed to be included in the archive [15–37].

Another irrelevant set of tools for our objective were the converters, to and from TEI. Most of these tools use XSLT and are simple one-step converters without providing full functionality. Although some of these tools may have interesting features to include in *LdoD Archive*, making them relevant to analyse them architecturally in the context of larger applications. [38–58].

Other tools not studied were XML databases because *LdoD Archive* does not use XML database and is not the objective of this work to change the already in use database. [59–64].

It is not relevant in this context to study TEI customizations generators because the TEI customization for the archive is already established and there is no need to specify new customizations. These customizations are changes done to the TEI specification that allows users to adapt the specification conveniently to the work being developed, i.e. the customization only contains the modules, elements and attributes that the project uses. Although these tools may become useful when testing new functionalities against different TEI customizations to understand the correctness and coverage of different cases [65, 66].

There is another considerable set of tools that were not studied because their function is somewhat different from the purpose of our work. The Classical Text Editor [67] is a tool to prepare texts for physical or digital publishing while *LdoD Archive* is responsible for publishing texts that were already processed and prepared for publication. Collatex [68] is an interesting tool for collating several witnesses of a text

and can output the result in diverse formats, one of which is TEI, but it is not relevant for this work because all the witnesses are already encoded together and ready to be published. There is Context [69], a software for typesetting high-quality documents that separates the content from the appearance, writing plain texts with resource to specified macros, like LaTeX, that accepts TEI as input files. DC-dot [70] is a tool to extract meta-data from web pages arranging it according to a specific schema, and it can be outputted to TEI. DLXS [71] is a tool, no longer maintained, developed by the University of Michigan, that works as a digital library system and accepts TEI as a format for its entries while *LdoD Archive* is not a library collection manager but a tool for visualizing different witnesses of a text. Data Dictionary Generator (DDG) [72] is a tool intended to run inside oXygen [26] to generate profiles of every element and attribute appearing in a TEI file and aims to quickly create stronger encoding guidelines for the collaborators within a project. ETDPub [73] was firstly designed as an Electronic Thesis and Disseration system but evolved into a complete digital library also allowing the publication of TEI content. The EXMARaLDA [74] is a toolset for creating and analysing spoken language corpora, supports the TEI guidelines for Transcriptions of Speech. The Image Markup Tool [75] is a Windows application for annotating images using TEI P5 which is not the objective of this work. Morphadorner [76] acts as a pipeline manager for processes performing morphological adornment of words in a text by attaching pictures and marginal comments. TEI Critical Apparatus Toolbox [77] is a tool for people preparing a natively digital TEI critical edition. This is not useful for this work because the preparation of the witnesses was already done. TEITags [78] is an extension for MediaWiki that allows the display of TEI markup in a wiki. Text-Image Linking Environment (TILE) [79] is a web-based tool for creating and editing image-based electronic editions and digital archives of humanities texts. TILE receives a text encoded in TEI P5 and it allows the linking between the text and the original manuscript outputting a new TEI file. Although it is a good functionality it is not a priority for *LdoD Archive* for now. WebLicht [80] is ”an execution environment for automatic annotation of text corpora. Linguistic tools such as tokenizers, part of speech taggers, and parsers are encapsulated as web services, which can be combined by the user into custom processing chains.” which is not currently relevant for *LdoD Archive* functionalities. CoreBuilder [81] is a tool designed for providing an interface capable of helping during the process of encoding variations between different witnesses into TEI format, which is not relevant at the moment since all this work was previously done using Oxygen.

Moving to the tools that really matter for our work, there were seventeen tools that had significant relevance for the subject under study. These tools are directly related to the visualization and analysis of documents encoded in XML TEI. Studying these tools is a way of understanding what is already done and how it is done.

Not all tools work the same way, neither do they aim to fulfil the same functions. There are tools focused on the visualization of any document/project encoded in TEI that require less configuration from

the user to be able to see the result; tools that are easy to adapt to any project but still require some configuration by the user (e.g. configuration of some XML files) to be able to present the document/project; and finally there are tools that require a complete new implementation for each new document/project they are used in.

The first set of tools deliver a result that falls short of expectations for the work we aim to produce, mainly because the few configuration needed results in simple visualization and analysis options that do not fit the needs of the literature specialists, which does not mean that there is nothing to be learned with the way these tools work. These tools are basically converters, that use XSLT transformations to convert TEI XML into HTML with direct transformations. These conversion methods are interesting for part of the functionalities to be included in *LdoD Archive*.

The second one is able to provide better results with a little more effort. Tools like EVT offer the possibility of analysing and comparing two or more editions of a text simply by adding those editions to the main XSL file. In the particular case of the EVT, it is possible to add personalized style-sheets for the result to be more suited to the project.

Finally there are tools like our tool which were entirely designed for a project only and thus require full configuration to be functional, i.e. require changes from all parts of the project, from the database to the presentation itself.

There are obvious advantages and disadvantages for each of the three types of solutions. For our tool there is not much that we can reuse from the first set of tools, but there is definitely something that we can learn from the other two set of tools. We studied the code and architecture of each one of the tools to find the main similarities and differences between these tools and our own tool.

Inside this bigger sets of tools there are two major types of solutions: the ones based mainly on XSLT, with low to none use of traditional programming languages, and the ones, like our tool, that base their work in a more traditional programming solution. In this last group there are tools that delegate some of the work to XSLT mainly because it is highly efficient when dealing with XML files consumption and processing.

The architecture of this type of tools are mainly a Pipe and Filter that start with a TEI file or a collection of TEI files with the literary work and end up, most of the cases, with an HTML file that contains all the functionalities for analysis and visualization of the work.

## 2.7 EVT

By default, EVT can be used to create two edition levels, diplomatic and diplomatic-interpretative. Every transcription that is encoded using elements of the appropriate TEI module are mostly compatible with EVT. The current main use of EVT, *The Vercelli Book* [82], is based on standard TEI schema, without

any custom elements or attributes added. Currently there is an ongoing research to add the possibility of having critical editions. EVT is currently having a major change in its new version, EVT v. 2.0, described as a reboot of the project. In this version the tool works with the data itself without any need of loading the data. It is also an objective of the last version to take EVT beyond the project-specific tool spectrum. With this new approach it is possible for the editor to create an edition with very little configuration, by simply applying XSLT stylesheet to the TEI encoded texts the result is a web-ready edition, i.e. the web interface based on the TEI file of input. Since web edition is based on a client-only architecture it does not require any additional server software. It can simply be stored in a web server and having clients being served by it.

When the user reaches the generated website of EVT [83], he/she is presented with the manuscript on the left and the transcription on the right, the "Image-Text View", which is the default view. There is also "Text-Text View" used to compare different editions levels, diplomatic and interpretative editions are the ones accepted at the moment, selected by a drop-down menu and "Bookreader" that shows the manuscript with a double-side view, currently it only allows images to be doubled-sided and is not possible to have single folio images. There are several tools available to improve the experience of the analysis of the manuscripts such as the "Magnifier" that helps when exploring the manuscripts with the maximum possible level of precision by showing not only a zoomed image of the selected area but also an image with bigger resolution; the "HotSpot" that highlights areas of the text that have specific notes or details, which will appear on a window by clicking on them; the "TextLink" that links the corresponding text between the manuscript and the edition text and vice versa; and finally the "Thumbnail" that will show a small image of all the available manuscripts' digitized folios. The search functionality is currently under development and is almost complete after the drawbacks of using a complete client-side application, without the possibility of using a XML database with all the related functionalities like indexes. EVT keeps track of people, places and organizations that are referred to in the texts, allowing users to click an element of those three in any text and showing what are the other texts where it is referred to. All the described functionalities require the necessary information to be in the source TEI file.

The EVT is an excellent example of the exclusive use of XSLT for providing a reasonably complete tool for analysing and visualizing a literary work. It is an XLST files chain that begins in a single style sheet, `evt_builder.xsl`, which must indicate the editions that the user wants to generate by adding an <edition> element in the stylesheet. It allows different types of editions that may generate different files based on the rules for each one of the editions. Then, it is necessary to establish a connection between each one of the editions listed in the `evt_builder.xsl` and the @mode attribute values, e.g. establishing that the Diplomatic edition is obtained by applying the transformations that have "dipl" as value for @mode. This allows for different rules for the different types of editions of a text. It is also possible to recall other XSLT to transform only certain fractions of the file. The editors are free to add

their own stylesheets to manage the different levels of the desired edition, tweaking the final appearence to a better fit for their work. For this it is required that the users copy their own XSLTs files into the directory containing all the other stylesheets, include the new XSLT in the `evt_builder.xsl`, and finally establish what is the value of @mode attribute corresponds to the new version. The chain of XSLT has two main types of style sheets, the XML processors, responsible for the processing of the TEI file, and the HTML generators that make the linking between the processed TEI and the corresponding HTML elements.

EVT has a Configuration Generator that makes the adaptation of the tool easier for new contexts. It just requires some general information about the edition that is being added. Among other things it is necessary to specify what are the files that are being used, the main files, sources file and analogue files, which must be at `data` directory, the edition type and level, so it can be Critical, Diplomatic, Interpretative or multiple combinations of the first three. It is then necessary to decide what view modes the edition is going to feature from the ones already mentioned before, what is the initial view mode, the bibliography style, the available tools, like pin entities, image-text linking and entities selector and what are the languages of the texts. Finally, for critical editions only, it is necessary to set what are the level of apparatuses that are going to be shown inline, the tools available, like variant heat map, and finally some advanced XML related settings, where it is possible to define what are the possible lemma filters and possible filters for variants.

The final result of this entire process applied to the source TEI files is a tool that is able to provide all the previously mentioned functionalities for the provided source files.

## 2.8 Juxta Commons

Juxta Commons [84] allows the comparison between two or more witnesses, up to twenty. Each witness is uploaded through a file. After selecting all the witnesses that are meant to be compared they are added to a comparison set. When the set is complete it starts the collation process. It also allows to add files to the set after the collation is done, but the collation must be generated again, which can take a few minutes to complete depending on the length of the chosen files.

Juxta Commons is able to deliver an heat map of a base witness against different witnesses. It colours the different witnesses based on the difference of the witness to the base witness. It is possible to see more details about the differences by clicking in the intended witness. Juxta Commons reports three types of differences, different text at a specified location, text in the witness that is not present at the base witness and text missing in the witness relatively to the base witness. It also allows to have a side-by-side view between the selected witness and the base witness.

It is possible for the users to annotate, i.e. commenting, the texts by clicking in a difference highlighted

by the heat map. The are two ways of annotating, basic witness annotations and regional annotations. The first one allows the user to annotate a specific difference marked in the heat map tied to the base witness. When there is a large set of differences in region of a witness it allows to annotate that region of the witness independently of the base witness. These annotations are saved for the user.

Juxta Commons provides a histogram that offers a macro visualization of the differences between the base witness and a selected witness, being possible to visualize where there are bigger and smaller differences between the witnesses.

For collaboration between users the tool allows users to share comparison sets and witnesses with other users, sending them by URL, email or embedded iframe.

Juxta Commons is composed of three different applications, Juxta Web Service that is a Java application which uses a MySQL database and serves the Juxta Commons, a Ruby on Rails application. Juxta Web Service is based on microservices that are then called by the specified API, also known as the Gothenburg Model. It has three major services, the tokenizing, the collation and alignment and finally, the visualization. Although Juxta Commons has a microservice architecture the logical flow of the data comes close to the typical pipe and filter architecture, starting by the tokenizing, passing by the collation and alignment and ending at the visualization phase, where each of the steps rely on the previous one. The biggest advantage of these tools is the fact that they do not need any special information in the TEI files, the information about the differences between the versions is generated in the pipeline. The Gothenburg Model breaks collation in a set of steps.

- Addition of the raw content, XML or TXT, to the system as a source.
- The sources are transformed into a ”witness”, using XSLTs to transform the XML sources and the TXT are passed through adding some metadata to it.
- These witnesses go through the tokenizer that breaks the text into tokens based on whitespaces and/or punctuation. These tokens are stored in the database as offset/length pairs.
- The diff component is then fed with streams of witnesses tokens, finding the differences between witnesses, using the java-diff-utils from Google.
- The witnesses are then aligned by inserting gaps into the token streaming. These gaps fill the areas where a witness has a lack of content relatively to another witness. Once the tokens are aligned, then ones that have differences are paired and the information is stores in the database.
- Finally, the texts are injected with the information required to be rendered by the presentation application.

Juxta Commons accepts all forms of XML, but has default behaviour for TEI encoded files and Rosseti Archive Markup, which means that there is information that is parsed without additional configuration.

When a witness is loaded into Juxta Commons will automatically apply a XSLT to the file to render the first view. Then it is possible to view the XML of the witness and although it is not possible to directly edit the content of the file it is possible to remove elements that are not relevant for the desired collation, like footnotes. It is also possible to edit the content and create derived witnesses from the same source file and add them to the comparison set.

## 2.9 Lombard Press

LombardPress Web Publication Framework is a reading and analysis interface that provides a place for user accounts, comments, manuscript facsimile viewing, on demand collation and other functionalities. It relies on external services to retrieve texts and images, Sentence Commentary Text Archive (SCTA) API and IIIF API [85] respectively. By using external services to retrieve the information, it is possible to collect data from different sources without any internal change. [86]

Lombard Press offers the possibility to view all the transcriptions available at the texts' archive while providing several functionalities at the paragraph level for the transcribed texts. It is possible to read and leave comments that are specific to the paragraph, visualize manuscript images, see textual variants, notes written directly in the source file, collation of the available witnesses, analyse the source XML code and associated metadata.

LombardPress has its own Schema defined both for critical and diplomatic editions, so for having a working project it is required to adapt the texts to the narrow customization of the general TEI specification defined by LombardPress. This is necessary to assure that the specificities of a text and the relations between texts can be correctly expressed without loss of information.

During the experiments with the tool it was noted that it had a lot of unfinished texts, some without transcriptions, others with transcriptions but without any functionality available, and when the functionalities were available there were some of them whose request ended in a internal server error.

The Lombard Press [86] uses XSLT to generate the information to be stored at the database, but then uses a Ruby engine to access the database and generate the different views. The tool is designed so that it is easily reusable, there are three layers: the Repository Layer where all the data is stored, the SCTA Layer that is responsible for populating the database and for serving the Applications Layer. All the functionalities are implemented in the SCTA Layer that works as a producer and consumer of the database, being then responsible for the serving of the Applications request. The SCTA generates all the necessary HTML for the literary works to be displayed and analysed. The process starts with a `projectdata.xml` file with a list of <item>s where each one, typically, represents a TEI file with its own <teiHeader>. Next it is necessary to standardize all the files, so a XSLT, `rdfextraction.xsl`, is applied to the `projectdata.xml`, and that follows every pointer for the files of the <item>s, that results

| | *LdoD Archive* | EVT | Juxta Commons | LombardPress |
|---|---|---|---|---|
| **Visualization modes** | 4 | 3 | 6 (2 experimental) | 1 |
| **Accepts new files** | No | Yes, but with some effort | Yes | No |
| **Input Files** | TEI custom Schema | TEI standard Schema | Any XML or TXT file | TEI custom Schema |
| **File processing** | Context specific parser | XSLT transformations | XSLT transformations | XSLT transformations |
| **Software Architecture** | Spring Boot | View generated from XSLT transformations applied directly to source file | MySQL + Spring and Restlet + Ruby on Rails | External repository + SCTA layer + presentation layer |
| **Effort for new context** | High | Medium | Low | Medium |

**Table 2.1:** Comparison summary of studied tools.

in a large set of RDF triples in XML to be imported to a server so it can be queried using SPARQL, a RDF query language. At this point the archive can already list the transcription of each of the <item> as ”available”, ”in progress” or ”not yet started” for the interested researcher, but that is not all. If the document is conveniently encoded it is also possible to extract some information like the number of mentions of distinct authors, or the use of titles, references, quotations, or other key words and technical terms or phrases. The RDF are stored in a Fuseki triple store with a frontend written in Ruby. With the use of the RDF.rb library it is possible to navigate the RDF data, make unique connections, and perform searches within and outside the data set. This information is available through an API that can be used by other applications. An example of use beyond the LombardPress Web is Mirador [87], which is another tool to read and study the texts available at the SCTA. There is also SCTA Statistics which reads from the data base with the purpose of collecting statistics about the texts stored in the database.

## 2.10 Comparison with *LdoD Archive*

It makes sense to compare both the functionality and implementation of the three tools that were considered above as the most relevant given the aim of this work. In this comparison it is mandatory to understand what is the necessary amount of effort for the tool to be implemented in a new context.

When visualizing the texts all studied tools provide different view modes, allowing for the comparison between different versions of the text and access the facsimile for that text, with the exception of Juxta Commons which does not allow access to the facsimile. *LdoD Archive*, EVT and Lombard Press provide information about the text under study, e.g. the information about the physical format of the text, like the type of paper and ink used for writing. A feature that is only available for EVT is the linking relation between the manuscript and the transcription. Something exclusive to Juxta Commons is the histogram view, showing where the texts diverge and converge the most.

When it comes to annotating the texts, EVT does not allow any dynamic annotation, being necessary to annotate directly into the source TEI files. The other 3 tools allow annotations in different styles, *LdoD*

*Archive* allows versatile annotation, from a single word to annotation of phrases and sentences, Juxta Commons allows two different types of annotations - those associated associated with the difference between editions and those associated with the text itself. Finally, Lombard Press allows annotation at the paragraph level.

A feature that differentiates *LdoD Archive* from the other tools- which is related to the fact that is being used for an unfinished book, open to interpretation- is the creation of virtual editions, where the user is allowed to select texts and order them at will. To this it further adds the possibility of extending the current texts with versions written by the users.

The adaptation of the tools for a new project requires different levels of effort. Juxta Commons is the easiest tool to adapt to new projects, with a system that processes files and shows them with little effort, with no required schema, accepting xml or txt files. But this has consequences, since the presentation of the files falls short of expectations when compared to the other tools, but clearly the objective of the team was not to build a complex digital archive, but to have a simple and fast tool to compare different versions of texts. The study of this tool was not related with its presentation affordances but with its source file consumption efficiency.

Lombard Press has less flexibility when adapting for new contexts but it still needs less effort than *LdoD Archive*. It is required that the source files follow a specific Schema, in which each edition has its own source file, contrarily to *LdoD Archive*, where the multiple versions must be encoded in the same source file. With Lombard Press the files pass through a consumption process that prepares all the data to be accessed with the presentation application.

Finally, EVT accepts input following standard TEI schema and requires a configuration file for each different edition that is going to be loaded to the archive. Comparing with *LdoD Archive*, the use of a standard TEI schema instead of a context specific TEI schema, as the one currently used in the *LdoD Archive*, allows a faster transition process to different contexts.

As stated in table 2.1 and in the more detailed explanation of the differences between the tools what pops out as a major difference is that, contrarily to all the other tools, *LdoD Archive* never uses XSLT along the process of processing and presenting the texts which may be one way of facilitating the transition of the tool to new contexts. However, *LdoD Archive* presents some features that are not available in any other tool, and for those features there are some compromises to be done, like loosing flexibility for changing the context.

# 3

# Initial *LdoD Archive* Architecture

The *LdoD Archive* is a unique Web Application that connects to several interfaces, from the database to the authentication services of different social networks, as stated in **Figure** 3.1.

The users access it through the browser, where all the previously mentioned features are available. The Web Application allows the users to sign-in using the authentication services provided by Twitter, Google, LinkedIn and Facebook while having its own sign-up system. The citation system uses the Twitter Search API to search for citations of the fragments available in the *LdoD Archive*. The Web Application accesses an Object Oriented Database that stores both the fragments, that are loaded from the TEI Document Repository, and the application data, such as the users and its annotations.

## 3.1 Features

*LdoD Archive* has a set of features, show in **Figure** A.1, that together fulfil the requirements for the operation of the application.

The `About` feature is a simple static page with all the information about the project, and the application itself.

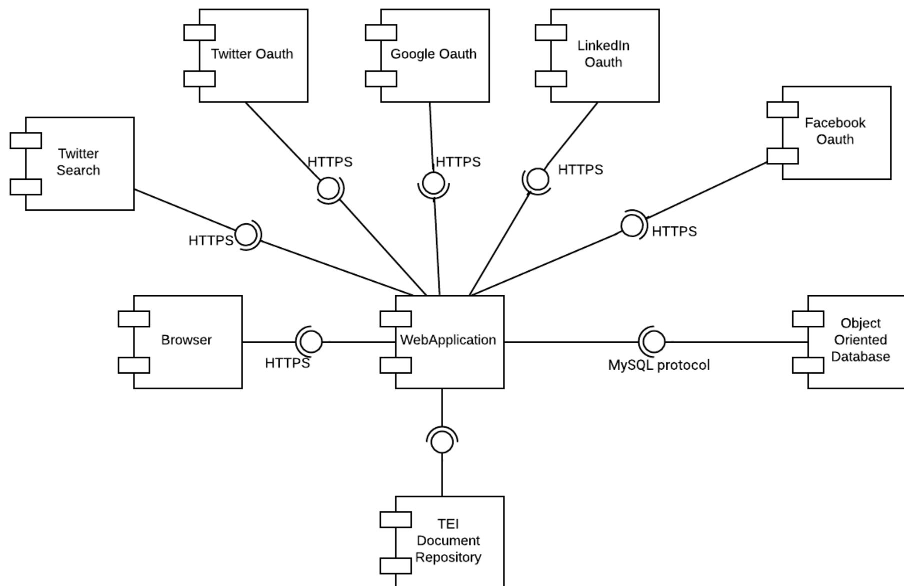


**Figure 3.1:** Component and Connector Diagram of *LdoD Archive*

The `Reading` feature is where a user can read the several fragments from the Book of the Disquiet with the support of the `Recommendation` feature that is able to suggest an order for the fragments that the user is reading, based in a set of parameters, such as, heteronym, date, text and taxonomy.

The `Texts` feature is the core feature of literary digital repositories and it is where the user can see all the texts available in the archive, compare the different versions of the same document and see information about a specific text such as author notes, scholars' notes and meta-information. It is also possible to import and export TEI documents.

The `Editions` feature is responsible for separating the fragments into their correspondent edition, pointing then to the `Text` feature where all the features described above are available for the chosen fragment.

The `Search` feature can be decomposed into two different features, the `Simple Search` and the `Advanced Search`. The `Simple Search` consists of simply searching all the fragments based on a string, while `Advanced Search` is more like a filtered search, where it is possible to search according to all the parameters described in **Figure A.1**, under the `Advanced Search` node.

The `Virtual` features refer to the features related to the Virtual Editions of *LdoD Archive*, where it is possible to annotate a fragment, from a simple character to the full text present in that fragment. It is also possible to classify the fragments according to a user-created taxonomy and browse through all the fragments of a given taxonomy, this can be done automatically through `TopicModelling` feature, which allows to classify the texts automatically according to its content. The work done by the users can be exported to XML and then imported, which works as a backup in case something goes wrong with the

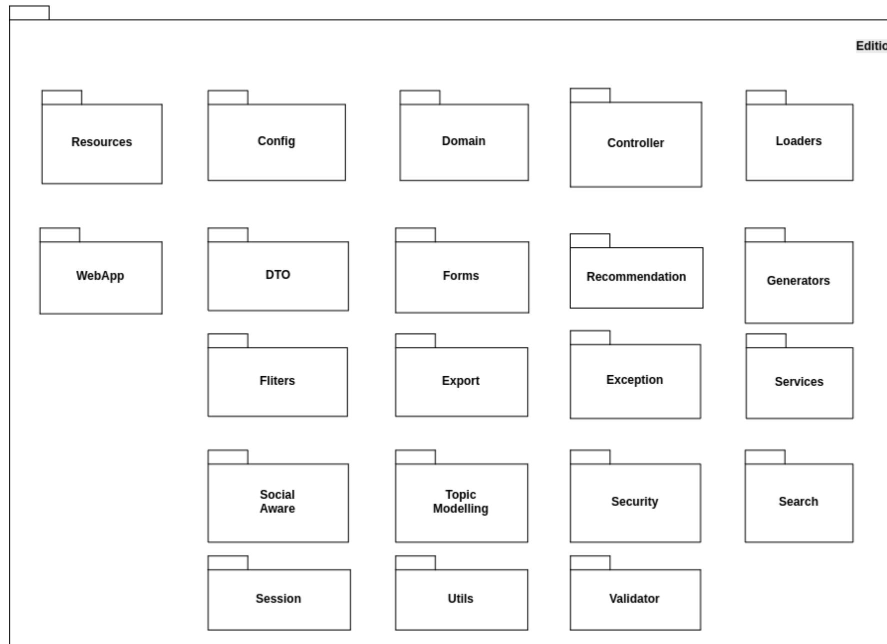


**Figure 3.2:** Package Diagram of *LdoD Archive*

application or the database where the data is stored.

Finally, the `Users` feature provides the possibility for users to create accounts, login, create their own Virtual Editions and write their annotations inside their Virtual Editions.

## 3.2 Packages

The packages show in **Figure** 3.2, apart from the `Domain` package that will be discussed later, are separated and do not have inter-dependencies. Almost every package has a self-explanatory name, such as the `DTO` or the `Exception` packages. Although, there are cases where the same package contains more than one functionality, as in the case of the `Controller` package, that contains all the controllers of the application, that could be easily separated because there are no dependencies among them. It is a case where there is low coupling but there is no high cohesion.

There are packages that are directly associated to a feature, although there are packages that are addressed to the general behaviour of the application, such as the `Config` and the `Filters` packages that have implementations about the configuration of the application itself, the `DTO`, that implements all the necessary DTOs across the application, the `Exception` that implements all the necessary exceptions of the application, the `Validator` and the `Forms` packages that implement all the forms and their validation, the `Security`, as the name suggests implements all the security related aspects of the application. The package `Utils` implements the start-up of the application and the email system.

Apart from the packages mentioned before, there are the ones that directly address a feature. The `Reading` feature and its sub-features are implemented by the code in package `Recommendation`. The `Texts` feature and sub-features are implemented by the `Generators` package, that is the code responsible for generating the HTML that presents the texts, and the packages `Loaders` and `Export` that are responsible for loading and exporting the texts. The `Search` feature is implemented by the `Search` package. The `Virtual` feature and its sub-features are implemented by the `Topic Moddeling`, the `Social Aware` packages and also the `Loaders` and `Export`.

## 3.3 Domain

Due to the use of the Fénix-Framework, and because at the beginning it was decided to have a single project, the domain package contains all the domain entities, which are actually associated to different features.

The domain was a problem because, with all the classes at the same level, there were a lot of crossed dependencies between those. This meant that in order to separate the different features it was not possible to simply move each set of classes of a given functionality to a new module. As shown in **Figure** 3.2 there were some packages designated to a specific feature but there were also packages that contained code from everywhere, e.g. the previously mentioned domain package, which contains classes from all the different features of *LdoD Archive*. This in itself required an analysis of all the classes that depended on each other in order to decide which ones had to be rewritten.

On the domain of *LdoD Archive*, all the classes shown on **Figure** A.2 were present in the same package, the domain package of the application. There were no separation of the classes in any way, which meant that there was no separation between classes responsible for different things. In fact, there was a gigantic class called LdoD that contained code for different responsibilities. For instance, it has methods for finding users and methods for creating Virtual Editions.

# 4

# The new architecure

In order to make *LdoD Archive* a truly reusable framework it is necessary to be able to adapt it according to the necessities of a given project. A way to accomplish that is that features can be added or removed according to what are the goals of the project that intends to use the framework.

The work began by studying the three tools that were considered relevant for this research problem, *LdoD Archive* itself, EVT and Lombard Press, and trying to replicate the last two using the code from *LdoD Archive*, i.e. as result of our work, we intend to define a set o reusable modules, which can be composed to create different variations of digital humanity archives. In particular, we intend to obtain three compositions, *LdoD Archive*, EVT and LombardPress.

## 4.1 Features Decomposition

After studying EVT and Lombard Press it was clear that their features were a subset of the *LdoD Archive* features, so it was possible to decompose *LdoD Archive* into the other two tools in terms of features.

Starting from **Figure** A.1 it is possible to keep only the necessary features to reproduce EVT and LombardPress. EVT only requires the `About` feature and the `Texts` feature and its sub-features. Lom-

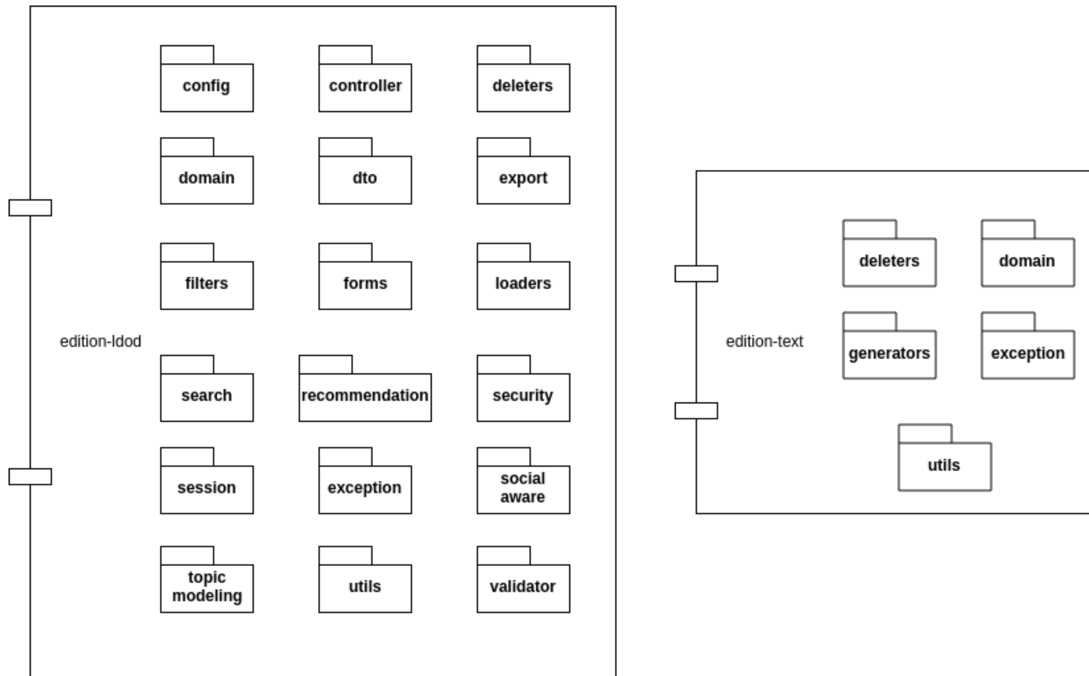


**Figure 4.1:** Decomposition of the packages into the two modules

bardPress keeps the same two features as EVT but also adds the `Annotations` feature, that allows the users to annotate at the paragraph level, which to be accomplished through our tool would require a change in the interface, that would only allow selection at the paragraph level instead of at current character level, and the `Users` feature that is required to be able to use the `Annotations` feature. The **Figure** A.3 illustrate these features selection. The features that are painted yellow are the ones that are common to the three tools, the one with orange colour is the one that *LdoD Archive* and Lombard Press have in common and all the others are exclusive to the *LdoD Archive*.

There is a feature that is the main focus of all three tools, the presence of texts that can be read by the users. Therefore, the module that provides the access to the texts and all the functionality required was considered the basic module that every other module of the application requires.

## 4.2 Packages Decomposition

The packages were separated to the modules that implement the feature that they support, as mentioned in **Section** 3.2.

The main focus was separating the domain, but there are implementations that were needed in the `edition-text` module that were in one of the modules associated with multiple features, such as the `utils` and the `exception` packages, that were divided into the two modules. The **Figure** 4.1 shows the new architecture of packages that was necessary to accomplish an independent module.

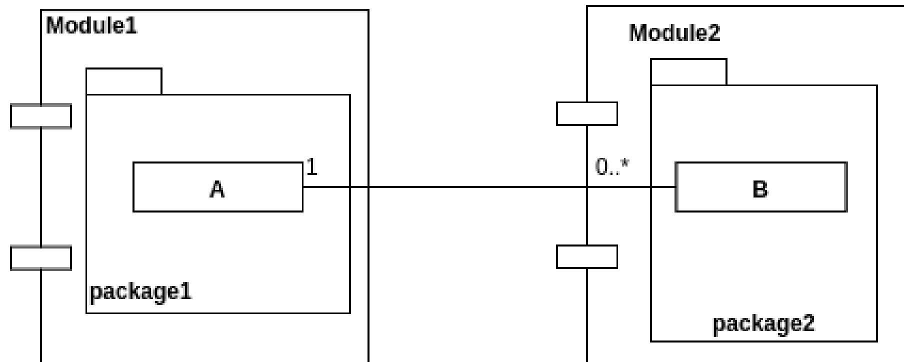


**Figure 4.2:** Uses relation across modules

There are still a few modules, or part of them, that could be moved to the `edition-text` module, such as some of the controllers and some of the loaders and exporters, because there are loaders and exporters specifically to import and export the TEI documents, i.e. the ones encoded in TEI format that are the result of expert's work, and there are loaders and exporters that deal with the Virtual Editions, and only save the work from the users of *LdoD Archive*. Those are the first ones mentioned that should be moved to the `edition-text` module.

## 4.3 Domain Decomposition

Finally, the domain was separated accordingly to the classes that were necessary to implement the features related with the `text` features. Some of these classes had dependencies that needed to be solved before moving them to the new module, but that will be discussed later on this section. The resulting work is described in **Figure** A.4. There are some dependencies between the two modules that will also be discussed later on this section.

### 4.3.1 Relations between classes of different modules

When the modules are created and there are classes in a module that have a relation to a class in a different module it causes a problem. If before there was a ≪*uses*≫ relation between two classes of the domain, after separating the application in modules, and those classes are in two different modules, like what happens in **Figure** 4.2, one of classes will not know about the existence of the relation because that relation will be established in the module that uses the other, i.e. if the relation is specified in `package2` it will not be possible to access it from `package1`.

This happens because the relations are specified in the DML of Fenix-Framework, and the relations between classes when using Fenix-Framework do not work as simple as it is in **Figure** 4.2, but they

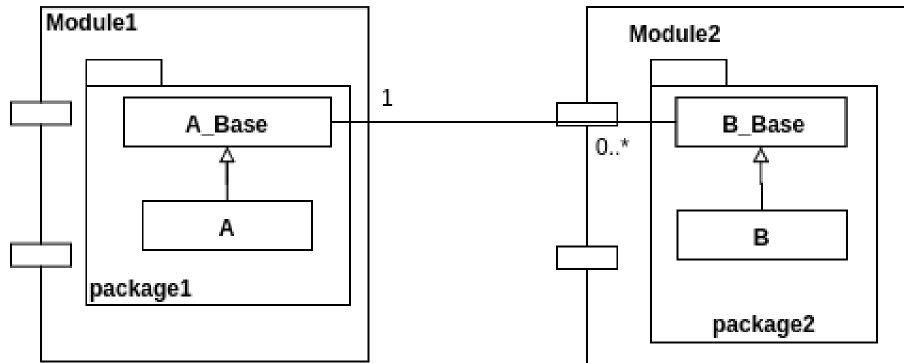


**Figure 4.3:** Uses relation across modules with Fenix-Framework

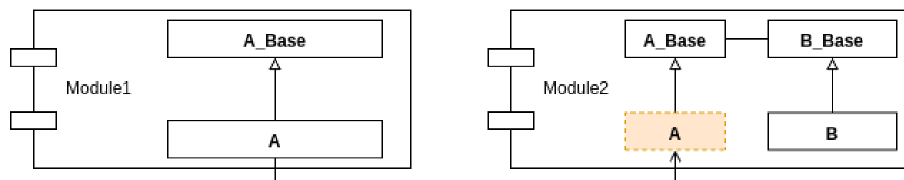


**Figure 4.4:** How classes are recompiled with Fenix-Framework

work as shown in **Figure** 4.3, the relation defined in the DML is generated between the base classes and not the main classes.

Taking the example from the **Figure** 4.3, if the relation is specified in `Module2` as:

```
1  package package2;
2
3  class B;
4
5  relation AhasBs {
6      .package1.A playsRole a {multiplicity 1..1;}
7      B playsRole b {multiplicity 0..*;}
8  }
```

Then `Module2` has a dependency for `Module1`. When `Module1` is compiled, the classes `A` and `A_Base` do not have any code referring the classes `B` or `B_Base`. Later, when module `Module2` is compiled, the class `A_Base` is recompiled and the code implementing the relation between classes `A` and `B` is added, this means that within `Module2` it is possible to invoke the generated methods for the relation but not from `Module1`. In **Figure** 4.4, the class A coloured in orange is exactly equal to the one in `Module1`, while the other classes have been recompiled.

This ensures that the code about a relation created by a new module is kept only where the relation

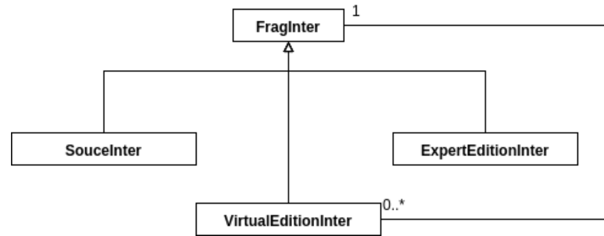


**Figure 4.5:** Relation between the class `FragInter` and its subclasses

exists. One of the problems found in the code, due to this issue, is the deletion of an object, because Fenix-Framework only allows to remove an object from database after disconnecting himself from all the object it is connected with. This is a special case that deserves a closer look.

### 4.3.2 *LdoD Archive* case

For the new architecture it is proposed a solution for a reusable framework that can easily add or remove features from the application without any major concern, our focus being, for now, the partition of the domain.

In order to keep the Web Application as the only application running to provide all the features of LdoD the implementation of the *LdoD Archive* should contain all the modules resulting from the decomposition.

Therefore, considering that the `edition-text` module is the core module to be included in any composition, we start by breaking the application into two modules, `edition-text` and `edition-ldod`, where the `edition-ldod` will be decompose in future steps, e.g. it includes the `Users` and `Virtual` features.

When trying to create this two independent modules the problems described previously in this chapter were raised in some occasions, the classes that were planned to stay in *edition-text* module contain methods that only exist because of the virtual features and there are methods of the Virtual Edition features that depended on classes from the `edition-text` modules which know the existence of the virtual classes. We have also identified situations where the existing abstraction hierarchy is not adequate to the decomposition. All these problems can be summed up in problems related with the classes hierarchy and the relations between those classes.

#### 4.3.2.A Relations among classes from different modules

Related to the problem described in Section 4.3.1, in *LdoD Archive* there was the relation between four classes, `FragInter`, `ExpertEditionInter`, `SourceInter` and `VirtualEditionInter`. The relations among them were of inheritance and usage, are described in **Figure 4.5**.

The class `VirtualEditionInter` inherits from the `FragInter` class but also uses this class, i.e. each `VirtualEditionInter` object uses a `FragInter` object. This can create a chain of uses that are made of

FragInter classes, where all the classes, except the last one in the chain, are VirtualEditionInter, and the last class in the chain is either SourceInter or ExpertEditionInter.

The following code was taken from the class *ExpertEditionInter* before the changes in the code.

**Listing 4.1:** Methods of ExpertEditionInter used by VirtualEditionInter

```
@Override
public FragInter getLastUsed() {
    return this;
}

@Override
public List<FragInter> getListUsed() {
    List<FragInter> listUses = new ArrayList<>();
    return listUses;
}

@Override
public int getUsesDepth() {
    return 0;
}
```

There are places in the code, like the HTML writers, that require the object at the end of that chain to be used, i.e. a SourceInter or a ExpertEditionInter. That explains the implementation of the method getLastUsed, in this two classes, to return the object itself. In order to allow this implementation, the getLastUsed method was declared as abstract in the superclass FragInter requiring the three subclasses to implement the method. After the separation into two modules, this method could not continue to exist on the classes that were kept at the edition-text module because their goal is related to the virtual edition features.

The alternative was to rewrite the method in the VirtualEditionInter class. Once the method was always called from that class it only needed to be rewritten there without relaying on the implementation of the other subclasses of FragInter. The code before the changes was a simple recursive method, that would stop when called in a SourceInter or ExpertEditionInter as shown in Listing 4.1. The following code shows the implementation of the getLastUsed method in the VirtualEditonInter class.

**Listing 4.2:** Method `getLastUsed` in the `VirtualEditionInter` class before the changes

```
@Override
public FragInter getLastUsed() {
    return getUses().getLastUsed();
}
```

After the change the implementation changed to the following:

**Listing 4.3:** Method `getLastUsed` and its auxiliary method in the `VirtualEditionInter` class after the changes

```
public ScholarInter getLastUsed() {
    if (usesVirtualEditionInter()) {
        return ((VirtualEditionInter) getUses()).getLastUsed();
    }
return (ScholarInter) getUses();
}

public boolean usesVirtualEditionInter() {
    if (getUses() instanceof VirtualEditionInter) {
        return true;
    }
    return false;
}
```

To be able to take the method from all the `edition-text` module classes it is necessary to use `instanceof` and cast, because the uses relation is between the super-classes but the resulting chain of objects is very specific, only the last one is not a `VirtualEditionInter`. Therefore, an auxiliary method was created that checks if the `FragInter` that the `VirtualEditionInter` is using is another `VirtualEditionInter` and in that case, returns it, or returns null otherwise.

It was also created a new subclass of `FragInter` that is called `ScholarInter`. This new class is now the superclass of `SourceInter` and `ExpertEditionInter`. This class replaced the use of the `FragInter` when the class that was intended to be used was either `SourceInter` or `ExpertEditionInter`. This classed is now used in the HTML writers, that before accepted `FragInter` objects, which should be from one of the two classes mentioned before.

Once the relation defined for the `VirtualEditionInter` is with `FragInter`, the method `getUses` of the classe `VirtualEditionInter`, generated by Fenix-Framework, returns a `FragInter`. When calling the HTML writers, that require a `ScholarInter` as argument the method `getLastUsed` is called to obtain the last `FragInter` of the chain, which is a `ScholarInter`. This way the cast to the class `ScholarInter`

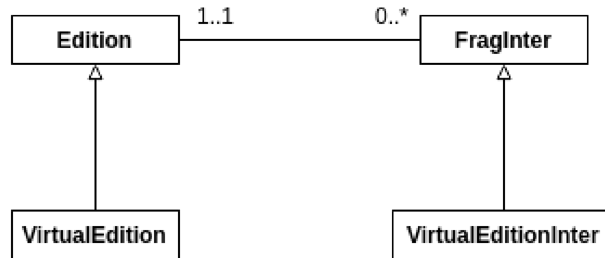


**Figure 4.6:** Relation between the classes `Edition`, `VirtualEdition`, `FragInter` and `VirtualEditionInter`

is done encapsulated inside the method `getLastUsed`, that is responsible to go through the chain and returned the last one already casted into its true class.

Another problem related with the relation between classes from different modules is the relation between `VirtualEdition` and `VirtualEditionInter`. The relation between these two classes is, as shown in **Figure** 4.6, obtained by their super-classes. This means that the getters generated by Fenix-Framework of these classes are from their super-classes, and they return objects of the super-class type.

It is not possible to override these methods because they return a set, and it is not possible to override a method replacing the return type from a set of one type to a set of another type. There are cases where it is necessary to invoke methods that are exclusive of the `VirtualEditionInter` class. In those cases, after getting the set of `FragInter`, they must all be cast to `VirtualEditionInter`, as shown in the Listing 4.4 in lines 7 and 8 because by construction, every `FragInter` related with a `VirtualEditionInter` is a `VirtualEditionInter`.

**Listing 4.4:** `VirtualEditionController`

```
1  (...)
2  VirtualEdition virtualEdition = VirtualManager.getInstance().getVirtualEdition(
       acronym);
3  (...)
4  else {
5      String intersFilesPath = PropertiesManager.getProperties().getProperty("
           inters.dir");
6      List<TranscriptionDTO> transcriptions = new ArrayList<>();
7
8      virtualEdition.getIntersSet().stream()
9              .map(VirtualEditionInter.class::cast).collect(Collectors.toList()).
                   forEach(inter -> {
10         FragInter lastInter = inter.getLastUsed();
```

```
11          String title = lastInter.getTitle();
12          String text;
13          try {
14              text = new String(
15                      Files.readAllBytes(Paths.get(intersFilesPath + lastInter.
                          getExternalId() + ".txt")));
16          } catch (IOException e) {
17              throw new LdoDException("VirtualEditionController::getTranscriptions
                    IOException");
18          }
19
20          transcriptions.add(new TranscriptionDTO(title, text));
21      });
22
23      return new ResponseEntity<>(new EditionTranscriptionsDTO(transcriptions),
            HttpStatus.OK);
24  }
25
26  (...)
```

#### 4.3.2.B Delete Object

A particular situation where the relations between different modules was a problem was deleting objects. The code to remove an object of the class `Fragment` was the following:

**Listing 4.5:** Method `remove` of the class `Fragment` before refactoring

```
1       @Atomic(mode = TxMode.WRITE)
2       public void remove() {
3           setLdoD(null);
4           getTextPortion().remove();
5           for (VirtualEditionInter inter : getVirtualEditionInters()) {
6               inter.remove();
7           }
8           for (FragInter inter : getFragmentInterSet()) {
9               inter.remove();
10          }
11          for (Source source : getSourcesSet()) {
```

```
12              source.remove();
13          }
14          for (RefText ref : getRefTextSet()) {
15              ref.remove();
16          }
17          getCitationSet().stream().forEach(c -> c.remove());
18          deleteDomainObject();
19      }
```

With the new architecture, the `VirtualEditionInter` class and the `Citation` class are not part of the `edition-text` module anymore, which means that their relation with the `Fragment` class was also moved to a different module. In this case, the methods `getVirtualEditionInters` and `getCitationSet` were not accessible from within the `Fragment` class, and it did not make sense to be, because it would cause a dependency from the `edition-text` module to the new modules and it should not know about the existence of such modules.

This type of situation happens not only in the `Fragment` class but also in the `FragInter`, `ExpertEditionInter`, `SourceInter` and `SimpleText` classes. When those classes were removed there were relations that must be deleted, but it was not possible to do such thing from these classes.

The solution found to solve these problems was the injection of a class that would know about the new relations created within the new modules. Classes responsible for the removal were created with the suffix ”Deleter”, in this case, `FragmentDeleter`, `FragInterDeleter`, `ExpertEditionInterDeleter`, `SourceInterDeleter` and `SimpleTextDeleter`. The injection directly into the respective classes was not possible because of conflicts between the Spring annotation `@Component` and the classes generated by the Fenix-Framework. The solution was to create a static inner class that would have the Deleter injected.

The code inside the `Fragment` class became the following:

**Listing 4.6:** Method `remove` of the class `Fragment` after refactoring

```
1  @Atomic(mode = TxMode.WRITE)
2  public void remove() {
3      Remover.remove(this);
4      deleteDomainObject();
5  }
6
7  @Component
8  public static class Remover {
9      public static FragmentDeleter fragmentDeleter;
```

```
10     @Autowired
11     Remover(FragmentDeleter fragmentDeleter) {
12         this.fragmentDeleter = fragmentDeleter;
13     }
14     public static void remove(Fragment fragment) {
15         fragmentDeleter.remove(fragment);
16     }
17 }
```

Using an inner class maintains the cohesion of the code, the invocation of the `remove` method for the `Fragment` is in the `Fragment` class. The code related to the relations within the `edition-text` module was moved to the class `FragmentDeleter` while the code related to the relations outside the module was moved to the respective module and called `FragmentDeleterVirtual` that inherits from the `FragmentDeleter` class.

The class `FragmentDeleter` has the following code:

**Listing 4.7:** `FragmentDeleter` class

```
1  public class FragmentDeleter {
2      public void remove(Fragment fragment) {
3          fragment.setCollectionManager(null);
4          fragment.getTextPortion().remove();
5          for (FragInter inter : fragment.getFragmentInterSet()) {
6              inter.remove();
7          }
8          for (Source source : fragment.getSourcesSet()) {
9              source.remove();
10         }
11         for (RefText ref : fragment.getRefTextSet()) {
12             // the reference is removed
13             ref.remove();
14         }
15     }
16 }
```

As it is possible to observe, it contains the code that previously was on the `remove` method from `Fragment` class except the methods that refer to the `VirtualEditionInter` and `Citation` classes.

The class `FragmentDeleterVirtual` has the following code:

**Listing 4.8:** `FragmentDeleterVirtual` class

```
1  public class FragmentDeleterVirtual extends FragmentDeleter {
2      @Override
3      public void remove(Fragment fragment) {
4          fragment.getFragmentInterSet().stream()
5                  .filter(VirtualEditionInter.class::isInstance)
6                  .map(VirtualEditionInter.class::cast)
7                  .collect(Collectors.toList())
8          .forEach(inter ->
9                      inter.remove()
10             );
11         fragment.getCitationSet().stream().forEach(c -> c.remove());
12         super.remove(fragment);
13     }
14 }
```

This is the class that actually is being injected. At line 12 it calls its super class to complete the removal. The method in this class is different from the initial code from the `remove` method of class `Fragment` because the method `getVirtualEditionInters` had to be moved because it could not continue inside the class `Fragment`. That method was replaced by the procedure that goes from line 4 to line 10, and is responsible for filtering all the `VirtualEditionInter` from the relation and delete them.

All the previous code about deleters does not make sense without defining the injection correctly. A `DeletersConfig` class was created inside the Spring-Boot Application configuration package to choose which class to inject for each composition. The code is the following:

**Listing 4.9:** `DeletersConfig` class

```
1  @Configuration
2  public class DeletersConfig {
3      @Bean
4      public FragmentDeleter fragmentDeleter() { return new FragmentDeleterVirtual
           (); }
5
6      @Bean
7      public ExpertEditionInterDeleter expertEditionInterDeleter() { return new
          ExpertEditionInterDeleterVirtual(); }
8
9      @Bean
```

```
10      public SourceInterDeleter sourceInterDeleter() { return new
            SourceInterDeleterVirtual(); }
11
12      @Bean
13      public FragInterDeleter fragInterDeleter() { return new
            FragInterDeleterVirtual(); }
14
15      @Bean
16      public SimpleTextDeleter simpleTextDeleter() { return new
            SimpleTextDeleterVirtual(); }
17  }
```

In this composition, the deleters returned are always the ones from the `VirtualEdition` package. In other compositions it may make sense to implement new deleters that may replace or extend the ones used now, which means that this solution is extensible.

#### 4.3.2.C Further decompositions

Beyond the work already done, there are still some modules that could be created to increase the modularity of the *LdoD Archive*.

The first one is the creation of the `edition-virtual` module. This allows the `edition-ldod` module to be a truly composition of modules that implement features, with only the application specific implementation. This may happen at the same time that the `edition-user` module, a module that implements the `Users` feature, is created. Considering this situation, `edition-ldod` would be the composition of three modules, `edition-text`, `edition-user`, and `edition-virtual`.

However, the decomposition process can go even further, there are features inside the `edition-virtual` module that can be extracted to their own module. This is the case of the `Annotation`, the `Reading`, the `Search`, the `Aware` and the `TopicModelling` features. Each one of these features should have their own module. The `edition-search` and `edition-topicmodelling` modules do not have any domain class. After this decomposition there are nine modules, eight related with features and the one responsible for the composition of the features modules.

The proposed domain decomposition is show in **Figure** A.5. There are three modules that are not represented because, as mentioned before, there are modules that do not have any domain classes associated, which is the case of `edition-ldod`, `edition-search` and `edition-topicmodelling`. The remaining modules are show in the figure.

The final composition of packages are shown in **Figure** A.6. Apart from `edition-ldod` all the other eight modules implement a feature or a small set of features that were represented in **Figure** A.1.

This modules are not totally independent from each other, although they implement well defined features there are still dependencies that need to exist. The base module is `edition-text` that do not have any dependency. The module `edition-user` does not have any dependency either, but does not have any relevance for the work without other modules.

Then, there is the `edition-reading` module that is able to implement its features with only the presence of the base module.

The `edition-virtual` module requires the `edition-text` and the `edition-user` to provide its features.

There is the `edition-annotation` module that depend on the presence of the `edition-virtual` module. It works over a Virtual Edition. The annotations and taxonomy are only written over fragments of a Virtual Edition, the `VirtualEditonInter` objects.

The `edition-aware` module depends on the `edition-annotation` module, because the citations are written in the texts using the annotation feature.

The `edition-reading` is a module that provides two different features depending on the module that is using it. If used together with `edition-text` it offers the possibility of reading the fragments using an order accordingly to the defined weights. If used together with `edition-virtual` it allows to order the fragments of a Virtual Edition accordingly to the same criteria as the `Reading` feature.

There is a module that is different from the others, the `edition-search` implementation depends on the compositions where it is involved. The several sub-features of the `Advanced Search` depend on other features of *LdoD Archive*, therefore, it should only offer the options that correspond to the chosen modules for the composition in use.

Another extension of the `edition-annotation` module is the `edition-topicmodeling`. This module generates the taxonomy for a Virtual Edition through an algorithm that goes through the fragments and labels them according to their content.

Finally, the `edition-ldod` is the composition of the previously mentioned modules. This module is responsible for implementing the specificities of each project. In the case of the *LdoD Archive* this module is responsible for things such as the configuration of the deleters that should be called, i.e. if the `edition-virtual` module is present then the application should use the deleters associated to that module.

# 5 Presentation of *LdoD Archive*

When analysing how to redesign the interface so that it becomes more composable it was clear that it could not continue to use JSPs as the technology responsible for it, because it is strictly coupled with the Spring application, and it is difficult to replace them according to the needs of the composition.

The solution found to make the interface easy to redesign or even replace was to change all the current JSPs to a REST API that would be used by a JavaScript implementation of the user interface, because it is easily replaced by any other new implementation that may be developed and even accept multiple versions of the presentation to be served simultaneously.

Therefore, it is possible to have a default interface defined but anyone can easily redefine the interface without changing any implementation in the core project, i.e. once the API is defined anyone can design a new user interface that performs requests to the endpoints without any concerns about the implementation of the Spring application. This way, the implementation of the user interface is completely independent from the application implementation, where the only concern when developing the user interface is to make sure the endpoint performs the intended task.

It was decided that the default interface would be developed using ReactJS. This interface would have the same look and feel but with a different technology.

**Figure 5.1:** Division between ReactJS elements and JSP content

The problem related with such decision is that something like this takes an huge amount of time to implement. The process of moving every user interface to the new technology would take a lot of implementation time.

To facilitate the transition between technologies, following an incremental strategy, it was developed a ReactJS application that replicates the original user interface but uses the original content supported by the JSPs.

This only required some minor changes in the original application's JSPs. All the common elements of the pages were generated by the ReactJS application, which means that they could be removed from the JSPs. This way, the JSPs kept only the central element of every page, while the headers, menu and footers were already in the ReactJS side. In **Figure** 5.1, the ReactJS elements are the ones highlighted in green and the content from the JSP is the one highlighted in blue.

When developing the interface in ReactJS a particular attention was given to the modularity of the user interface. To match the possibility of the application having different modules according to the

composition that was done, the header and the menu is generated from an array that can be defined according to each specific composition. At the moment the array is hardcoded but can be easily replaced by a REST call.

Each element of the menu is generated using the code in **Listing** 5.1. First, it is necessary to check if the menu element that is going to be created has subsections, if not, it is created a new `Link` that contains the title of the element and sets the location of the page that element links to.

If the menu element has subsections, then it is necessary to define what are the base link of those subsections, i.e. if the subsections links are of the type `about/subsection`, where the subsection is any subsection of the about element, then the base link is `about/`. Then, for each subsection of the menu element, it is generated a `MenuItem` that needs to be surrounded by a `LinkContainer` to be possible to indicate where that element is going to navigate when clicked. If the list of subsections need a divider between any of its elements it should have the index of where the divider should be placed passed as the `division` in its properties. Finally, it is returned a `NavDropdown` element that contains all the generated subsections inside.

**Listing 5.1:** Code to generate each menu element

```
1  function TopBarElement(props) {
2      if (!props.subsections) {
3          return (
4              <li>
5                  <Link key={props.baseLink} to={`/${props.baseLink}`}>{props.title
                       }</Link>
6              </li>
7          );
8      }
9      let baseLink = '';
10     if (props.baseLink) {
11         baseLink = `/${props.baseLink}`;
12     }
13     const subsecs = props.subsections.map(subsection =>
14         (<LinkContainer key={subsection.link} to={`${baseLink}/${subsection.link
               }`}>
15             <MenuItem key={subsection.link}>{subsection.title}</MenuItem>
16         </LinkContainer>),
17     );
18     if (props.division) {
```

```
19          subsecs.splice(props.division, 0, <MenuItem key={0} divider />);
20      }
21      return (
22          <NavDropdown key={props.baseLink} title={props.title} id={'Navigation
                Menu'}>
23              <div className={'dropdown-menu-bg'} />
24              {subsecs}
25          </NavDropdown>
26      );
27  }
```

The elements to be placed in the menu need to have its title, what is the base url of the endpoint and its subsections if it has any. Each of the subsections has a title and the corresponding endpoint.

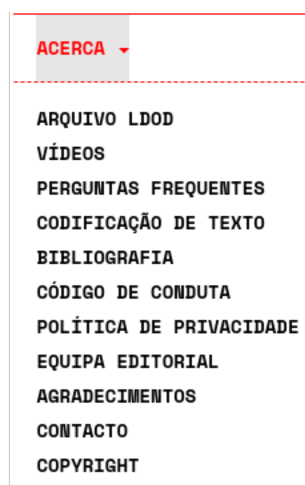


**Figure 5.2:** Menu element About

In this case, a menu element like the one in **Figure** 5.2 would be generated by the following code:

**Listing 5.2:** Code to generate the About menu element

```
1  <TopBarElement
2      title={<FormattedMessage id={'topBar.about.title'} />}
3      baseLink={'about'}
4      subsections={[
5          { title: <FormattedMessage id={'topBar.about.archive'} />, link: 'archive
               ' },
6          { title: <FormattedMessage id={'topBar.about.videos'} />, link: 'videos'
               },
```

```
7           { title: <FormattedMessage id={'topBar.about.faq'} />, link: 'faq' },
8           { title: <FormattedMessage id={'topBar.about.encoding'} />, link: '
                encoding' },
9           { title: <FormattedMessage id={'topBar.about.articles'} />, link: '
                articles' },
10          { title: <FormattedMessage id={'topBar.about.conduct'} />, link: 'conduct
                ' },
11          { title: <FormattedMessage id={'topBar.about.privacy'} />, link: 'privacy
                ' },
12          { title: <FormattedMessage id={'topBar.about.team'} />, link: 'team' },
13          { title: <FormattedMessage id={'topBar.about.acknowledgements'} />, link:
                 'acknowledgements' },
14          { title: <FormattedMessage id={'topBar.about.contact'} />, link: 'contact
                ' },
15          { title: <FormattedMessage id={'topBar.about.copyright'} />, link: '
                copyright' },
16      ]}
17  />
```

The application is already developed to support multiple languages. That is the reason for the title field in each element being a `FormattedMessage`, that is then fetched accordingly to the language selected by the user, or the one from the locale of the browser if none is selected. These messages are located in the `Resources` package of the ReactJS application. The multiple versions of each message are written in the same file, a json file, that is simply a json object that is composed of multiple json objects, where each of these objects translate the messages to a language. In the case of the *LdoD Archive*, there are three json objects, `'pt'`, `'en'` and `'es'`, where inside of each one of these objects is the translation for each `FormattedMessage`.

This feature is supported by the use of `IntlProvider` element from the `react-intl` library. To be possible to change the language of the application during its use, the `IntlProvider` element was extended by a custom element called `UpdatableIntlProvider`. This classe was defined as shown in **Listing** 5.3

**Listing 5.3:** Base code of the `UpdatableIntlProvider`

```
1  export default class UpdatableIntlProvider extends React.Component {
2  (...)
3      constructor(props) {
4          super(props);
```

```
5          // Define user's language. Different browsers have the user locale
               defined
6          // on different fields on the `navigator` object, so we make sure to
               account
7          // for these different by checking all of them
8          const language = (navigator.languages && navigator.languages[0]) ||
               navigator.language;
9          // Split locales with a region code
10         const languageWithoutRegionCode = language.toLowerCase().split(/[_-]+/)
               [0];
11         // Try full locale, try locale without region code, fallback to 'en'
12         const messages = localeData[languageWithoutRegionCode] || localeData[
               language] || localeData['pt-PT'];
13         this.state = {
14             locale: language,
15             myMessages: messages,
16         };
17     }
18 (...)
19     render() {
20         return (
21             <IntlProvider
22                 messages={this.state.myMessages}
23                 locale={this.state.locale}>
24                 {this.props.children}
25             </IntlProvider>
26         );
27     }
28 }
```

By default, the `IntlProvider` element loads a set of messages when mounted. To extend this behaviour it was necessary to change the loaded messages after the element is already mounted. A function was defined, `updateLocale`, in **Listing** 5.4, that receives a new locale as argument, loads the messages accordingly to the selected locale and changes the state of the element. This change of state generates a rerender of the element, this time with the messages of the selected locale.

**Listing 5.4:** `updateLocale` of the `UpdatableIntlProvider` element

```
1 updateLocale = (locale) => {
```

```
2       const myMessages = localeData[locale];
3       this.setState({ locale, myMessages });
4   }
```

The `updateLocale` function is passed down to the element that generates the language toggle by context, as shown bellow:

**Listing 5.5:** Passing `updateLocale` function to the context

```
1   static childContextTypes = {
2       updateLocale: PropTypes.func,
3   };
4
5   getChildContext() {
6       return { updateLocale: this.updateLocale };
7   }
```

To the language toggle element be able to access the function from its context it must be a child of the `UpdatableIntlProvider`. For all the elements to be able to access the translated messages they also need to be a child element of the `UpdatableIntlProvider` element. To accomplish this, the base element of the whole application is the `UpdatableIntlProvider`, as shown bellow:

**Listing 5.6:** Base element of the ReactJS application

```
1   function App() {
2       return (
3           <UpdatableIntlProvider>
4               <Router>
5                   <div>
6                       <TopBar />
7                       <Switch>
8                           <Route exact path={'/'} render={() => <LegacyPage url={'/
                                '} />} />
9                           <Route render={props => <LegacyPage url={props.location.
                                pathname} />} />
10                      </Switch>
11                  </div>
12              </Router>
13          </UpdatableIntlProvider>
```

```
14      );
15  }
16
17  ReactDOM.render(
18      <App />,
19      document.getElementById('root'),
20  );
```

To reuse the JSPs from the original Spring Application it was necessary to be able to reuse the HTML code that was returned when accessing the application. The content for each request is fetched by a `LegacyPage` element. The returned content from the Spring Application is a raw HTML.

In ReactJS it is no possible to directly write HTML to be displayed, therefore, it is necessary to parse the raw HTML and generate the currespondent React elements.

To do this, it was used another ReactJS library, `react-html-parser`, that was extended to match the necessary behaviour for this specific case. There were specific cases that needed to be handled beyond the simple conversion from HTML to ReactJS element. The special cases were the links inside the HTML, that need a special attention in ReactJS, because the history of visited pages is handled by the `react-router-dom` library, and the navigation is only possible to be done using the `Link` element from the library. So, every `href` from the original HTML needed to be converted into a `Link` element of the `react-router-dom` library, as shown in the code below:

**Listing 5.7:** Code to handle the `href` inside the legacy HTML

```
1  function transform(node) {
2      if (node.type === 'tag' && node.name === 'a' && node.attribs.href && node.
           class !== 'infobutton') {
3          if (node.attribs.href !== '#') {
4              return (<Link
5                  to={node.attribs.href}>
6                  {node.children.map((child, index) => convertNodeToElement(child,
                       index, transform))}
7              </Link>);
8          }
9          return (
10             <a
11                 className={node.attribs.class}
12                 onClick={() => eval(`window.${node.attribs.onclick}()`)}>
13                 {node.children.map((child, index) => convertNodeToElement(child,
```

```
                    index, transform))}
14          </a>
15      );
16  }
17
18 return undefined;
```

The code in **Listing** 5.7 is a function from the element `customHTMLParser` that is called at every HTML node from the HTML fetched. If it matches any of the conditions in the `transform` function it returns the ReactJS element that should replace such element from the HTML. If `undefined` is returned, then the node is simply converted to the equivalent ReactJS element.

After the HTML being parsed it is only necessary to add it to the `LegacyPage` to be rendered. The render code of the `LegacyPage` is the following:

**Listing 5.8:** `render` function of `LegacyPage`

```
1  render() {
2      const { error, isLoaded, html } = this.state;
3      const parsedHTML = customHTMLParser(html);
4      if (this.state.error) {
5          return <div>Error: {error.message}</div>;
6      } else if (!isLoaded) {
7          return <div>Loading LdoD...</div>;
8      }
9      return (
10         <div className={'container ldod-default'}>
11             {parsedHTML.props.children}
12         </div>
13     );
14 }
```

The HTML content from the application is fetched through the following code:

**Listing 5.9:** Fetch of the HTML from the Spring Application

```
1  class LegacyPage extends React.Component {
2
3      static baseURL = 'http://1.1.1.10:8080';
4
```

```
5       constructor(props) {
6           super(props);
7           const langConst = `/?lang=${props.intl.locale.split(/[_-]+/)[0]}`;
8           this.state = {
9               error: null,
10              isLoaded: false,
11              html: '',
12              lang: langConst,
13              url: LegacyPage.baseURL + props.url,
14          };
15      }
16
17      htmlRequest(url) {
18          fetch(url)
19              .then(res => res.text())
20              .then(
21                  (result) => {
22                      this.setState({
23                          isLoaded: true,
24                          html: result,
25                      });
26                  },
27                  (error) => {
28                      this.setState({
29                          isLoaded: true,
30                          error,
31                      });
32                  });
33      }
34
35      componentDidMount() {
36          this.htmlRequest(this.state.url + this.state.lang);
37          window.scrollTo(0, 0);
38      }
39  (...)
```

To make sure that the requested page is in the same language as the ReactJS application, at the end of every request a query is added to fetch the page in the correct language, as seen in **Listing** 5.9 in Line 7 where the query string is built, the Line 12 where it is added to the state of the element and

finally in Line 36 where it is passed as part of the url to fetch the HTML.

There were some problems that were not solved in the ReactJS application.

The first one is the use of sessions, not being possible to login in the application, which means that it is not possible to use any feature that requires a user to be logged in. To solve this it is necessary to implement all the authentication system in the ReactJS side.

The other problem is related with pages that have JavaScript written within the HTML file returned by the Spring application. When a page requires some code of JavaScript to work there were problems that were not entirely solved. To try solving this problem it was used another ReactJS library called `react-helmet` that allows to add JavaScript code to the head of HTML pages to be loaded. The custom JavaScripts written within the Spring application, and the JavaScript libraries, such as `JQuery` and `bootstrap` were successfully loaded sometimes, but sometimes there were problems loading those. The only definitive solution is to implement the pages that require JavaScript code to work directly in ReactJS. So, these pages are the main candidates to be the first ones to be rewritten to ReactJS once the effort required to adapt them to be used directly from the HTML is at least the same as implementing from scratch in ReactJs.

In short, it was achieved a user interface that reuses what was already implemented so it is possible to build in a incremental strategy, starting from here. Although there are a few pages of the interface that could not be simply adapted into the new implementation. Since the long term objective is to implement the whole interface with this technology, the pages that cause problem are the perfect candidates to begin that transition.

# 6 Validation

The new LdoD Archive architecture worked as planed, i.e. the application kept its behaviour and features while becoming a more modular solution. The new user interface developed proved that it was possible to have an interface that support the transition between the use of JSPs and the creation of the REST endpoints, but it still has problems in some pages of LdoD Archive that require JavaScript code. The new user interface is the only work done till now that cannot be used in production environment.

Changing the LdoD Archive architecture required some reasoning over the implementation and the dependencies that existed. This study allowed to define a set of techniques that can be applied to the different type of dependencies between classes of the domain. These techniques are explained through **Section** 4.3.

We intend to validate our results by identifying the dependencies that occur in each of the compositions in **Section** 4.3.2.C and analyse whether the techniques developed in this work will also apply to these new cases.

## 6.1 `edition-user` module

The only domain class of the `edition-user` module that has relations with classes of other modules is the `LdoDUser` class. It is used by the classes `RecommendationWeights` from `edition-reading` module, `Member` and `VirtualEdition` from `edition-virtual` and `Tag` and `Annoation` from `edition-annotation` module.

The relations mentioned before, from the point of view of the `LdoDUser`, only requires a small change in the class to be extracted to the module. It should have apply the same technique as used in **Section** 4.3.2.B, creating a `LdoDUserDeleter` that would then be extended in the modules that use this class to assure that all the objects associated with the `LdoDUser` object being deleted are also deleted.

## 6.2 `edition-annotation` module

This module has some relations with classes from other modules, the module depends on other modules but there are also other modules that depend on this one.

The class `Taxonomy` depends on the classes `VirtualEdition` from the `edition-virtual` module and `LdoDUser` from `edition-user` module. The class `Tag` depends on the `edition-virtual` module class `VirtualEditionInter`. Finally, the class `Annotation` depends on the class `LdoDUser` from the `edition-user` module.

With the mentioned dependencies it is necessary to implement three different deleters, the `VirtualEditionDeleter`, the `VirtualEditionInterDeleter` and the `LdoDUserDeleter`.

## 6.3 `edition-aware` module

The `edition-aware` has a class that extends a class from the `edition-annotation` module and depends on the classes `Fragment` and `FragInter` from the `edition-text` module, `VirtualManager` and `VirtualEdition` from the `edition-virtual` module.

This case of inheritance is different from the one explained in **Section** 4.3.2.A because there is no other relation between the class and its superclass. The only problem to solve in this case is the need to implement deleters for the classes that this modules depend on, `Fragment`, `FragInter` and `VirtualEdition` except for the `VirtualManager` because the object of this class is never deleted.

## 6.4 `edition-search` module

The only identified case wher is not possible to create an individual module that implements its features applying the developed techniques is the `Search` feature. This feature is a special case that depends directly from the composition of the application. In this case it is not a problem of the domain, but a problem in the implementation of the feature itself.

The feature was implemented according to the features available in the LdoD Archive, but after the decomposition it may happen that not all features are present in the composition. In the current implementation, the `search` package contains the implementation to perform the advanced search for all the features represented in **Figure** A.1.

The solution is to refactor the `Search` feature, such that the module itself is a composition of the different search criteria. Each search criteria would be implemented inside the module that implements the feature that is being searched, e.g. the Taxonomy search is implemented in the `edition-annotation` module. Then, the search feature implementation would be in the composition module, in the LdoD Archive case, would be in the `edition-ldod` module. Although, the problem goes beyond the scope of this work, that intended, as a first step, to be able to separate the initial domain, which is not the problem here.

## 6.5 Deleters configuration

Even though the way the deleters worked how they were implemented with only two modules, it becomes a problem when the number of modules grow. With two modules the deleter from the module that depended on the other module would remove the objects from the relation and then would call the deleter from the module without dependencies.

This do not work when there are two modules with classes that depend on a class of the same module, i.e. when a module B and a module C depend both on a module A. When this happens, first the deleters of the modules that depend on other should be called and only then, the deleter from the module without dependencies, i.e. the deleters from modules B and C should be called first and only then the deleter from module A. But once the modules B and C are independent from each other, they could not be responsible for calling each others deleters.

For this to work, the deleters need to be refactored, to stop calling other deleters, and the order they are called is specified in the composition module, to assure that all the necessary deleters are called in the correct order.

## 6.6 Composing the imports and exports

Currently the importers and exporters are implemented based on the full set of features from the *LdoD Archive*. After the implementation of the several modules, those features could not be all available in a given composition. On those cases, the importers and exporters should be adapted to only work with the available data.

A way of accomplishing this is to implement a solution similar to the one suggested in **Section 6.4**. The importers and exporters to each feature would be implemented in the module that implement such feature, with a class of the composition module responsible for calling the importers and exporters such that all the data could be correctly processed.

# 7

# Conclusion

The work described in this document was a sample of what is possible to do with Archive LdoD to make it more modularized and consequently more reusable. Altough there is still some work to do, it was proved that the initial idea was feasible and has a defined path to its conclusion.

## 7.1 Conclusions

It was proved that is possible to take something finished, that was intended to be a single context project, and make it reusable for different contexts using the features that were already implemented. In this case, *LdoD Archive* was developed with only one project in mind and could be used to became a more comprehensive tool, decoupled from the specific needs of a project and its presentation, allowing different projects to completely reuse the core of the application and its defined presentation or compose the necessary features and rewrite a different presentation if necessary. This also increases the extensibility of the tool, making it easier to develop new features or even defining different presentations for the same project.

A set of techniques were applied and documented during the resolution of the several setbacks that

can be applied to other places in the implementation that turn out to have equal or similar problems, some of them already identified, but which remain unsolved. From all the identified problems related with application domain, the hardest was already solved, and, as described in **Chapter 6**, the remaining problems are based in the use of deleters.

The separation of the presentation from the application allows a easier reuse of the application, letting anyone to access the API and present it at will. The two main points that were identified as more difficult to generalise were the loading of the documents and the presentation according to the context of the project. By decoupling the presentation from the project it is easier to implement a new presentation that fits the several needs that may appear.

The modules that were implemented and the ones that were proposed have two different perspectives. The base module can be be applied to general projects, i.e. to projects of literary works of any kind, while the remaining modules make more sense to be applied to projects of the same nature as the *LdoD Archive*, i.e. fragmentary works with an order that can vary which can lead to different editions.

## 7.2 System Limitations and Future Work

There is still a long work to do until the application is completely redesigned.

First, there are some classes that should have their name changed to something less related to *LdoD Archive*. Classes like `LdoDDate` and `Fragments` should be changed to `ArchiveDate` and `Texts` respectively. The current names are too tied to *LdoD Archive* context.

Then, the work should focus on dividing the current `edition-ldod` module into smaller modules, such that the finished `edition-ldod` module would only be a composition of other modules and the configuration needed to implement the necessary features. The smaller the resulting modules are the more modularity the application has, because there is a wider number of possible combinations for the compositions. Although, we believe that the main problems were already identified this decomposition is feasible by applying the type of solutions we developed.

Inside this separation of modules by feature, there is a feature that crosscuts several other features. The `Search` feature, when present, should adapt to the modules chosen, because it allows the search of parameters that exist within different modules, e.g. the `Virtual Edition Search` feature only exists when the module that implements the Virtual Edition is present in the composition.

After building all the necessary modules, all the JSPs files should be replaced by REST endpoints that would serve any application, decoupling the presentation layer from the rest of the application. For every JSP that is replaced in the Spring application it is necessary to create the corresponding page in the ReactJS application that fetches the information from the new endpoints. This way the transition between the two technologies is smoother.

The objective of creating a dedicated module to the `Users` feature raises a problem related with the current state of the application, where the TEI documents requires an Admin user to load them. The solution is to automate the load of the documents when the application boots for the first time. This way the module that implements the `Users` feature could be completely decoupled from any other module.

A more ambitious work that could be done would be to rewrite the loaders of the TEI documents such that they accept different TEI specifications. This way the documents would be independent from the tool that is used to interact with them. This work has some problems associated, such as also adapting the view to the TEI specification used by the loaded documents. This feature together with the current implemented features would make *LdoD Archive* the perfect tool for any similar project, but it also takes the risk of becoming too general and lack the accuracy required by the different contexts.

# Bibliography


[1] D. Berry, *Understanding Digital Humanities*. Palgrave Macmillan UK, 2012.

[2] D. Fiormonte, T. Numerico, F. Tomasi, D. Schmidt, and C. Ferguson, *The Digital Humanist: A Critical Inquiry*. Punctum Books, 2015.

[3] M. Gold and L. Klein, *Debates in the Digital Humanities 2016*, ser. Debates in the digital humanities. University of Minnesota Press, 2016.

[4] S. Schreibman, R. Siemens, and J. Unsworth, *A New Companion to Digital Humanities*, ser. Blackwell Companions to Literature and Culture. Wiley, 2016.

[5] P. Svensson and D. Goldberg, *Between Humanities and the Digital*. MIT Press, 2015.

[6] G. Bruseker, L. Kovács, and F. Niccolucci, "Digital humanities," *ERCIM NEWS*, no. 111, 10 2017.

[7] J. Honn. A guide to digital humanities. [Online]. Available: https://web.archive.org/web/20150919224700/http://sites.northwestern.edu/guidetodh/values-methods/

[8] A. Burdick, J. Drucker, P. Lunenfeld, T. Presner, and J. Schnapp, *Digital_Humanities*. MIT Press, 2012.

[9] Text encoding initiative. [Online]. Available: http://www.tei-c.org

[10] F. Pessoa, *Livro do Desassossego*, J. do Prado Coelho, Ed. Ática, 1982.

[11] Fénix framework. [Online]. Available: https://fenix-framework.github.io

[12] Fenixedu™. [Online]. Available: https://http://fenixedu.org

[13] T. E. I. Consortium. Teiwiki: Tools. [Online]. Available: https://wiki.tei-c.org/index.php/Category:Tools

[14] Tapor. McMaster University and University of Alberta. [Online]. Available: http://tapor.ca

[15] M. I. of Technology in the Humanities. Angles. [Online]. Available: http://umd-mith.github.io/angles

[16] B. Bones. Bbedit. [Online]. Available: http://www.barebones.com/products/bbedit

[17] C. W. R. Collaboratory. Cwrc-writer. [Online]. Available: http://www.cwrc.ca/projects/infrastructure-projects/technical-projects/cwrc-writer

[18] M. Holloway. Doctored.js. [Online]. Available: http://holloway.co.nz/doctored

[19] JAPISoft. Editix. [Online]. Available: http://www.editix.com/

[20] C. Ltd. Exchanger xml editor. [Online]. Available: http://www.exchangerxml.com/

[21] FontoXML. Fonto editor. [Online]. Available: https://fontoxml.com/products/fonto-editor

[22] E. Tröger, M. Brush, C. Wendling, F. Lanitz, N. Treleaven, and D. Hopf. Geany. [Online]. Available: https://www.geany.org

[23] Grobid. [Online]. Available: https://github.com/kermitt2/grobid

[24] jedit. [Online]. Available: http://www.jedit.org

[25] D. Ho. Notepad++. [Online]. Available: https://notepad-plus-plus.org

[26] S. S. SRL. oxygen. [Online]. Available: http://www.oxygenxml.com/

[27] sxedit. [Online]. Available: https://github.com/gimsieke/sxedit

[28] S. Rahtz. Tei-emacs. [Online]. Available: https://github.com/sebastianrahtz/tei-emacs

[29] G. Dickie. Tei lite editor. [Online]. Available: https://github.com/jdickie/TEILiteEditor

[30] D. S. A. Schlitz and G. S. Bodine. Teiviewer. [Online]. Available: http://teiviewer.org/

[31] MacroMates. Textmate. [Online]. Available: macromates.com/

[32] K. Mörth. Viennese lexicographic editor. [Online]. Available: https://clarin.oeaw.ac.at/ccv/vle

[33] L.-D. Dubeau. Wed. [Online]. Available: https://github.com/mangalam-research/wed

[34] G. Schmidt and Z. U. Ji. Xml copy editor. [Online]. Available: http://xml-copy-editor.sourceforge.net/

[35] XMLmind. Xmlmind xml editor. [Online]. Available: http://www.xmlmind.com/

[36] Altova. Xmlspy. [Online]. Available: https://www.altova.com/xmlspy-xml-editor

[37] Xeditor. Xeditor. [Online]. Available: http://www.xeditor.com/en

[38] B. Pytlik-Zillig and S. Ramsay. Abbot. [Online]. Available: https://github.com/CDRH/abbot

[39] R. R. C. for History and N. Media. Anthologize. [Online]. Available: http://anthologize.org/

[40] C. Muller. Biblstruct-to-ris. [Online]. Available: http://www.acmuller.net/xml-tei-tut/index.html

[41] T. Erjavec. Docx2tei. [Online]. Available: http://nl.ijs.si/tei/convert

[42] epub-tools. [Online]. Available: https://code.google.com/archive/p/epub-tools

[43] M. L. Jockers. Gutenbergtotei.py. [Online]. Available: http://www.matthewjockers.net/2010/08/26/auto-converting-project-gutenberg-text-to-tei

[44] F. Glorieux. Odette. Œuvres. [Online]. Available: https://github.com/oeuvres/Odette

[45] ——. Teinte. Œuvres. [Online]. Available: https://github.com/oeuvres/Teinte

[46] F. Glorieux and M. Perret. Livrable. Œuvres. [Online]. Available: https://github.com/oeuvres/Livrable

[47] F. Glorieux. Debook. [Online]. Available: https://github.com/oeuvres/Debook

[48] S. Rahtz. Oxgarage. [Online]. Available: https://github.com/TEIC/oxgarage

[49] D. Cavar. Pg2tei. [Online]. Available: https://github.com/dcavar/PG2TEI

[50] J. MacFarlane. Pandoc. [Online]. Available: http://pandoc.org

[51] P. Lopez, R. Loth, and L. Romary. Pub2tei. [Online]. Available: https://github.com/kermitt2/Pub2TEI

[52] B. M. T. LLC. Thutmose ii. [Online]. Available: http://blackmesatech.com/2010/12/thutmose/userdoc.xml

[53] S. Rahtz. Vesta. [Online]. Available: https://github.com/sebastianrahtz/Vesta

[54] B. Desgraupes and S. Loiseau. wiki2tei. [Online]. Available: http://wiki2tei.sourceforge.net/

[55] Xproc e-book processor. [Online]. Available: https://github.com/grtjn/xproc-ebook-conv

[56] T. Elliott. Zotero-rdf-to-tei-xml. [Online]. Available: https://github.com/paregorios/Zotero-RDF-to-TEI-XML

[57] Tei-xsl. TEI-C. [Online]. Available: http://www.tei-c.org/release/doc/tei-xsl/

[58] Transpect. [Online]. Available: http://transpect.le-tex.de

[59] Basex. [Online]. Available: http://basex.org/

[60] Oracle. Berkeley db xml. [Online]. Available: https://www.oracle.com/database/berkeley-db/xml.html

[61] existdb. [Online]. Available: http://exist-db.org

[62] U. of Chicago. Philologic. [Online]. Available: https://www.lib.uchicago.edu/efts/ARTFL/philologic/

[63] M. Hoenick. Refdb. [Online]. Available: http://refdb.sourceforge.net/

[64] M. Corporation. Marklogic server. [Online]. Available: http://www.marklogic.com/

[65] N. Burlingame. Byzantium. University of Oxford. [Online]. Available: http://tei.oucs.ox.ac.uk/Byzantium/

[66] A. Mittelbach and S. Rahtz. Roma. TEI-C. [Online]. Available: http://www.tei-c.org/Roma/

[67] D. of Classical Studies. Classical text editor. University of Salzburg. [Online]. Available: http://cte.oeaw.ac.at/

[68] R. H. Dekker and G. Middell. Collatex. [Online]. Available: https://collatex.net

[69] Context. [Online]. Available: http://wiki.contextgarden.net/TEI_xml

[70] A. Powell. Dc-dot. [Online]. Available: http://www.ukoln.ac.uk/metadata/dcdot

[71] D. L. P. . Services. Dlxs. University of Michigan. [Online]. Available: http://www.dlxs.org

[72] J. Easterly. Data dictionary generator. [Online]. Available: https://github.com/rochester-rcl/data-dictionary

[73] E. Gruber. Etdpub. American Numismatic Society. [Online]. Available: https://github.com/AmericanNumismaticSociety/etdpub

[74] T. Schmidt, K. Wörner, T. Lehmberg, and H. Hedeland. Exmaralda. [Online]. Available: http://exmaralda.org/en/

[75] M. Holmes. Imagemarkuptool. [Online]. Available: http://tapor.uvic.ca/~mholmes/image_markup/

[76] N. U. I. Technology. Morphadorner. [Online]. Available: http://morphadorner.northwestern.edu/morphadorner

[77] M. Burghart. Tei critical apparatus toolbox. [Online]. Available: https://ciham-digital.huma-num.fr

[78] R. Davis. Tei tags. [Online]. Available: https://www.mediawiki.org/wiki/Extension:TEITags

[79] Tile. Maryland Institute for Technologies in the Humanities. [Online]. Available: http://mith.umd.edu/tile/

[80] Weblicht. Universität Tübingen. [Online]. Available: https://weblicht.sfs.uni-tuebingen.de/weblichtwiki/index.php/Main_Page

[81] corebuilder. [Online]. Available: https://github.com/raffazizzi/coreBuilder/wiki

[82] Vercelli book digitale. [Online]. Available: http://vbd.humnet.unipi.it/

[83] R. R. D. Turco, G. Buomprisco, C. D. Pietro, J. Kenny, R. Masotti, and J. Pugliese, "Edition visualization technology: A simple tool to visualize tei-based digital editions," *Journal of the Text Encoding Initiative*, no. 8, 2014.

[84] NINES. Juxta commons. [Online]. Available: http://juxtacommons.org/tech_info

[85] Iiif image api 2.1.1. [Online]. Available: http://iiif.io/api/image/2.1/

[86] The sentences commentary text archive: Laying the foundation for the analysis, use, and reuse of a tradition. [Online]. Available: http://www.digitalhumanities.org/dhq/vol/10/1/000231/000231.html

[87] Mirador. [Online]. Available: http://projectmirador.org/

# Diagrams

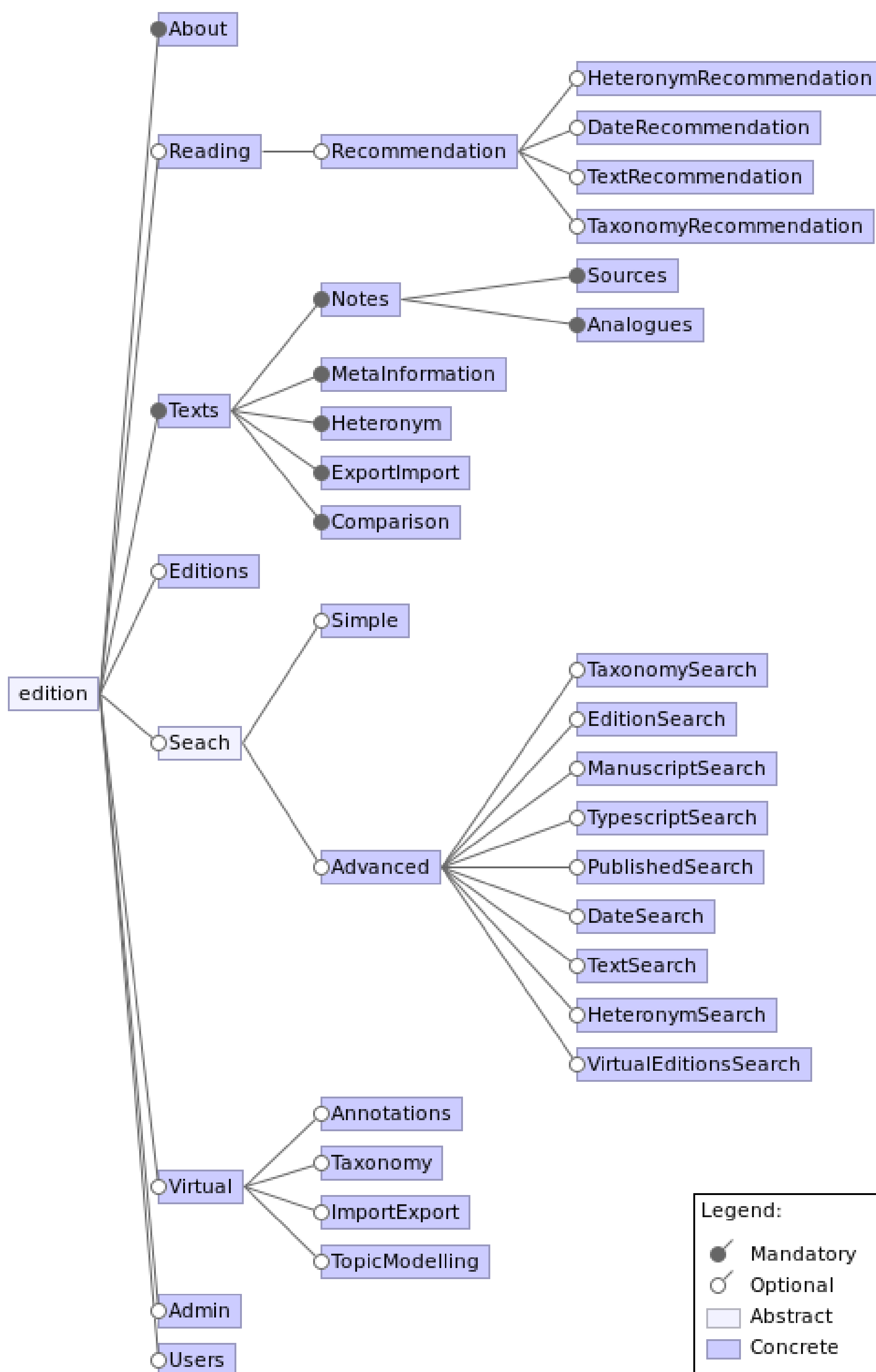


**Figure A.1:** Feature Diagram of *LdoD Archive*

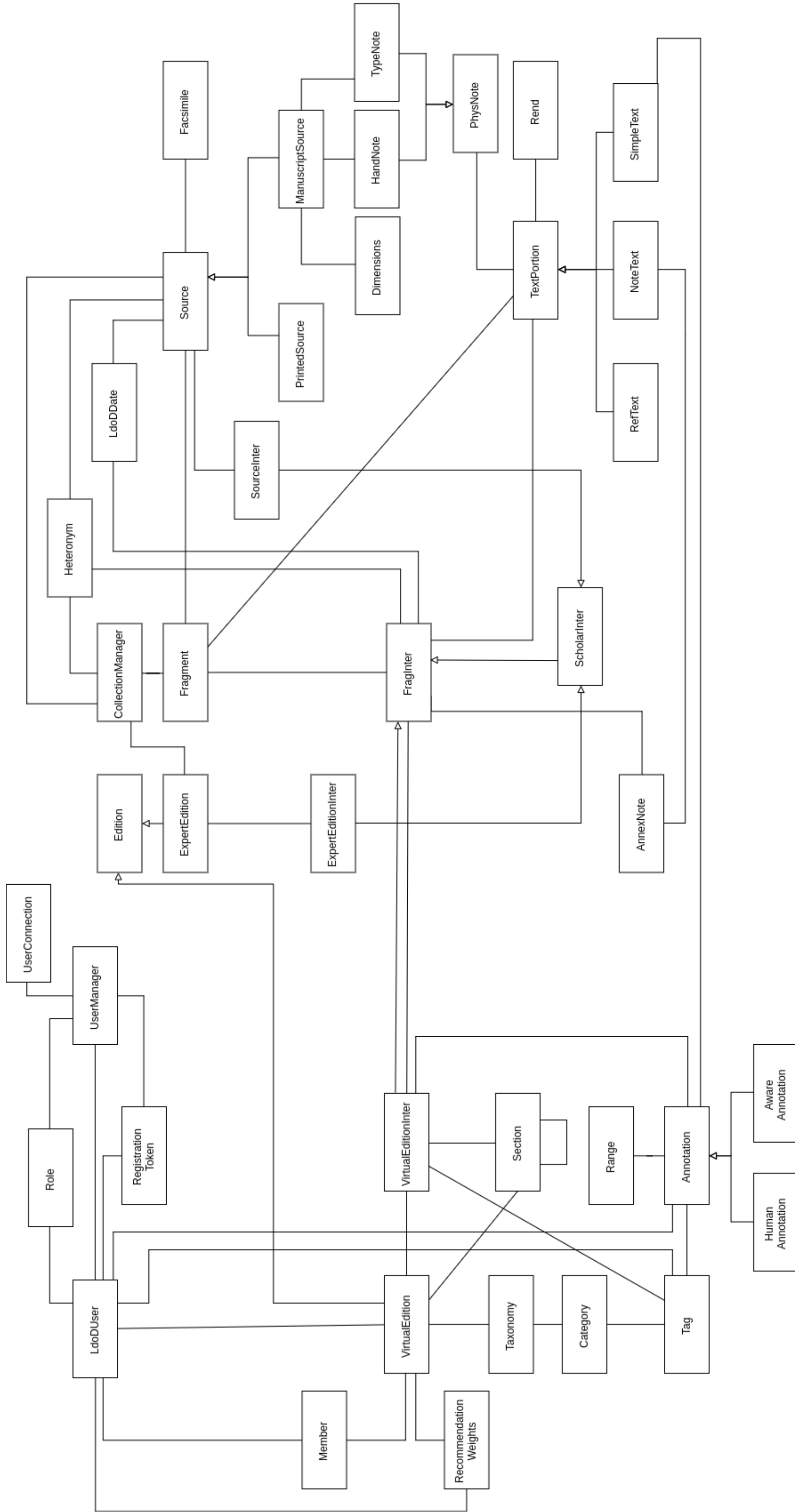


**Figure A.2:** Domain Diagram of *LdoD Archive*

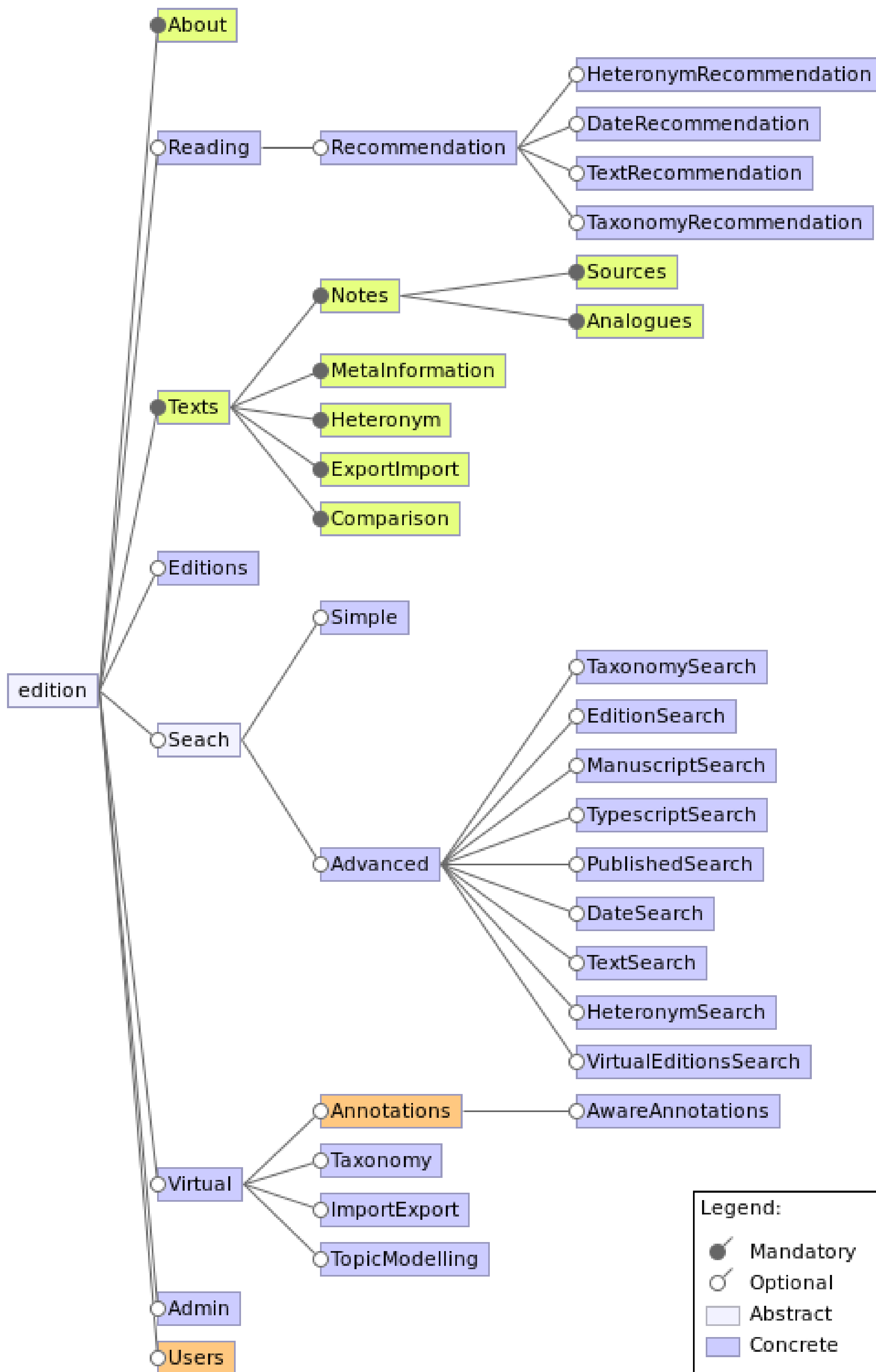


**Figure A.3:** Feature Diagram comparing the three tools

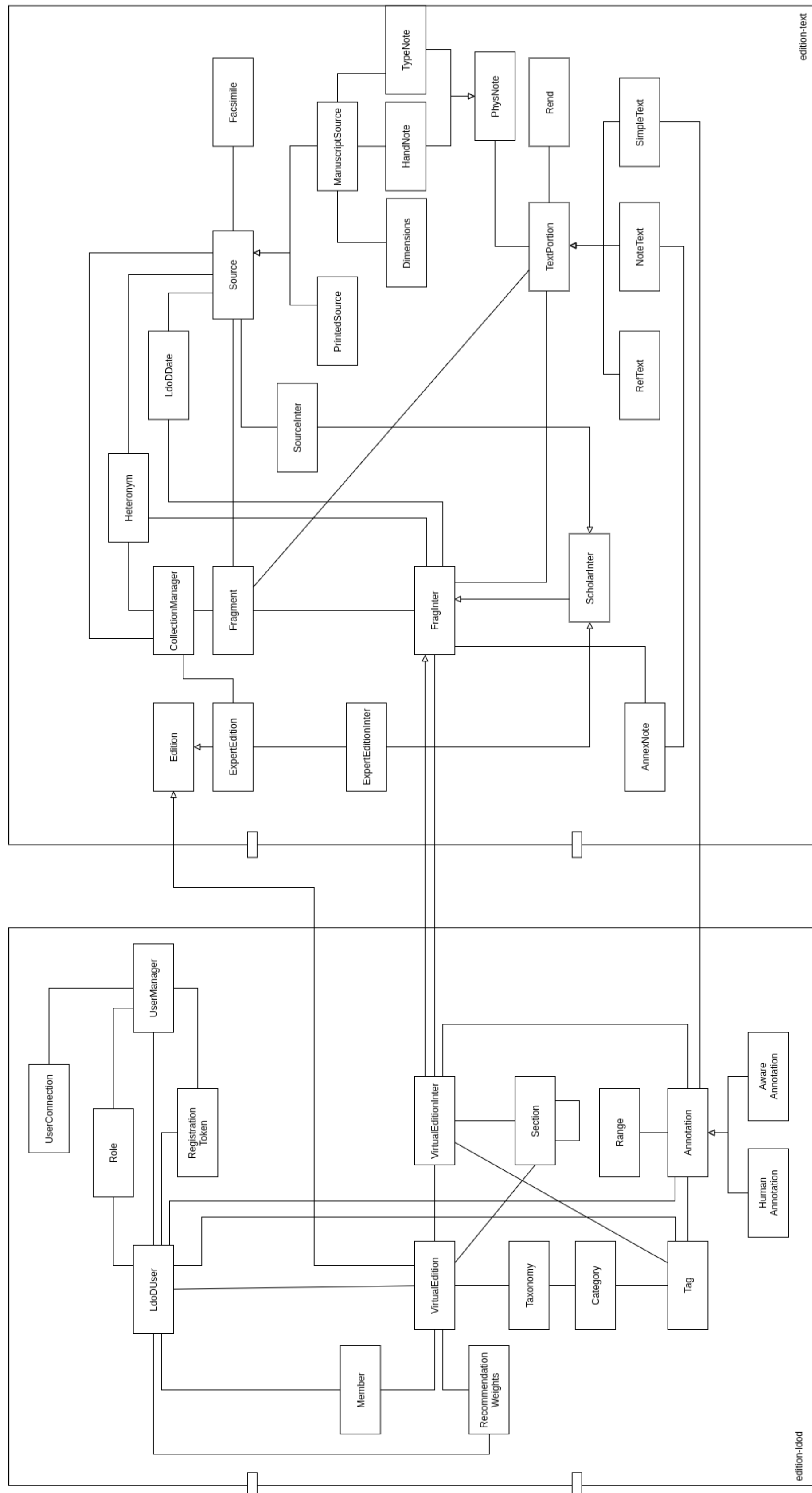


**Figure A.4:** Decomposition of the domain into the two modules

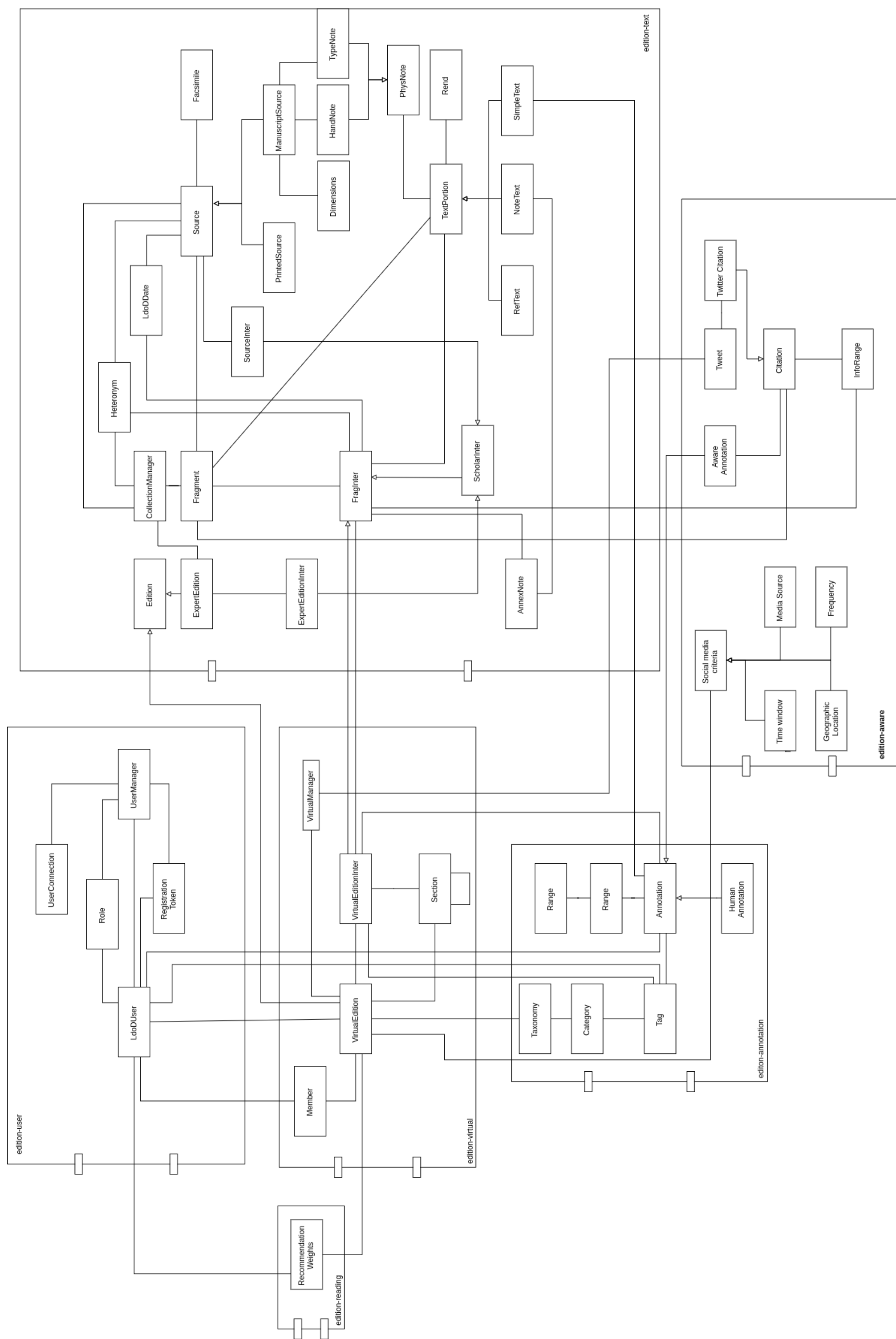


**Figure A.5:** Proposed decomposition of the domain according to the features

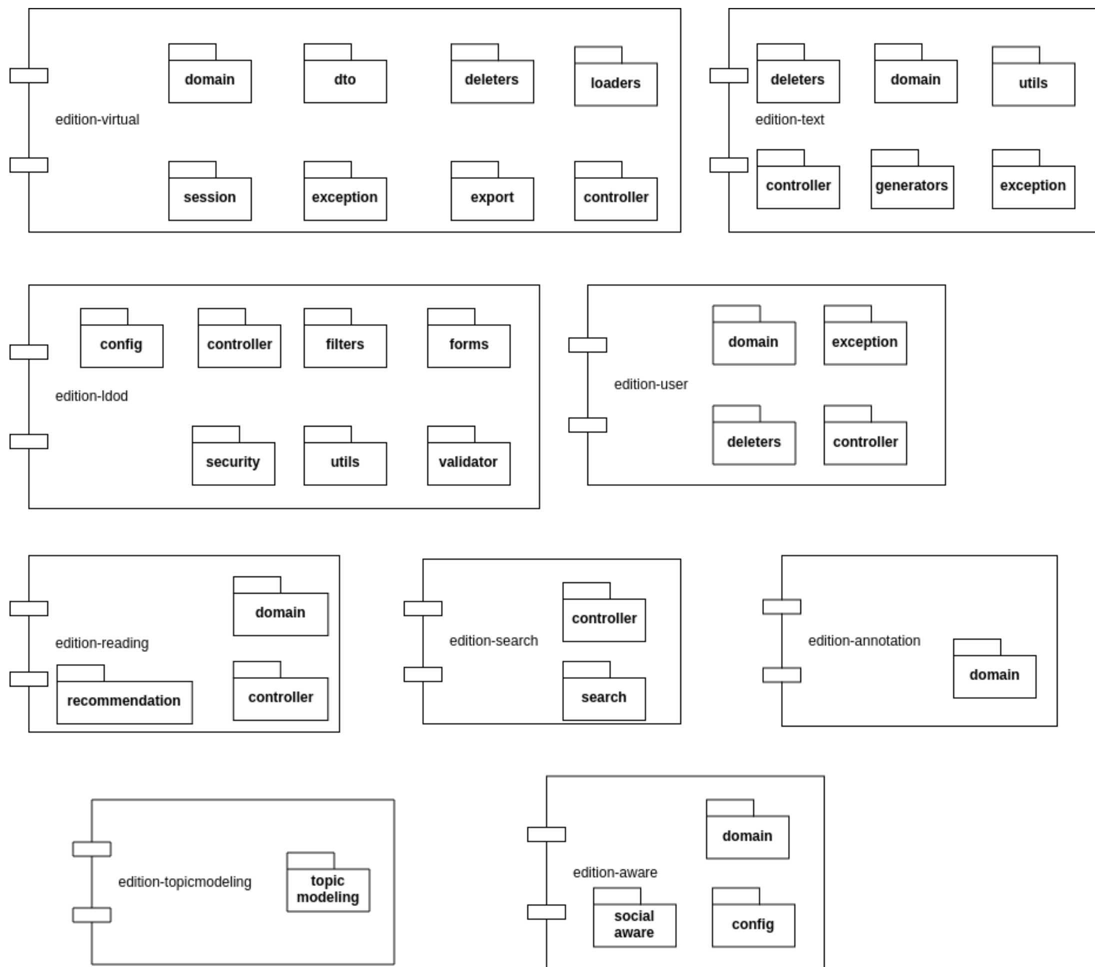


**Figure A.6:** Decomposition of the packages into the nine modules