Reusable Framework for Digital Humanities

A Case Study with the LdoD Archive

Miguel Cruz
Instituto Superior Técnico
Lisbon, Portugal
miguelsantoscruz@tecnico.ulisboa.pt

***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.**

***Index Terms*—Digital Humanities, TEI, Digital Archive, The Book of the Disquiet, Fernando Pessoa, LdoD Archive, Software Architecture**

I. 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 LombardPress. 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 incremental 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.

II. Related Work

A. 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, which definition is always changing 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, 3, 4, 7, 8]

B. LdoD Archive

Fernando Pessoa's *The Book of Disquiet* is an unfinished book written between 1913 and 1935 that was never published.

LdoD 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.

C. Fénix Framework

Fénix Framework [2] 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.

D. Existing Tools

From the universe of all DH tools, our investigation was done only within the ones directly related with the TEI specification. It is possible to separate all the tools related with TEI into twelve categories.

Administrative, Analysis, Conversion and pre-processing, Development environments, Documentation access, Header Creation, Publishing and Delivery, Querying, TEI Encoding, Testing and Quality Assurance (QA).

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.

E. Analysis

The tools that matter for our work, which are Digital Archives that use TEI directly, are two that were considered relevant for the context of our work, EVT and LombardPress.

1) 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* [11], 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 [10] 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. 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) 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 [5] respectively. By using external services to retrieve the information, it is possible to collect data from different sources without any internal change. [9]

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 [9] uses XSLT to generate the information to be stored at the data base, 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 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 [6], 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.

F. Comparison with LdoD Archive

When visualising 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. *LdoD Archive*, EVT and Lombard Press provide information about the text under study, i.e. meta-information. A feature that is only available for EVT is the linking relation between the manuscript and the transcription.

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 two tools allow annotations in different styles, *LdoD Archive* allows versatile annotation, from a single word to annotation of phrases and sentences, while Lombard Press allows annotation at the paragraph level.

A feature that differentiates *LdoD Archive* from the other tools 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.

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.

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.

III. Initial *LdoD Archive* Architecture

LdoD Archive is a web application that is accessed through a browser.

A. Features

LdoD Archive has a set of features that allow the Archive to be one of the most complete Digital Archives available online. Beyond the simple text presentation, it also allows to compare different versions of the same text or annotate those texts. It also allows to create our own editions, with texts chosen and ordered at our will. Theses editions can have the texts classified. It is possible to read the book in an order recommended by the system, accordingly to a set of restrictions applied by the user. It is also possible to search by a simple set of keywords, or to search accordingly to an extensive set of properties, such as the date or the heteronym of the texts.

B. Packages

The packages that compose the *LdoD Archive* are separated by functionality, apart from the domain, that contains all the domain classes necessary for the well functioning of the application. The class of the domain have dependencies among them that do not allow to perform a simple refactor by moving the classes around, while the remaining packages have no dependencies among them and are easily refactored.

C. 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.

When implementing a domain that has all the classes accessible there were some dependencies that were created that cause problems when they need to be moved apart. Apart from the dependencies between classes there were also a class called `LdoD` that had code from different features, from the users to the virtual editions.

IV. The new architecture

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.

A. Features Decomposition

EVT and Lombard Press features are a subset of the *LdoD Archive* features, so it was possible to decompose *LdoD Archive* into the other two tools in terms of features.

There is a feature that is the main focus of all three tools, the presence of texts that can be read by the users, alongside with the information about the texts and its meta information. 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.

While EVT is repoduced by only that set of features, LombardPress also supports the existence of users, and allows them to annotate the texts at the paragraph level.

B. Packages Decomposition

The packages were separated to the modules that implement the feature that they support.

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.

C. Domain Decomposition

Finally, the domain was separated accordingly to the classes that were necessary to implement the features related with the `Texts` 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. There are some dependencies between the two modules that will be discussed later on this section.

There are relations that before the changes existed within the single domain, but after moving the classes with dependencies out of the module, it causes a problem because of the Fenix-Framework. Even though Fenix-Framework allows to establish relations between classes of different modules, there is a problem associated with such change. The module that is being used in the relation specified in another module is not changed when that relation is implemented. This means that the class used in the relation does not have any method to access the relation. When the code in the module that specifies the relation between the classes is compiled, a new class base for the class outside the module is generated. This means that only the classes from within the code that specifies the relation can call the methods generated for the relation. The class from outside the module is exactly the same, therefore, it can not call any method associated with the relation.

The cases that needed to be solve in *LdoD Archive* were the inheritance from a class outside the module and the "uses" relation of classes outside the module.

There was the relation between four classes, `FragInter`, `ExpertEditionInter`, `SourceInter` and `VirtualEditionInter` that had the two cases mentioned at the same time.

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`.

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 **??**. The following code shows the implementation of the `getLastUsed` method in the `VirtualEditonInter` class.

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 super-class 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` 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 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, every FragInter related with a VirtualEditionInter is a VirtualEditionInter.

A particular situation where the relations between different modules was a problem was deleting objects.

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.

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.

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.

D. 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.

There are three modules that do not have any domain classes associated, which is the case of edition-ldod, edition-search and edition-topicmodelling. Apart from edition-ldod all the other eight modules implement a feature or a small set of feature.

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.

V. Presentation of *LdoD Archive*

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.

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.

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 way, at first, the only ReactJS elements in the application are the common ones. The pages with the content are still provided by the Spring application using JSPs. The elements that are now implemented in ReactJS were removed from the JSPs, where the content of the JSPs are now only the page with its content.

The creation of the ReactJS elements are done dynamically which means that it can receive the data from a REST call and create the menu accordingly to the response. This allows the user interface to be prepared to changes in the modules composition, allowing to hide unavailable features from the user.

Each change of page triggers an HTTP request, that fetches the corresponding page from the Spring application that comes in raw HTML format, that is then parsed using a dedicated library, `react-html-parser`, which was tuned to generate the correct ReactJS elements according to this project specificities.

This new user interface also supports the three languages that were already in the JSP version. The application has a json file in the resources folder that contains the translation of the messages to the three languages, which allows to have only one interface implemented that works in the three languages. The request done to the Spring application always carry a query text that guaranties that the page retrieved is in the correct language. In the ReactJS context, the translations are assured by a custom class that extends the class `IntlProvider` from the `react-intl` library.

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 has 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, but being the long term objective to implement the whole interface with this technology, the pages that cause problem are the perfect candidates to begin that transition.

VI. 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.

We intend to validate our results by identifying the dependencies and analyse whether the techniques developed in this work will also apply to these new cases.

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. 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.

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.

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 simple to solve 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.

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 available criteria.

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.

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) 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 VI-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.

VII. 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.

A. 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 VI**, 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.

B. 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.

References

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

[2] *Fénix Framework*. Dec. 20, 2017. URL: https://fenix-framework.github.io.

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

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

[5] *IIIF Image API 2.1.1*. Dec. 17, 2017. URL: http://iiif.io/api/image/2.1/.

[6] *Mirador*. Dec. 18, 2017. URL: http://projectmirador.org/.

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

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

[9] *The Sentences Commentary Text Archive: Laying the Foundation for the Analysis, Use, and Reuse of a Tradition*. Nov. 28, 2017. URL: http://www.digitalhumanities.org/dhq/vol/10/1/000231/000231.html.

[10] Roberto Rosselli Del Turco, Giancarlo Buomprisco, Chiara Di Pietro, Julia Kenny, Raffaele Masotti, and Jacopo Pugliese. “Edition visualization technology: A simple tool to visualize tei-based digital editions”. In: *Journal of the Text Encoding Initiative* 8 (2014).

[11] *Vercelli Book Digitale*. Dec. 17, 2017. URL: http://vbd.humnet.unipi.it/.