Manuzio: An Object Language for Annotated Text Collections

Marek Maurizio and Renzo Orsini

Dipartimento di Informatica, Università “Ca’ Foscari” di Venezia,
{marek,orsini}@dsi.unive.it

Abstract. Traditionally, text collections are represented as text files with some kind of markup to define extra-textual information, like metadata, annotations, etc. We propose an approach which uses the natural structure of a literary text to build specialized objects abstractions on text collections, objects which can be used to make non-hierarchically nested, multi-level annotations, to create complex metadata, and to perform complex queries and analysis on the collection. The language Manuzio is the result of this approach, and in this paper we introduce its main features, as well as the sketch of a system, based on the language, to manage persistent text collections and write complex applications over them.

1 Introduction

The traditional way of representing texts for automatic processing is through some kind of enrichment of the text with other, distinguished text, carrying some information, like annotations, metadata, formatting instructions, variant texts, etc. This approach, in which the distinguished text is called the markup, has been widely diffused also by the availability of standard markup languages, like SGML, and in the recent years, XML, which made possible the definition of families of standards specific for literary texts, like the TEI ones [1].

The great advantages of a marked text is that it can be read and written with relative ease from a human being, as well as efficiently processed by a computer program. Moreover, when the marking of the text follows some widely accepted standard, it can be exchanged among different systems, processed by different applications, and, in general, used in a robust, interoperable way.

These advantages are, however, balanced by several, noteworthy, shortcomings, due primarily to the fact that marking, as any other text, is linear, and can be applied only to contiguous segments of text. So, for instance, one can mark a text only with hierarchical structures and can annotate only those structures, without the possibility of exploiting other types of structures in a text, or marking non-contiguous text, or at different, independent, levels, or making multiple marking of same type on partially overlapping text, etc. Solutions exists to overcome this shortcomings, as described in the next sections, but they are often cumbersome, produce complex unreadable text and notably increase the complexity of programs dealing with such texts. The markup approach, moreover,
makes difficult to write systems which treat text as other data types, for instance to make complex applications which modify it, perhaps by adding programmatic annotations as result of text analysis, or to allow concurrent access by different users which make annotations, etc.

Another important disadvantage of the markup approach is, in our opinion, the fact that programs for processing marked text are not easily written, requiring the mastering of complex query languages, not specialized for the particular domain, like those typical of XML (for instance, XPath, XQuery, XSLT). In particular, we feel that such languages are not easily grasped by scholars and researchers in the humanities, which, on the contrary, should have the possibility of at least understanding, if not writing, queries or simple programs over such kind of data.

For the above reasons, in this paper we propose a radically departure from the traditional markup approach, departure based on representation of text with abstract structures in some way similar to those present in object oriented data models, typical in many software engineering areas. Our approach uses the natural structure present in literary texts to build specialized software objects, called textual objects, which can be used to view the text with as many structures as necessary, even not hierarchical ones; to make annotations at different levels, with a great flexibility, overcoming the limitations of textual marking; to add metadata to text; and, finally, but equally important, to define a language to make text queries and analysis in a simple and powerful way.

The Manuzio language, introduced in this paper, is the result of such approach. Its aim is to be the base language of an integrated working environment for representing text collections, performing multi-level, flexible, annotations from concurrent users, querying such annotated text collection with a simple query language, as well as allowing the writing of even complex textual applications with powerful operators.

Proposals based on text structure, instead that marking, have been already presented in the literature, like, for instance, NITE [2], MdF [3] and TOMS [4], but, as will be shown in the next section, they have some limitations with respect to Manuzio in their scope, abstraction mechanisms, annotation possibilities.

Moreover, in our opinion, the benefits of the markup approach can still be provided with our proposal in two ways: by providing simple and robust mechanisms to export and import text through markup languages, like XML, and by providing simple and flexible tools and editors to visualize and work with textual objects, which is one of the objectives of our research.

The Manuzio language is still a work in progress: however, we feel that it is important to publish written documentation about it to receive feedback by researchers in this area and potential users, so that we can better tailor it to the users needs. For experimental purposes, we are currently applying it in a research carried at the Venice University about latin poetry.

The rest of the paper is organised as follow: in section 2 a brief introduction to the problems addressed by the approach is given, referring to related works in literature, and discussing the originality of our proposal. Next, in section 3 the
Manuzio main features are introduced. We continue in sections 4 and 5 giving a complete description of the basic characteristics of the language. In section 7 an overview of the query constructs is given. A sketch of an architecture for a system is presented in section 8 and, finally, in section 9 conclusions are given and future work is addressed.

2 Related Works

2.1 Markup Languages

In recent years the XML markup language have been confirmed as the de facto standard for the encoding and interchange of literary texts, along with other forms of electronic textual data. One of the most important initiatives in this field is represented by the Text Encoding Initiative (TEI) [1]: a consortium of institutions which maintains and develop a standard for the representation of texts in digital form. TEI uses XML as its encoding language, giving the end users the possibility to use a vast amount of tools to store and handle textual data, keeping in the same time the model simple. However, in XML it’s difficult to encode multiple hierarchies of objects that does not nest neatly in each other. This problem is often referenced as “overlapping” and is discussed by a large amount of papers in literature, like [5, 6]. One of the simplest examples is the bible book/chapter/verse and book/story/paragraph diverging hierarchies. The basic ways of dealing with this problem is described in the TEI guideline itself. Simply adopting a language with concurrent markup capabilities was proposed in early versions of the guidelines with the SGML concur feature. This solution is not viable in XML since it does not support such a construct as other meta-languages like LMNL [7]. Other solutions propose to fragment the overlapping elements so that the resulting fragments does nests neatly, to use empty start and stop elements or to simply adopt a redundant memorization of non nesting data. All those solutions are workarounds that needs specific applications or transformations to be processed and privileges one hierarchy over the others. A full description of those techniques is beyond the scope of this paper, but see for instance the survey in [8]. Another approach has been discussed in [9–11] and has been incorporated in the TEI guidelines. This solution, called “standoff XML”, distributes the marking on several external files so that each one contains a different hierarchy. External marking uses a link mechanism, like Xinclude and Xpointer XML features¹, to identify an unique portion of a base text, usually kept read-only and with no or basic markup. External markings are completely independent, so each one can be modified, added or deleted without affecting the others. One of the first proposals for stand-off markup of linguistic corpora can be found in [12]. This approach has been used in several experiments [13, 14] and has been proven robust and functional, although it can be at times a cumbersome approach and produces complex XML documents which are difficult to read and require ad-hoc complex software tools. On the other hand, only

¹ See http://www.w3.org/TR/xinclude/ and http://www.w3.org/TR/xptr/.
few specialized applications are available on the TEI or other XML encoding for dealing with specific tasks of literary texts processing.

2.2 Object Models

More complex solutions are based on the ideas first presented in [15], where text is seen as one or more hierarchies of textual objects. The article propose a view of text as a hierarchy of objects, in a spirit similar to our proposal. This idea is further refined in [16], where the concept of multiple hierarchies is discussed more in depth.

An example is the NITE object model [2], where a mathematical formalisation for the representation of richly annotated multi-modal language data is presented. The model is represented by a graph where the nodes, called elements, are strings, have additional attribute-value pairs, and have special constrains. Because of these constraints, the graph decomposes into a collection of intersecting tree-like structures, reflecting textual hierarchies. While not based on any particular implementation, its tree-based approach brought, among other reasons, to the adoption of XML as implementation language. The NITE object model is designed to take timing informations into account directly, making it a suitable model for spoken text and other types of textual data tied to a precise timeline.

The MdF model [3] is based on two entities: monads and objects. A monad represent an indivisible portion of text; each monad is identified by a progressive number. Objects are composed of monads, and organized in sets of object types. The model has a minimal object type in which each object is composed of exactly one monad, usually representing a word. Each object type defines a set of functions, called features, each one taking an object and returning a value of arbitrary type that can be used to add metadata and annotations to the text. The hierarchy formed in this way, is actually a one-level hierarchy, since even complex objects are composed of monads, and not by other objects. For this reason, the MdF model can represent objects with gaps, or overlapping objects of the same type with ease, giving great flexibility to the model at the cost of not allowing a multilevel hierarchy. Complex operators exists to use the relations formed by overlapping objects.

The Textual Object Management System (TOMS) [4], on the other hand, is a similar system where objects are instead composed of other objects. TOMS defines sets of types, called textual object types, and organize these types in one or more hierarchies. A formal text called grammar uses four structural constructs (repetition, choice, sequence and parallelism) to formally define those hierarchies. Each object type T is tied to a recognizer function that, given a textual input, returns the boundaries of the objects of type T present in the text. These functions can be defined with a programming language or simply with regular expressions or manual enumeration. When both grammar and recognizer functions are defined, TOMS is ready to process an input text and create an index of textual objects. TOMS, like NITE and MdF, define a query language that can be used to retrieve and analize stored textual data using objects perspective.
2.3 Comparison with Manuzio

Differently from the pure mark-up proposals, like TEI and stand-off annotations, Manuzio is aimed to provide a theoretical model not tied to a single implementation. This model is designed to cope well with textual data coming from an humanistic context, but will prove useful for other kind of data too\(^2\). Like MdF and TOMS Manuzio defines a hierarchy of textual objects but, unlike the first, we aim to the construction of a strict multilevel hierarchy, instead of a single level, open one. This multi-hierarchy, composed of one or more eventually overlapping hierarchies, is defined with a modeling language that is in some senses similar to TOMS grammars to which Manuzio add a cleaner presentation and a powerful inheritance mechanism, more in the spirit of classical object-oriented languages. One of the main features of Manuzio, mostly missing in other solutions, will be the definition of annotations of complex type; annotations can also be seen as textual objects, so that they can prove useful in solving problems like filologic analysis, text criticism, and so on.

3 Manuzio Overview

Manuzio is a language to define schemas of complex, annotated, textual corpora and to query and operate on them. The language data model is based on abstraction mechanisms which have some similarities from object data models and languages commons in many software engineering areas, but are peculiar to the particular domain involved.

We consider textual information in a twofold way: as text as well as structured object. The textual aspect derives from the fact that each object refers to a portion of the full text, as a sequence, for instance, of Unicode characters, “normalized” in a way that we will discuss later. We will call such text the underlying text of an object. The structural aspect derives, on the other hand, from the abstraction mechanisms which will be used to describe the different relations among textual objects, like aggregation and specialization: for instance, a poem is composed by a title and a set of stanzas, which are composed by verses, a sonnet is a particular kind of poem, etc.

The main features of the language are:

1. The language has been designed for dealing with text corpora, with the objective of being the basic language for literary text management systems. For this reason, the language semantics is based on a single textual object, which is the full text corpora, and which is the text underlying the top-level textual object. All other objects are component of this. In the rest of the paper this text is called the full text, and in the language it is denoted by the predefined identifier text.

\(^2\) see section 9
2. The language is typed. Each object has one (or more) types which describes its structure. Types are declared in the textual schema, which defines all and the only types of objects allowed over the full text. An object type can be considered an interface which describes the ways in which the textual objects of that type can be operated upon. The type of text is Text.

3. A textual object is uniquely determined (and identified) both by its minimal type, and the portion of the full text which underlies it, and which can comprise non contiguous spans of text. This way of defining the identity of an object means that two objects are equal if they have the same type and their underlying text is equal. This means, for instance, that the objects corresponding to the same word in different parts of the text are different.

4. A textual schema defines the set of textual object types for a full text managed by the system. The types are defined using the following abstraction mechanisms:

   **Aggregation:** a textual object can be composed by other, potentially overlapping, textual objects contained in it, called its *components*. For instance, a word can be composed by characters, a poem by lines and, at the same time, by sentences, etc.

   **Repetition:** (or aggregation of homogenous objects) an ordered list of textual objects, the *elements* of the list, which can be seen as a single textual object.

   **Specialization:** a textual object can have a type which inherits all the characteristics of another textual object type, as well as adding new ones. For instance, a scientific article is a particular type of article, a septenary is a verse with seven syllables, etc.

   **Attribution:** a textual object can have associated particular values, called *attributes*. This mechanism allows annotations upon objects, as well as other information which are not part of the text. For instance, a poem can have an author, a poem verse can be annotated to explain its meaning, a word can be annotated with its “part of speech” information, etc.

5. The schema for a specific text management system is populated from the full text through a text parsing component which will is not discussed in this paper, and whose properties are summarized in section 8.

   The schema in figure 1 illustrates an example concerning simple poems and novels.

   A **SimpleWork** is composed by a **Title**, and **Sentences**, and has an attribute **Identification** which provide information about its author and publication year.

---

3 An object can have more than one type because of the specialization hierarchy. The *minimal* type is defined as the most specialized one, and sometimes it is loosely referred as “the type” of the object.

4 This is in contrast with traditional object-oriented data models, where any object has an identity which is a fresh value created when the object is instantiated, and two objects are different if they have a different identity, even if they are structurally equal.
In the aggregations the plural form of the name means a repetition, so that, for instance, Sentences means ‘a repetition of Sentence’. There are two specialized type of works: Novels and Poems. A Novel inherits Title and Sentences from Work, and has no other components. A Poem, on the other hand, in addition to inheriting from Works Title and Sentences, has another repeated component, Stanzas. Note that, in this way, a Poem has two components Sentences and Stanzas, which are independent ways of considering a poem. The basic, undecomposable type (called unit) is Character, which is predefined as an unicode character. The other two types, Word and Punctuation, inherit from Token. Every other type inherits implicitly from the top level object type Text. Other abbreviations are used to simplify the example and will be discussed in the next sections, where the language features are presented and motivated.

4 Objects and Types

4.1 Textual Objects

A textual object is characterized both by the underlying text and the basic “component-of” relations it has with other objects of the system, as well as by its attributes values. A basic property must hold for all the objects: a component C of an object T must have an underlying text which is strictly contained in the text underlayed by T. We will call this the “containment property” of the textual objects. Another constraint, which could have been imposed, is that the
text underlying all the components should coincide with the text of the composite object. However, we chose to exclude this constraint to make the language more flexible.

Each textual object has a type, which specifies the relations in which it is involved, and, implicitly, the operations that can be applied to it. Types are defined in the schema: they essentially describes all the possible ways in which the full text can be decomposed. As it will be clearer in the following, objects can have more than one type, because of the subtyping hierarchy: for instance, in the previous example, a textual object which correspond to a poem has both the more specific type Poem, as well as the more general type Work.

Another important point to stress is the fact that the textual objects of the system are manipulated by the language, but they do not need to be explicitly created: a part from the object corresponding to the full text, identified by the reserved name text, they arise as result of the language operators, and persist until they can be referenced by other operations. From the language semantics point of view, the user could simply think that all the objects composing the full text and those built from them are always available, can be operated upon, annotated, and so on. The implementation, on the other hand, will take care of optimizing the system memory requirements.

In the next paragraph the basic syntax for defining object types is described.

4.2 Basic object type definition

A textual object is usually defined by aggregation within a basic object type definition. In such a definition all the object components are listed. Their names must be different, and the order is not significant. In this paragraph we concentrate only on the basic features on this definition, while attributes and inheritance will be discussed in the following sections.

The simplified syntax of this mechanism is the following:

```
BasicDefintion = BasicTypeDefinition | AttributeDefinition

BasicTypeDefinition =
    def TypeIdentifier [ (pl. TypeIdentifier) ] [is TypeIdentifier]
    has
    {, Component}
    attributes
    {, AttributeIdentifier}
    end

Component = [Identifier :] TypeIdentifier

AttributeDefinition =
    attribute AttributeIdentifier
    Name : DataType
    { Name : DataType }
```

5 It has also the type Text, as every other object.

6 In the syntax, square brackets represent optionality, while curly braces represent repetition.
For instance:

```plaintext
def Poem (pl. Poems) is Work
has
  title: Sentence
  sentences : Sentences
  stanzas : Stanzas
attribute
  AuthorInfo
end

attribute AuthorInfo on SimpleWork
  Author: string
  Year: int
end
```

Data types includes elementary types, like `int` or `string`, as well as more complex types. They will be discussed together with attributes in section 6.

When the plural name is made by simply suffixing the name with a “-s”, which is a very common in english, it can be omitted in the def clause for the sake of readability. An important consequence of the singular and plural form of definition is that, in effect, we are defining at the same type two types: for instance, Poem and Poems are both types of textual objects. The first is the type of the objects corresponding to a single poem, while the second one is the type of textual objects formed by a repetition of poems.

For what concerns the components, we must specify their name and type, with the keyword `rep` indicating that the component value is a repetition of values of the type. As abbreviation, if there is no ambiguity, components can be called with the same name of the type (with the plural form of the name meaning that the type is a sequence), and also in this case the type can be omitted.

Moreover, since Text is a supertype of all the types, each type which is not defined as subtype of another is assumed subtype of Text.

**Component Selection**

Objects defined by aggregation have the fundamental selection operation which specifies the component name to be selected. For instance, if W is a work object:

```plaintext
Title of W
```
returns the title of the work as a textual object of type `Sentence` that can be further operated upon.

When a component is a repetition, it can be selected as a single textual object with the `of` operator, like in (`P` is a poem object):

```plaintext
Stanzas of P
```
which returns an object of type `Stanzas` composed by the stanzas of `P`.

Since the result of a component selection is a textual object, this operation can be repeated on it, like in:

```plaintext
Words of Title of W
Lines of Stanzas of P
```
In the first case, the result is a repeated textual object, of type \texttt{Words}, containing all the words of the title of \textit{W}. In the second one, the result is again a repeated object, of type \texttt{Lines}, containing all the lines of the stanzas of P (hence, all the lines of P). Other operators can be applied to textual objects and will be discussed in the sequel:

- Repeated textual objects have operators for building textual objects from a selection of their elements either through indexes or query conditions. These operators will be discussed in section 7 on the query language.
- Because of the specialization hierarchy, one can apply selection components also inherited from supertypes, for instance to select sentences from a poem, since a poem is also a work.
- Components selectable from an object include also those defined in its inner components, and this is due to fact that, as we will see in the next section, there is another important relation among types, the component relation. For instance, one could select directly the lines of a poem, or its words. Note that this is another important point in which this language differ from the traditional object-oriented ones.

5 Type Relations

5.1 Specialization Relation

The classical object-oriented specialization abstraction mechanism allows the definition of a type as subtype of another one, the supertype, so that the former can inherit the operators of the latter (the relation is a partial order, since it is reflexive, antisymmetric and transitive). In our language, the type \texttt{Text} is supertype of all the other types, and a type can be defined as subtype of another with the \texttt{is SuperType} syntax.

In the example in section 3, a \texttt{Poem} is defined as subtype of \texttt{Work}. The implication of this is that an object of type \texttt{Poem} is also of type \texttt{Work}, and everything can be done to a object of the latter type (including the selection of its component), can be done also to an object of the former one, as shown above.

The specialization hierarchy in Manuzio is a simple inheritance hierarchy (a type can have only a direct supertype). Since an object can have a hierarchy of type, its \textit{minimal} type is defined as the most specialized type. For instance, a poem object has types \texttt{Text}, \texttt{Work}, \texttt{Poem}, and its minimal type is \texttt{Poem}.

Because of the inheritance, the components which can be selected from an object are also those defined in its supertype, so that, for instance, since a \texttt{Poem} is also a \texttt{Work}, the following expressions are correct:

\footnote{Note that this object is different from the \texttt{Title} itself, which is of type \texttt{Sentence}}
\footnote{The result, in this case, is a \textit{single} repeated object. The reader with experience with other traditional programming languages can notice that in such languages this operation should return a list of repeatable objects. In other words, in this case there is an automatic flattening of the result.}
5.2 Component Relation

The specialization among types is not the only type relation in Manuzio: since the main mechanism to define a type is through aggregation\(^9\), we define also a component relation:

**Definition 1.** A type \(T_1\) is directly component of type \(T_2\) if \(T_1\) or seq \(T_1\) is the type of a component of \(T_2\). \(T_2\) is called an enclosing object type of \(T_1\).

The composition relation is a strict partial order (the relation is irreflexive, antisymmetric and transitive). Since the relation is transitive, we can say that, for instance, Line, is which is a direct component of Stanza, is also a component of Poem.

The implication of this relation is that we can select for an object of type \(T_1\) also the components of \(T_2\), if \(T_2\) is component of \(T_1\). For instance:

```
Lines of P
```

is correct, since lines are components of stanzas. The result is a repeated textual object of type Lines which consists of all the lines of the poem P.

The following situation can arise: an object of type \(T_3\) is component both of \(T_2\) and \(T_2'\), which are two components of \(T_1\). For instance, a word is component both of sentences and lines, which are components of poem. In this case, one could think that there would be an ambiguity when selecting the words of a poem. In reality, there is no ambiguity, since the selection operator returns a textual object, which, because of the containing property, is composed by all the different words contained in the poem. In this case, in poem, all the words of stanzas coincide with all the words of sentences. In general, the component selection operators performs the union of all the components of the object.

Another important consequence of the component relation, that will be shown in section 7 on the query language, is that, in addition to the operators which retrieve objects by specifying conditions on their attributes, we could use also the attributes of the objects in which they are components. So, for instance, we could say that the Line “Shall I compare thee to a summer’s day?” has Author Shakespeare if this Author value is an attribute of the entire poem.

6 Attributes

The attribute mechanism is the way of associate to a textual object a value which can be as simple as a string or complex as a set of other textual objects. The language allows in a simple way the creation of these values and their association

---

\(^9\) The repetition is considered a special kind of aggregation.
to textual objects, as well as their removal or modification. Note that a single textual objects can have many attributes, while an attribute can be associated only to a single object.

An attribute value is similar to a record of other languages, in that it is a set of couples field, value. Since Manuzio is a typed language, the type of an attribute must be defined, as well as the type of the textual objects to which they can be associated. The set of data types of the values which can be used in attribute types include:

- simple atomic values: numbers (values of types number), strings (of type string), boolean values (of type bool);
- textual objects;
- records and sequences, whose types are defined through the type constructors record[name: type; ...] and seq type.

Let us now show some examples of attribute type definitions.

```plaintext
attribute Grammar on Words
    part_of_speech: string
end

attribute MeterInfo on Poem, Lines
    meter: string
end

attribute AuthorInfo on Poem
    author: [name: string; surname: string; birthYear: integer]
    year: integer
end

attribute Comment on Text
    contributor: string
    comment_text: string
end

attribute SimilarLines on Line
    other_lines: Lines
end
```

In the first example the attribute about the part of speech of a single word or multiple words of a sentence is defined. The MeterInfo attribute could define, through some standard code, the meter of an entire Poem or of a set of Lines. AuthorInfo simply defines the author information about a Poem. Comment, on the other hand, is an attribute to associate whenever comment is desired to a textual object (note the use of type Text, which means that any type of textual object can be the target of an association (for instance, a non contiguous set of lines of a poems). Finally, the last example shows how to use textual objects inside attributes, where a Line is annotated with a set of other Lines, which are considered similar according to some suitable definition.

In the envisioned multi-user system, attributes are referred to the users that create them.
Attribute operators

Once defined an attribute type, its instances can be created and associated to textual objects with the `new` operator. The `modify` and `remove` operators can be used to modify or remove them. The syntax is the following:

```plaintext
AttributeCreation =
    new AttributeType on TextualObjectExpression
    field = Expression
    {field = Expression}
end

AttributeModification =
    modify AttributeExpression on TextualObjectExpression
    field = Expression
    {field = Expression}
end

AttributeRemove =
    remove AttributeExpression from TextualObjectExpression
```

where `TextualObjectExpression` is an expression which returns a textual object, `Expression` returns a generic value, while `AttributeExpression` returns one or more attribute values. For instance, suppose `p` is a `Poem`, we can create the following attribute values:

```plaintext
new AuthorInfo on p
    author = [name = "Dante"; surname = "Alighieri"; birthYear = 1265]
    year = 1306
end

new Comment on first line of p
    contributor = "John Smith"
    comment_text = "some comment"
end
```

and modify or delete them:

```plaintext
modify Comment on first line of p
    comment_text = "some other comment"
end
remove Comment from first line of p
```

Note that a textual object can be annotated with several attributes, even of the same type.

For this reason, the last `remove` of the previous expression removes all the Comments on the selected object. To remove specific comments, they must be identified first, as in the following expressions where only the comments made by John Smith are removed from the first line of poem `p`:

```plaintext
remove Comment with contributor = "John Smith" from first line of p
```

The query expressions which return attribute values will be discussed with more details in the next section, where it will be shown also the use of the attributes to retrieve textual objects.

7 The query language

In addition to the operators presented on the previous sections, the language has a range of operators on attribute values and objects which forms the base of
a query sublanguage. This will be fully described in the forthcoming reference manual, while in this section we will show only the main operators.

Firstly, is important to note that, since the object text is the full text manipulated by the system, each expression can make an implicit reference to it. For instance, the selection first line of text can be simply expressed as first line. In the examples we will use this abbreviation consistently.

### 7.1 Operators on repeated textual objects

The operators defined over textual objects which are repetitions are either positional, which works by giving selecting elements of a repetition through some form of position (i.e. first, last, etc.), or query-like ones, with a syntax reminiscent of other query languages, like SQL, and which allow the user to select a textual object by specifying conditions on its components, elements or attributes.

The first category of operators includes selection either by ordinal adjectives or by numbers. For instance:

- second line
- first line
- first 3 lines of second poem
- lines(1-3) of second poem
- last stanza of first poem
- stanza(last)

The first expression returns the second line of the first poem of the collection. The second and the third ones are equivalent, where the numbers in parenthesis shows the alternative form of selection (like the final expression). last means the obvious thing, as well as the qualifier middle.

The second category includes operators to select objects by conditional expressions and allow the construction of complex queries. They include:

- `Id in RTO`
- `RTO where B`
- `select E from RTO`
- `each RTO has B`
- `some RTO has B`

where:

- RTO is a repeated textual object;
- B is a boolean expression;
- E is a generic expression;
- Id is a generic identifier.

**Id in RTO**

returns the repeated object RTO where the identifier Id can be used, in the rest of the query, to denote each element of RTO.

**RTO where B**

returns a subset of RTO that satisfy the boolean expression B. The expression has the same type of RTO.
select E from RTO
returns the sequences of the values of the expression E evaluated for each
element in RTO. If the value of the expression is a non repeated textual
object, the result is the repetition of the values. If it is a repeated textual
object, the result is formed by combining all the repetitions in a single one,
so that the type of the result is always a repetition.

each RTO with B
returns true if every element in RTO satisfies B.
some RTO with B
returns true if at least one element in RTO satisfies B.

A boolean expression can be formed with the usual boolean operators. Here
are some simple examples of the use of these operators (count returns the number
of the elements in a repeatable object).

p in Poems where count(Lines of p) > 50
returns the textual object composed by all the poems of the collection with more
than 50 lines. This could be abbreviated in:
Poems where count(Lines) > 50

Lines of first Poem where last Word = "love"
returns a textual object of type Lines which contains all the lines of the first
poem of the collection where the last word is "love".

select last Line of Stanzas from Poems where count(Stanzas) > 3
returns a textual object of type Lines containing all the first lines of all the poems
of the collection which have at least 3 stanzas.

select Word(1-2) of Lines from Stanzas of first Poem
returns a textual object of type Words containing all the first two words of all
the poems of the collection.

Poems where each Line has first Word = "love"
returns all the poems in which each line starts with the word "love".

7.2 Retrieval of objects through attribute values
The query part of Manuzio allows the retrieval of objects also through their
attributes, with the conditions exists AttributeType or exists AttributeType with con-
dition. For instance:

Lines of p where exists Comment
Poems of text where exists AuthorInfo with author.name = "Dante"

The first expression returns a textual object composed by the Lines of Poem p
which have one or more attributes Comment, while the second one returns all the
Poems whose author is Dante. Of course these conditions can be used together
with the others available on textual objects:
An important point to note is that, because of the component relation among object types, the exists operator retrieve objects also if the attribute is present not directly on them, but on their enclosing objects. So we could write:

```
Lines where exists AuthorInfo with author.name = "Dante"
```

which returns all the Lines of the Poems of our collection whose author is Dante. In this way, if, for instance, an attribute Meter is defined as:

```
attribute Meter on Poem, Stanzas, Lines
    code: string
end
```

then we could put a Meter value on an entire poem, if every verse has the same meter, or separately on the different stanzas, or lines. In any case, we could ask for all the lines with a certain meter, independently of the way in which it has been associated to the textual objects:

```
Lines where exists Meter with code = "G"
```

If we are interested in only the attributes directly associated to a certain object, we could use the exists direct operator:

```
Lines where exists direct Comment
```

### 7.3 Retrieval of attributes

Queries allow also the retrieval of attribute values from the text, by giving conditions either on their fields, on the objects to which they have been associated, or both. This is possible since the select operator can return, in effect, any type of language value. For instance,

```
select Comment with contributor = "John\_Smith" from Lines of p
```

returns a value composed by all the comments of the contributor ”John Smith” present on all the lines of the the poem p. Combining all the possibilities, we could form expressions like the followings:

```
select Comment with contributor = "John\_Smith" from Lines where first word = "love"
select AuthorInfo.name from Poems where count(Lines) > 300
```

The first expression returns the comments of John Smith attributed to the lines of the collection which start with “love”, while the second one returns the names of all the authors of poems with more then three hundred lines.

### 8 An overview of the basic architecture of the Manuzio System

We are designing the language presented in this work to be the kernel language of a system with the capability of storing and manipulating complex annotated text collections by different user in a coherent and cooperative way. The main objectives of the system are the following:
1. To provide an efficient way of storing and querying very large quantities of textual material, together with very large quantities of annotations, in the form of attributes.
2. To provide a simple language, Manuzio, to query such collection, extend the types and the texts stored, perform annotations and develop complex programs over them.
3. To allow the access to concurrent users, which can operate on text and annotation through an appropriate set of permissions.
4. To provide simplified mechanisms of search of text and annotations to casual users.
5. To provide mechanisms to visualize both the text and its annotations through a choice of different graphical formats and mediums.
6. To allow the exchange of texts and annotations with other systems by using XML as standard interchange format.

The diagram in Figure 2 depicts the main modules of such a system.

Fig. 2. The system architecture

The import and export modules are used to exchange XML material with external systems. A simple declarative language for mapping textual objects with annotations to XML DTDs will be designed to automatize the work of such modules\textsuperscript{11}. The Schema definition language module parses the type definitions\textsuperscript{11}.

\textsuperscript{11} In the current prototype, they are hand coded.
of the schema and generates automatically the metadata for the system work, as well as Recognizer Functions to import directly textual material. These functions are also used as target from the XML import module to insert data into the database. The data imported are the text, which will be “normalized” according to some user specification, giving a unique representation of it, and the textual objects which fully represent the text according to types defined in the schema.

The Textual Object Database is the persistent memory for text, textual objects and annotations. It will have associative structures to retrieve efficiently the textual objects resulting from the query. The language module will perform query translation, optimization and execution. It will be invoked by a concurrency/transactional module which will allow user interaction either through direct access to the language with a traditional desktop application, or a web application, or through an interface for a traditional programming language to develop complex applications.

Currently we are still designing the system architecture. In the meantime, by using the programming language Ruby, we developed a simple prototype of the language by defining the textual object types for a particular model, and the corresponding main operators, and loading a medium-sized set of data through a special purpose XML parser. We are now testing the main language features with users, as well as developing simple applications.

9 Conclusions

This paper is an introduction to a novel object-oriented model and language for dealing with annotated collection of texts, Manuzio. We have presented the most important language features to model and operate upon a collection of annotated texts, but we have not detailed all the language features, as well as several details on the concept discussed. Moreover, the language is still in the design stage, and we are presenting it in this incomplete form in order to stimulate useful comments from potential users of this technology so that it can be better shaped to the users’ needs.

*** Esempi di applicazioni ad altri contesti ***

Acknowledgments

The work presented in this paper has been stimulated by discussions with Paolo Mastandrea, of the “Dipartimento di Scienze dell’Antichità e del Vicino Oriente” of the University of Venice, and Luigi Tessarolo. Their help is much appreciated.

References


