Preprint. Final version has appeared in Interdisciplinary Science Reviews 40(4): 393–408 (2015)

Software as theory: a case study in the domain of text analysis

Aris Xanthos
Department of Language and Information Sciences
University of Lausanne
Anthropole 3136
CH-1015 Lausanne
aris.xanthos@unil.ch

Abstract

This article proposes a reflection on a specific way of envisioning and valorising the scholarly contribution of scientific software, namely by making explicit the model of data analysis that underlies it. It seeks to illustrate this way of studying a software construct by applying it to a particular text analysis program. Fundamental aspects of this program's design (input and output, data structures, process model, and user interface) are reviewed and discussed from the point of view of their implications in terms of theoretical commitments to a specific conception of text and text analysis. The conclusions of this case study notably emphasise the central role of user modelling in the assessment of scientific software's epistemological contribution as well as the necessity of extending the proposed approach to a broader range of software applications.

Keywords

Software, design, theory, model, text, analysis

Introduction

While the construction of digital artefacts, including software systems, is widely recognized as a key activity in digital Humanities research, there is arguably a less clear-cut consensus as to how the scholarly contribution of software development should be identified and interpreted. In their contribution to the *Debates in the Digital Humanities* volume, Ramsay and Rockwell (2012) review and discuss several ways in which the epistemological status of software products have been or can be envisioned. One of these conceptions revolves around the idea, notably articulated by Willard McCarty, that the primary function of computing in the Humanities is that of providing tools and methodologies for *modelling*, broadly construed as "an iterative process of constructing and developing something like a computational 'knowledge representation'" (McCarty 2002, 104). In this context, Ramsay and Rockwell ask, rhetorically, the question "whether the manipulation of features, objects, and states of interest using the language of coding and programming (however abstracted by graphical systems) constitutes theorizing" (83).

The present contribution relies on the assumption that at least in the case of scientific software, whose main functionality is often to assist users in the analysis of some kind of data, software development necessarily implies the conceptualisation of a formal model or theory of the considered type of data and its analysis. Part of this model can be readily apprehended through the visual inspection of a program's interface and the user interaction processes that it defines. Part of it, however, is bound to remain buried in the source code and concealed from non-technical observers unless it is explicitly documented in some form of discursive fashion: this is typically the case of the data structures and, to some extent, the algorithmic processes that underlie a program's operation.

The purpose of this contribution, then, is to explore the consequences of embracing the conception of programming as theorizing by examining a specific piece of scientific software, in particular text analysis software, in the light of this analogy. I will endeavour to expose and review fundamental aspects of the program's design, with the perspective of making explicit the theory of text and text analysis that is, in effect, formalised by these elements. This is not to say that software cannot "stand for itself wholly without written commentary and explanation" (McCarty 2008). Rather, it is an acknowledgement that although the potential theoretical value of software can be of considerable interest to prospective or effective software users, a sizable portion of is not, by itself, intelligible outside of the restricted circle of software programmers–in fact,

acquiring a basic understanding of a non-trivial program's theoretical underpinnings based on source code only will usually require a fair amount of effort even from seasoned developers.

Arguably the most common discursive approach to the scholarly valorisation of scientific software is by reporting on fruitful uses in specific research contexts. This corresponds to another way of envisioning the theoretical status of computer programs (and more generally digital artefacts) discussed by Ramsay and Rockwell, namely "to think of them as hermeneutical instruments through which other phenomena can be interpreted" (79). That the present contribution does not engage in that type of discourse should not be interpreted as a denial of its potential usefulness, but as a defence of the idea that theoretical discourse about software can "stand for itself wholly" without being supported by empirical results. Furthermore, it is my contention that the availability of such discourse may actually strengthen empirical assessments of a program's hermeneutical value, to the extent that it contributes to a better and more widely accessible understanding of its functionality.

The program that this contribution sets out to scrutinise, *Orange Textable* or simply *Textable* (Xanthos 2014), has been developed by the present author in the context of a pedagogical innovation project funded by the University of Lausanne in 2012, with the perspective of using it for teaching introductory courses on computer-aided text analysis.[1] It is packaged and distributed as an open source extension to the *Orange Canvas* data mining environment (Demšar *et al.* 2004). In order to give a picture of the model of text analysis that underlies *Textable*, the following sections will successively examine the type of data that it takes in input and produces in output, the internal data structures on which it operates, the algorithmic processes it executes, and the interface it presents to the user. The concluding *Discussion* will offer a synthesis of these observations and reflect on what such a case study may teach us about the conception of software as theory and ways in which it should be further developed.

Input and output

A coarse-grained overview of the model of text analysis embodied in Textable can be gained by examining the nature of the data on which it operates and that of the data it produces as a result of its operation. This specification, which directly relates to the question of interoperability in software development, is notably useful for understanding the wider context in which the analytical activity is thus theoretically embedded.

The problem of characterising the "nature" of the data is less innocuous than it seems, because it can be and sometimes needs to be done from several distinct point of views. For instance, merely stating that raw text is the only type of data that Textable accepts as input may not be sufficient for all purposes; a fuller account would include indications concerning the supported text encodings and formats. Because of the versatile approach to text processing that the program seeks to implement (see *Processes* below), the latter kind of information is not as easily specified as the former.[2] Formally, Textable does not require its input to conform to a particular text format and it can work with data in any format that can be expressed within the domain of raw text. The only format for the exploitation of which it offers *specific* facilities is XML, but as we will see, it has generic provisions for the manipulation of a variety of other formats. This mostly agnostic stance with regard to the format of input data is a design choice intended to favour the program's interoperability, that is, the diversity of analytical contexts in which it can be integrated: rather than setting specific expectations on the internal structure of the data, it attempts to provide means of adapting to the widest array of possibly unrecorded structures.

The main type of output that Textable produces are two-dimensional data tables. Whether these tables are exported in text format or represented internally with ad hoc data structures[3] matters little to the present discussion. On the contrary, that the program is fundamentally designed to convert raw text sources into tabular data is of particular relevance to the stated intent of this review. This is indeed the general framework

[1] See http://langtech.ch/textable.

[2] The program can deal with the entire range of encodings that are recognized by the underlying programming language, namely Python v2.7–in particular it is fully Unicode-compliant.

[3] The latter option is what makes it possible to further analyse the data produced by Textable using the tools and methods implemented in Orange Canvas.

in which every text-analytic workflow is ultimately expressed in Textable. In particular, it delegates virtually every operation *bearing on* tables (e.g. data clustering or visualization methods) to other software components, on the grounds that such operations do not pertain to text analysis proper, but to data analysis more generally.[4] This is a crucial observation for understanding the theoretical conception of text analysis of which Textable is a working model.

The second type of output produced by Textable are text data, typically resulting from the preprocessing, recoding, and/or structural reorganization of input text sources. This is another feature whose main purpose is to increase the program's interoperability. For the purpose of this discussion, we may say in summary that Textable implements a model of text analysis according to which text sources are processed to create either new text data or two-dimensional data tables. As we will see in section *Processes* below, the latter kind of output actually covers a variety of distinct table types.

Data structures

In the framework of Textable, the conversion of text strings to data tables (or to other strings) is mediated by a specific data architecture organised around the concept of *segmentation*. From a functional point of view, a segmentation serves as a formal representation of an analysis of a string, whereby an arbitrary large number of substrings are delineated, ordered, and possibly further qualified. These substrings, called *segments*, are specified by two elements: (i) an *address* consisting of a reference to a given string along with a start position and an end position (expressed in character offset relative to the string); (ii) a set of zero or more *annotations*, usually in the form of arbitrary text strings (annotation *values*), each of which is itself indexed by another arbitrary string (annotation *key*).

An illustration may be useful for making these notions more concrete. Consider the following strings:
(1) *a simple example*
(2) *another example*
In this context, the notation (1, 3, 8, {*type*: *ADJ*}) might be taken to conventionally denote the segment running from the third to the eighth character of string (1) and annotated with key *type* and value *ADJ*. The corresponding substring, *simple*, is called the segment's *content*. An example of segmentation, then, might be the following ordered list of segments:
1. (1, 3, 8, {*type*: *ADJ*})
2. (2, 1, 7, {})
3. (1, 3, 16, {*type*: *UNKNOWN*, *eval*: *redundant*})

This illustration emphasizes several important properties of the chosen data architecture–properties that ultimately define the range of possible analyses of a string or set of strings that Textable can formally represent. A key property is the hypothesis of a two-level structure, where a container (segmentation) is associated with a series of contained elements (segments), thus constituting what can be thought of as a minimal instantiation of an *ordered hierarchy of content objects* (DeRose *et al.* 1990). In contrast to the full-fledged OHCO model, however, there cannot be more (nor less) than two levels in this representation. This is a severe limitation, which makes it impossible to properly represent a number of analyses that have proven highly relevant for the description of text data, such as syntactic derivation trees or the hierarchical structure of documents (with chapters consisting of sections, themselves consisting of subsections, and so on). As a concrete consequence of this state of affairs, importing XML data in Textable can only be done at the cost of flattening their structure, which implies a loss of information in general.

It should be noted, however, that the representation of text data in Textable is considerably more flexible than what DeRose and colleagues call *a stream of content objects*, and for certain purposes, more flexible than the OHCO model: indeed, in Textable, it is perfectly acceptable for any two segments of a given segmentation to stand in a relation of containment,[5] partially overlap, or even correspond exactly to the

[4] Orange Canvas itself actually offers a large range of such algorithms.
[5] In this context, segment *x* is said to be *contained* in segment *y* if and only if (i) they refer to the same string, (ii) the start position of *x* is greater than or equal to the start position of *y*, and (iii) the end position of *x* is lesser than or equal to the end position of *y.*

same area of a string.[6] Furthermore, any number of distinct segmentations of a given string (or set of strings) can co-exist at any time, which means that hierarchical levels of segmentations can actually be stored (albeit independently of one another, i.e. without keeping track of hierarchical relationships between specific segments)–a possibility that Textable's analytical algorithms attempt to consistently exploit, as will be discussed in section *Processes* below. For the same reasons, the problem of multiple logical hierarchies mentioned by Renear *et al.* (1996) is easily overcome within this framework.

In fact, the Textable data structure shares less similarities with the OCHO model and its SGML/XML applications than with the *Tipster* architecture (Grishman 1997). The Textable representation can be envisioned as a version of the Tipster model simplified in mainly two ways. First, Textable does not adopt the elaborate typing system by which Tipster specifies annotation values: with very few exceptions, such values can only be strings in Textable, while Tipster distinguishes between Booleans, strings, integers, reals, collections, and so on. The rationale behind this simplification pertains to the program's expected user profile, which is typically that of a non-programming student or scholar in the Humanities, for whom it is common practice to annotate text with text. The Textable annotation model is thus intended to appear as a natural metaphor to such users, while remaining powerful enough to satisfy a variety of operational needs. Implementations of the Tipster model such as GATE (Cunningham *et al.* 1996) are clearly directed towards more technical users, and can arguably not be fully exploited without some background in programming. With regard to the objective of this contribution, this raises the question whether the idealized user type for whom a given piece of scientific software is designed could or even should be construed as an integral part of the underlying theoretical model, considering how significantly it may affect it. In this case, my inclination would be to answer positively: that Textable has been designed mainly for non-technical users is relevant for understanding the conception of text analysis that it embodies, and in particular how it differs from text *engineering*.

The second important way in which Textable's data structures are a simplification of the Tipster architecture is that in the former, a segment necessarily corresponds to a continuous area of a string. Consequently, discontinuity can only be accounted for at the level of the segmentation, with the constraint that a given segmentation should be used for representing *either* a single discontinuous unit *or* a sequence of continuous units. The Tipster model, on the other hand, makes it possible to represent discontinuous units alongside and on the same level as continuous ones. In this case, the design has not been guided by a theoretical preconception as to the irrelevance of the ability to deal properly with discontinuous phenomena, which would actually be very desirable. As a matter of fact, an attempt to create a prototype with this ability was made at some point of Textable's development, but the negative impact on the program's performance (in terms of calculation time) turned out to be prohibitive. The prototype was thus abandoned based on the assumption that discontinuity is not a frequent enough condition in text analysis to justify the computational penalty that the proper handling of discontinuity would imply for the far more typical processing of continuous units–which in itself is indicative of certain preconceptions about the application domain and their relation to the dynamic nature of user interactions that Textable's interface seeks to promote (see *Interface* below).[7]

Processes

The process model adopted in Textable has several notable precursors, the earliest of which is the set of text processing programs developed in the seventies and integrated in the *Unix* operating system (see e.g. Dougherty and O'Reilly 1987). These well-known command-line tools constitute a canonical illustration of the so-called "Unix philosophy" of programming that has been summarized by Douglas McIlroy (quoted in Salus 1994) as "Write programs that do one thing and do it well. Write programs to work together. Write programs that handle text streams, because that is a universal interface". At the core of this approach lie the

[6] Note that even perfectly overlapping segments will be ordered in a given segmentation.

[7] The same kind of reasoning holds for the fact that Textable is not able to represent zero length segments, i.e. elements that have a well-defined position in a text but no actual content, such as page break indications for instance. The possibility of safely assuming that every segment has a non-empty content (as opposed to explicitly testing it) entails a considerable gain of computation time.

notions of modularity and interoperability, which are equally important to another early ancestor of Textable, namely the *TUebingen System of TExt processing Programs* or *TUSTEP* (Ott 1979). The following characterization of TUSTEP can be applied almost without any modification to Unix tools and Textable:

> TUSTEP is a collection of relatively independent programs, each of which offers a well-defined subset of basic operations for processing textual data. It is the task of the user to combine these basic functions in order to arrive at the solution to a given problem (Ott 1994, 197).

This citation also emphasizes the active role of the user when working with software designed in this way: rather than merely feeding predefined analytical processes with new data, she is expected and in fact required to take part in the actual conception of processes. Providing her with a constellation of lower-level interoperable software components rather than with a single, integrated higher-level program simultaneously increases the diversity of analyses she may conduct and her degree of responsibility for the specification of theses analyses, thus blurring in effect the line that separates user from developer. This idea was explicitly thematised already in the mid-nineties in the context of a discussion on the notion of "methodological primitives" in Humanities computing (and in particular text processing) that Willard McCarty has been animating in the Humanist online discussion group:

> It seems to me that the most broadly useful invention would be a set of components the "garden-variety" humanist could assemble quickly into a process. Component-software is not a new idea, but it rests on a genuine intellectual problem, namely to identify the computing primitives of text-processing. If a basic set of primitives were suitably identified and component-packages implemented for them, then perhaps we could close the gap between the person with the problem (conventionally, the scholar) and the one who writes or adapts the tools to deal with this problem (the programmer). When the day arrives that these two can become one on a broad scale, then I think we'll see an enormous leap in the scholarly use of computing (McCarty 1995).

Two decades later, this day has yet to come, but Textable does implement a set of computing primitives on the basis of which a variety of text-analytic workflows may be assembled. There is a considerable degree of overlap between the functionalities offered by these primitives and those defined in Unix and TUSTEP; for instance, all three systems enable the user to view the context of occurrence of query terms in sets of text files, or count the number of word tokens these files contain. The main difference is that Textable is primarily designed to support a hermeneutic interaction with text data while the two others focus rather on text manipulation as part of an edition and publication process.[8]

Another distinctive feature of Textable is its adoption of an object-oriented framework (inherited from Orange Canvas). Thus rather than defining primitive components as *programs* that can be *executed*, as is the case of the Unix and TUSTEP systems, they are conceived as *classes* (called *widgets*), each of which can give rise to an arbitrary large number of *objects* (called *instances*) that the user connects to one another in order to specify a particular workflow (more on this in section *Interface* below). Widgets take various types of data in input and produce various types of data in output; they can be divided into three main categories based on their input and output types: text import widgets, segmentation processing widgets, and table construction widgets.[9] These categories correspond to the main steps of most Textable workflows.

Text import widgets are responsible for importing text data from various sources: keyboard (**Text Field** widget), files (**Text Files**), or web locations (**URLs**). Each instance of these widgets stores one or more strings in memory and returns a segmentation (see *Data structures* above) where each segment corresponds to an entire input string. It is worth stressing the somewhat counter-intuitive fact that when these instances are connected with other instances, the data that are exchanged between them are segmentations, not the

[8] In the case of TUSTEP, the preparation of scholarly editions is explicitly put forward as the system's main function; although Dougherty and O'Reilly's overview of Unix text processing similarly focuses on "the preparation of written documents, especially in the process of producing book-length documents" (1987, xi), the vocation of Unix tools is more general-purpose, as witnessed by their frequent use for system administration.

[9] A fourth, auxiliary category is characterized by the common purpose of its members, namely to facilitate *conversion* between various data types (e.g. from a table in the internal format of Textable to a table in a format that can be processed by the built-in widgets of Orange Canvas). A more detailed description of each widget can be found in the *Reference* section of Textable's online documentation (https://orange-textable.readthedocs.org/en/latest/reference.html).

imported strings themselves, contrary to what happens when two Unix programs are piped together for instance; thus besides their common import function, widgets of this category produce initial *analyses* (in the terms used in the previous section) of the imported data and make them available for further refinement by other widgets.

Segmentation processing widgets take segmentations in input and return new segmentations. Two widgets of this category additionally generate modified text data: **Preprocess** and **Recode**, whose respective purpose is to apply standard preprocessing operations (e.g. reduce uppercase to lowercase or remove diacritic signs) and to systematically replace text areas matching a given regular expression; instances of these widgets actually create and store new strings, which the segmentations they return describe. The other widgets of this category return segmentations that describe the same string(s) as their input segmentations with a different set of segments. In some cases, the output segments have the same address as pre-existing input segments: the **Merge** widget takes two or more segmentations in input and combines their segments into a single segmentation that can be further processed in a unified way; **Select** takes a single segmentation in input and filters part of its segments out based on different criteria (regular expression matching, frequency threshold, or random sampling); finally **Intersect** takes two segmentations in input and retains from the first only those segments whose content does (or does not) appear in the second, which is useful for emulating a "stop list" functionality for instance. Two widgets of this category have the potential for creating new segments, or more precisely segments with new addresses: **Segment** looks for areas matching a regular expression in its input segments and create new output segments for each such area, thus offering a standard way of dividing strings into lines, words, or letters for instance; **Extract XML** looks for occurrences of a given XML element and creates a new segment for each of them (as well as converting its attributes to annotation key–value pairs).[10]

In practice, segmentation processing widgets form the bulk of most Textable workflows, which indicates that the computational primitives they implement are central to the conception of text analysis that underlies the program's design. They account for most of the versatility mentioned in section *Input and output* above, notably thanks to the frequent recourse to regular expressions (in widgets **Recode**, **Select**, and **Segment**): indeed by combining instances of these widgets in various ways it is possible to exploit the structural characteristics of a wide range of input text data–although it may require a certain degree of ingenuity on the part the user. From that point of view, the need to formulate appropriate regular expressions for the configuration of widget instances and to judiciously assemble those instances for handling particular input data is arguably what sets the lower bound of technical expertise required for a fruitful application of the program's model of text analysis.

There is however a common feature of segmentation processing widgets that may have a facilitating effect on the construction of workflows: all of them rely on an implicit mechanism of iteration over input segments, thus enabling the user to dispense with any such concern. For example, when an instance of **Segment** performs a segmentation into words, it automatically processes each input segment successively, regardless of their number. This way, the structure that has been postulated for the representation of text analyses is being systematically taken advantage of for simplifying the conception of analytical workflows by the user.

The last category of widgets are those that take segmentations in input and produce two-dimensional data tables–a functionality that probably constitutes Textable's main trademark when compared to Unix tools or TUSTEP. Widgets belonging to this category usually operate either on a single input segmentation or on two of them; in the latter case, one segmentation is used to specify the *units* of calculation while the other provides the *contexts* in which calculations are performed, and the meaning attached to these terms presents some degree of variation across widget types. Consider the example of widget **Count**, which serves to produce frequency tables. On the basis of a single segmentation, it counts the number of occurrences of each segment type in the data and report the result in a table with a single row and as many columns as there are segment types. If a second segmentation is used to define contexts, separate counts are maintained for each

[10] This category includes a slightly marginal widget type, namely **Display**, whose main functionality is to visualise the details of a given segmentation (essentially the content, address, and annotations of segments). Moreover, it is this widget that offers the possibility of exporting text data as mentioned in *Input and output* above.

context segment type, based on the containment relation mentioned in section *Data structures* above (see note 5); thus the resulting table still contains as many columns as there are segment types in the unit segmentation, but the number of rows is equal to the number of segment types in the *context* segmentation.[11]

The variation in meaning of terms "units" and "contexts" may be illustrated by turning to the **Context** widget, whose purpose is to build concordances and collocation tables. In this case, the unit segmentation serves to identify occurrences of a segment of interest (typically a given word), while the context segmentation specifies the left and right neighbourhoods with which each occurrence of the segment of interest is presented to the user; thus the number of rows in the table is given by the number of *unit* segments and the number of columns is basically set to 3 (left neighbourhood, key segment, right neighbourhood). As this example demonstrates, understanding what counts as unit or context in a given widget is not always straightforward. This remark also holds for the remaining widgets in this category, namely **Length**, which serves to count the (average) number of items in segmentations, **Variety**, whose purpose is to evaluate the diversity of segment types, and **Category**, which makes it possible to extract categorical information associated with segments (typically based on their annotation values).

The flexibility of this model of "segmentation-to-table" conversion is increased by the possibility (very systematically offered by table construction widgets) to substitute the content of segments with their value for some annotation key. For example, given a segmentation of a set of texts into words annotated with their part-of-speech tags, the **Count** widget can be configured to measure the frequency of part-of-speech tags instead of word types. Furthermore, if the segments corresponding to the texts are also annotated, e.g. with indications of author, genre, or period, these metadata can be used for the specification of contexts (i.e. rows of the frequency table); thus if several texts correspond to the same annotation value, say {*genre*: *prose*}, then their word counts will be fused into a single row.

Overall, although the number of *types* of tables that can be produced with a Textable workflow is limited to a dozen (frequency table, concordance, etc.), thanks to the interplay of segmentation processing widgets and table construction widgets, along with the flexibility brought by regular expressions and the possibility of replacing content with annotation, the number of different tables that can be built on the basis of a given set of text data is virtually unlimited. With that said, the richness of possible outcomes which is specifically promoted by the strongly modular conception of text analysis adopted in Textable comes at the cost of an increased degree of involvement and technical expertise on the part of the user. As noted by John Bradley (2000), "Any tool meant to support activities as diverse as those that turn up in humanities text-based computing cannot possibly be trivial to learn or use".

Interface

At some point in the running online discussion about methodological primitives mentioned above, Willard McCarty reflects about a possible user interface for building programs based on these components: "What if we had a visually-orientated 'programming' environment of Lego-like primitives we could plug together?" (2000). In fact, such a visual programming environment had already been implemented in the form of a prototype named *Eye-ConTact* and described by Rockwell and Bradley (1998). Although I was not aware of this work when the first version of Textable was released in 2012, the similarities between the two programs are obvious and, I believe, indicative of a very close conception of text analysis.

From the point of view of interface, Eye-ConTact and Textable have in common the adoption of the so-called *dataflow* paradigm. In this framework, the user constructs workflows that are represented visually as directed graphs in which each node stands for a computational unit (a widget instance, in the Textable jargon, see previous section) and the edges connecting the nodes determine the flow of data in the system. Figure 1 below gives an example of such a visual program, in this case a Textable program designed to replicate an investigation conducted by Svensson (2009) regarding the frequency of expressions "Digital Humanities" and "Humanities computing" over time in the online archives of the Humanist discussion group.

---

[11] A typical example of such a table is the so-called "document–term matrix" used in information retrieval, which gives the frequency distribution of each word across a number of texts (see e.g. Salton and McGill 1986).

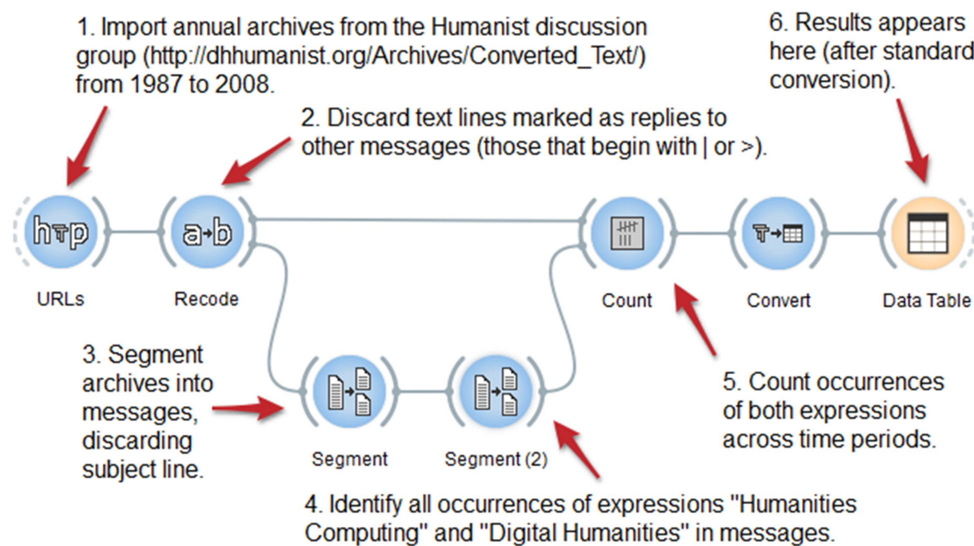

Figure 1. A user-defined visual program in Textable (reproduced with permission from Textable's online documentation).[12]

Each computational unit can and often must be further configured by displaying its individual interface. As an illustration, Figure 2 below shows the configuration of the **Recode** instance in the program represented on Figure 1, which essentially indicates that every occurrence of a line beginning with either a vertical bar (|) or a "greater than" symbol (>) should be deleted (i.e. replaced with an empty string).

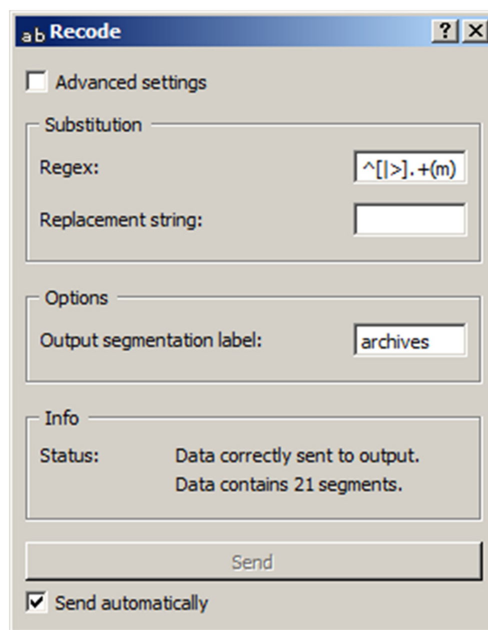

Figure 2. The interface for configuring an instance of the **Recode** widget in Textable (reproduced with permission from Textable's online documentation).[13]

User interfaces such as those of Eye-ConTact or Textable are a solution to the following issues in scientific software design:

> We need tools that ensure that our computing work is clearly and completely logged while it is developing, so that it can be accurately described and later recreated by others... what we need are research tools aimed, not

[12] See https://orange-textable.readthedocs.org/en/latest/illustration.html. The Textable program in question is available for download on that page.
[13] *Ibid.*

> at the researcher for his or her private exploration, but at the researcher's audience who wants to test the insight. (Rockwell and Bradley 1998).

Dataflow programming interfaces simultaneously address both problems raised by Rockwell and Bradley. Indeed, in this context, building a program *is* producing a visual record of an experiment, and sharing the record with an external audience actually makes the program available to them. The investigation of the Humanist archives reported by Svensson (2009) provides an useful example of this, and the reader may find it interesting to compare the representation of the experiment that she can build by reading Svensson's written account with the representation she can form by downloading and manipulating the visual program replicating the experiment (see note 11).

Another important characteristic of dataflow interfaces is the dynamic model of user interaction that they support. Visual programming environments can usually be configured (and sometimes are configured by default) to automatically perform computations rendered necessary by each modification of the interface (typically the creation/deletion of a processing unit or connection, or the modification of a processing unit's parameters). With this configuration, the construction of an analytical workflow becomes a fundamentally interactive process, where every action of the user is directly translated into a modification of the results of the analysis. A high degree of interactivity is a crucial design feature if, as suggested by Rockwell (2003), we are to engage users in a playful approach to text analysis:

> Some possibilities are discarded; some experiments are interesting. The tools make possible a playful interaction. Disciplined play privileges experimentation and modelling over hypothesis testing or concordance publishing. Playful experimentation is a pragmatic approach of trying something, seeing if you obtain interesting results, and if you do, then trying to theorize why those results are interesting rather than starting from articulated principles. (214)

On a rather prosaic note, this vision of interactive computing can only be realized if the execution time of computations does not exceed what is perceived by the user to be an acceptable delay for the task at hand: the benefit of having a modification of the interface immediately trigger calculations is at best considerably lessened and at worst irremediably lost if the result of said calculations is not made available in a sufficiently short time. This is the kind of situation where a trade-off between conflicting desiderata must be sought (in this case execution time versus data and/or process complexity), which is bound to have concrete consequences on the model of text analysis that the program embodies.

Discussion

Having reviewed the main aspects of Textable's design, we are now in position to summarise the conception of text analysis that underlies it. Text analysis is fundamentally construed here as an interactive and (playfully) experimental process by which a researcher builds an analytic workflow that specifies how a set of raw text sources should be processed. Executing such workflows typically results in the creation of various kinds of two-dimensional tables (frequency tables, concordances, etc.) or possibly modified versions of their input text data, which are then used in support of a hermeneutical approach of the sources. From this functional point view, there is a clear difference between text analysis as it is envisioned here and other text-based activities such as publishing or language engineering. It is also worth noting that in this conception, subsequent applications of formal methods (such as clustering, classification, and so on) to the tables resulting from workflow execution are, in effect, excluded from the field of text analysis proper, on the grounds that they pertain to data analysis more generally.

Workflow construction is performed in a visual way, in the sense that it is a direct result of the production by the researcher of a graphical representation of the workflow. Specifically, she creates instances of a predefined set of computational unit types, configures the behaviour of each individual instance, and draws connections between them to determine the flow of data. Each unit type corresponds to a basic text-analytic process–a "methodological primitive" of sorts–which usually belongs to one of the following classes of functionalities: text import, segmentation processing, and table construction. Segmentations are data of a particular type, which serve to represent segmental analyses of strings (by outlining continuous areas within strings and possibly linking them with arbitrary annotations) and which effectively mediate the transition from text sources to tables. By judiciously combining segmentation processing units and configuring them

(which often involves writing regular expressions), the researcher can flexibly adapt the analysis to a great variety of input data. Tables are then typically produced by relating two segmentations in a specific way, with one of them defining the units of analysis and the other providing the contexts of analysis. The range of possible analyses is made even wider by the systematic possibility of operating on annotation values associated with segments rather than on their contents.

The overarching role of the user model has emerged as a crucial aspect of this review of Textable. In a number of instances, our reflections have led us to thematise in an explicit fashion the role of an idealized user, the skills she is expected to have at her disposal, the objectives she supposedly seeks to fulfil, and so on. From this perspective, the case study points to the risk of understating the importance of user modelling in a context of "theoretical software analysis", and suggests that there is ample room for improvement along this line of research: in order to develop better models of what text and text analysis really are, we need to strive for a better understanding of what a text researcher really is.

The analysis that has been conducted in this article is but the first step of a broader endeavour. Indeed, applying this approach to a variety of software packages is desirable for at least two reasons. On the one hand, while the selection of design features that has been used here is fairly standard and well adapted to the classical software development paradigm, it could be profitably refined as a result of its application to other software.[14] On the other hand, disposing of comparative results is a precondition for abstracting away from idiosyncrasies and moving to the more fundamental–and more interesting–level of the theory of text analysis that underlies computerized text analysis in general (as opposed to any specific piece of software). This extension to the "meta-theoretical" level should take into account not only programs that can be used *in place of* one another, but also programs that can be used *in combination with* one another, in order for us to develop a better understanding of the wider data-analytic context in which text analysis is embedded.

A lesson that can be drawn from this case study is that while the idealized division of software design aspects that has been adopted here is a potentially useful structuring principle, many relevant observations can be made about the *interactions* between these aspects. An example of this is the necessary trade-off between execution time, which must be minimised in order to enhance user interaction, and complexity of data structures and processes, which in some cases is a necessary condition for a model of data analysis to be applicable to "real-sized" research questions. Such trade-offs usually result from the finiteness of certain resources, and while the resources in question may not have a straightforward analogue outside the domain of software development, the need to strike a balance between conflicting criteria is not a rare occurrence in more traditional ways of expressing theoretical constructions. For example, a scholar writing a book on literary theory might be similarly striving to find an optimal compromise between the desire (or constraint) to keep the book shorter than a given number of pages and the need to provide a sufficient number of examples of literary analyses. From that point of view, the gap between scientific software and scholarly publication may not be as wide as it is sometimes believed to be.

Acknowledgments

The development of the software program being analysed in this case study has been financed by the Teaching innovation fund, the Faculty of Arts, and the department of Language and Information Sciences at the University of Lausanne. This article has benefited from useful comments and suggestions by Tara Andrews and two anonymous reviewers.

References

Bradley, John. 2000. Re: 14.0258 methodological primitives? In Humanist 14.272 (26/09/00). http://www.dhhumanist.org/.

[14] As noted by an anonymous reviewer, using a more complex set of design features would be necessary for this approach to be extended to such models as genetic or adaptive programming, for instance.

Cunningham, Hamish, Wilks, Yorick, and Gaizauskas, Robert. 1996. GATE — a General Architecture for Text Engineering. In *Proceedings of the 16th Conference on Computational Linguistics (COLING-96), Copenhagen, August 1996*, 1057–1060.

Demšar, Janez, Zupan, Blaž, Leban, Gregor, and Curk, Tomaz. 2004. Orange: From Experimental Machine Learning to Interactive Data Mining. In *Knowledge Discovery in Databases: PKDD 2004*. Lecture Notes in Computer Science, 3202, ed. Jean-François Boulicaut, Floriana Esposito, Fosca Giannotti, and Dino Pedreschi, 537–539. Berlin/Heidelberg: Springer.

DeRose, Steven, Durand, David, Mylonas, Elli, and Renear, Allen. 1990. What is Text, Really? *Journal of Computing in Higher Education* 1(2): 3–26.

Dougherty, Dale, and O'Reilly, Tim, eds. 1987. *UNIX Text Processing*. Indianapolis: Hayden Books.

Grishman, Ralph. 1997. TIPSTER Architecture Design Document Version 2.3. Technical report, DARPA.

McCarty, Willard. 2008. Writing and pioneering. In Humanist 22.403 (24/12/08). http://www.dhhumanist.org/.

McCarty, Willard. 2002. Humanities computing: essential problems, experimental practice. *Literary and Linguistic Computing* 17(1): 103–125.

McCarty, Willard. 2000. if only we could speak it easily? In Humanist 13.443 (25/02/00). http://www.dhhumanist.org/.

McCarty, Willard. 1995. Programming for the humanities. In Humanist 9.301 (18/11/95). http://www.dhhumanist.org/.

Ott, Wilhelm. 1994. Modularity, Professionality, Integration: A Conception Revisited. In *ALLC-ACH 92, Conference Abstracts and Programme, Oxford, 6-9 April 1992*, 197–199.

Ott, Wilhelm. 1979. A Text Processing System for the Preparation of Critical Editions. *Computers and the Humanities* 13(1): 29–35.

Ramsay, Stephen, and Rockwell, Geoffrey. 2012. Developing Things: Notes toward an Epistemology of Building in the Digital Humanities. In *Debates in the Digital Humanities*, ed. Matthew K. Gold, 75–84. Minneapolis: University of Minnesota Press.

Renear, Allen, Mylonas, Elli, and Durand, David. 1996. Refining Our Notion of What Text Really Is: The Problem of Overlapping Hierarchies. In Susan Hockey and Nancy Ide (Eds.), *Research in Humanities Computing 4: Selected Papers from the ALLC/ACH Conference, Christ Church Oxford, April 1992*, 263–280. Oxford: Oxford University Press.

Rockwell, Geoffrey. 2003. What is text analysis, really? *Literary and Linguistic Computing* 18(2): 209–219.

Rockwell, Geoffrey, and Bradley, John. 1998. Eye-ConTact: Towards a New Design for Research Text Tools. *Computing in the Humanities Working Papers*, A.4.

Salton, Gerard, and McGill, Michael. 1986. *Introduction to Modern Information Retrieval*. New York: McGraw-Hill, Inc.

Salus, Peter. 1994. *A quarter century of UNIX*. New York: ACM Press/Addison-Wesley Publishing Co.

Svensson, Patrik. 2009. Humanities Computing as Digital Humanities. *Digital Humanities Quarterly* 3(3). http://digitalhumanities.org/dhq/vol/3/3/000065/000065.html.

Xanthos, Aris. 2014. Textable: programmation visuelle pour l'analyse de données textuelles. In A*ctes des 12èmes Journées internationales d'analyse statistique des données textuelles (JADT 2014), Paris, June 3–6, 2014*, 691-703.

Biography

Aris Xanthos is a senior lecturer in Humanities computing at the department of Language and Information Sciences at the University of Lausanne. His current research interests lie mainly in the study and development of innovative methods and interfaces for the analysis of Arts and Humanities data, with a particular emphasis on their textual and linguistic aspects. He is the author of several open source software products, notably the *Orange Textable* visual programming environment for text analysis.