Scholarly and Research Communication
VOLUME 6 / ISSUE 2 / 2015

Why Experimental Interfaces Should Include an Application Programming Interface

Stan Ruecker, Peter Hodges, Nayaab Lokhandwala, & Szu-Ying Ching
IIT Institute of Design

Jennifer Windsor & Omar Rodriquez
University of Alberta

Antonio Hudson
Illinois Institute of Techology

INKE Research Group

Abstract
An Application Programming Interface (API) can serve as a mechanism for separating interface concerns on the one hand from data and processing on the other, allowing for easier implementation of alternative human-computer interfaces. The API can also be used as a sounding board for ideas about what an interface should and should not accomplish. Our discussion will take as its case study our recent work in designing experimental interfaces for the visual construction of Boolean queries, for a project we have previously called the Mandala Browser.

Keywords
Application Programming Interface; API; Experimental interface design; Digital humanities; Prototype

Stan Ruecker is Associate Professor at the Institute of Design, Illinois Institute of Technology. Email: sruecker @id.iit.edu .

Peter Hodges is a PhD student at the Institute of Design, Illinois Institute of Technology. Email: phodges @id.iit.edu .

Nayaab Lokhandwala is an MDes student at the Institute of Design, Illinois Institute of Technology. Email: nayaab @id.iit.edu .

Szu-Ying Ching is an MDes student at the Institute of Design, Illinois Institute of Technology. Email: sching @id.iit.edu .

Jennifer Windsor is an MA student at the University of Alberta. Email: jjwindsor @gmail.com .

Antonio Hudson is a BSc student at the Illinois Institute of Technology. Email: ahudson2@hawk.iit.edu .

Omar Rodriquez is an IT specialist at the University of Alberta. Email: omar .rodrigueza@gmail.com .

CISP Press
Scholarly and Research Communication
Volume 6, Issue 2, Article ID 0201220, 11 pages
Journal URL: www.src-online.ca
Received June 1, 2015, Accepted July 13, 2015, Published October 14, 2015

Stan Ruecker, Peter Hodges, Nayaab Lokhandwala, Szu-Ying Ching, Jennifer Windsor, Antonio Hudson, & Omar Rodriquez. (2015). Why Experimental Interfaces Should Include an Application Programming Interface. Scholarly and Research Communication, 6(2): 0201220, 11 pp.

© 2015 Stan Ruecker, Peter Hodges, Nayaab Lokhandwala, Szu-Ying Ching, Jennifer Windsor, Antonio Hudson, & Omar Rodriquez. This Open Access article is distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/licenses/by-nc-nd/2.5/ca), which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Implementing New Knowledge Environments (INKE) is a collaborative research intervention exploring electronic text, digital humanities, and scholarly communication. The international team involves over 42 researchers, 53 GRAs, 4 staff, 19 postdocs, and 30 partners. Website: http://inke.ca

Introduction

In this article, we discuss the potential for a central role in experimental interface design for what some people consider a relatively secondary component of human-computer interaction systems: the Application Programming Interface (API). Researchers developing experimental ICT interface prototypes have historically treated the API as a system element meriting little attention. The goal of prototype development is to generate new knowledge with minimal resource expenditure. Prototype system capability to address a research question and begin producing evidence is sufficient resource expenditure. Seemingly necessary functionality is often not added to experimental prototypes, on the basis that everyone already knows this functionality is necessary – the prototype developer judges that any potential benefit is considerably less than the time, energy, and cost required to include it. Often developers judge that the API falls within this category. Primarily used for providing programmatic access to collection data for programmers not working directly with the collection, the API is often released to the public some years after the first release of a production system (e.g., Flickr, Twitter).

The concept of the API originated in the eighties in the software development community. It was used then, and is still used, as a means of exchanging data across organizational boundaries. With the advent of the Web, and particularly since the turn of the millennium, it has also been used to give public access to collection resources, whether for analysis or to allow the development of interface alternatives. Some popular APIs of this kind are available through Flickr, Twitter, and Google Maps.

In terms of technology, the most common way of creating a Web-based API is using Representational State Transfer (REST), where the underlying resources are accessed through a URL that returns XML or JSON rather than HTML. Discussions of APIs tend to focus on methods for measuring the usability and quality of both the software and the documentation. Umer Farooq and Dieter Zirkler (2010), for instance, propose evaluating usability through peer reviews, while Jens Gerken, Hans-Christian Jetter, Michael Zöllner, Martin Mader, and Harald Reiterer (2011) suggest that concept maps might be helpful. Hewijin Christine Jiau and Feng-Pu Yang (2012) offer a strategy for the reuse of crowdsourced documentation to reduce the bottleneck in open source documentation of APIs, while Robert Watson, Mark Stamnes, Jacob Jeannot-Schroeder, and Jan Spyridakis (2013) report from their survey of 33 popular open source projects that considerable attention to documentation was evident: most of the elements that had previously been identified as desirable for documentation to contain were present.

From a business perspective, the quality of an API can be of significant advantage – or disadvantage. For example, the API to the Flickr archive resulted in a variety of alternative interfaces for the photo collection, increasing its use. The API also provided researchers with an opportunity to use Flickr as a data source, further extending its reach and influence. Neil Mansilla (2012) lists another half a dozen companies that have been celebrated for their public APIs, including eBay, Expedia, *USA Today*, Rovi, and Klout. Netflix, on the other hand, has been widely recognized for its repeated failure to create a public API, although its internal API is another story (Jacobson, 2014). However, more than reputation among the "digiterati" might be at stake.

According to Joshua Bloch (2005), in some cases poor APIs have had a negative effect on the corporate bottom line, and have occasionally even led to bankruptcy.

On a more abstract level, Cleidson de Souza, David Redmiles, Li-Te Cheng, David Millen, and John Patterson (2004) discuss the sociological ramifications of the development and use of APIs within an organization. They point out that one of the benefits – namely the separation of concerns through "information hiding" – is also the source of one of the drawbacks, which is the reduced collaboration between the various programming teams involved. The solution they propose to this dilemma, on which we elaborate below in our own context, is to use the design of the API as a mediating artifact – an opportunity for the various teams to develop a shared understanding of the larger system of which the API is a component.

Separation of concerns

Programmers commonly speak of the "separation of concerns," which means, essentially, to divide a project's code into relatively independent parts. Strategies for accomplishing this include modularity, layering, and object-orientation. However, in practical terms the separation of concerns is not always a goal that is easy to meet. The reasons are several, including lack of clarity about the system architecture, uncertainty about where to draw the line between concerns, and fundamental disagreement with the principle. The conceptual framework of object-oriented programming arguably lends itself (e.g., Pree, 1991) to a kind of separation that is not useful, and may even be counterproductive, for the purposes of programming and testing a variety of human-computer interfaces that access the same data and processing.

The reason it might not be useful is that the connection between objects and their methods naturally leads to a mental frame where the separation of concerns has already been effectively managed. Since different objects represent unique concerns, the need to pay special attention is reduced or non-existent – the code will be automatically easier to maintain and reuse, since adjustments are only necessary at the appropriate place in the object hierarchy.

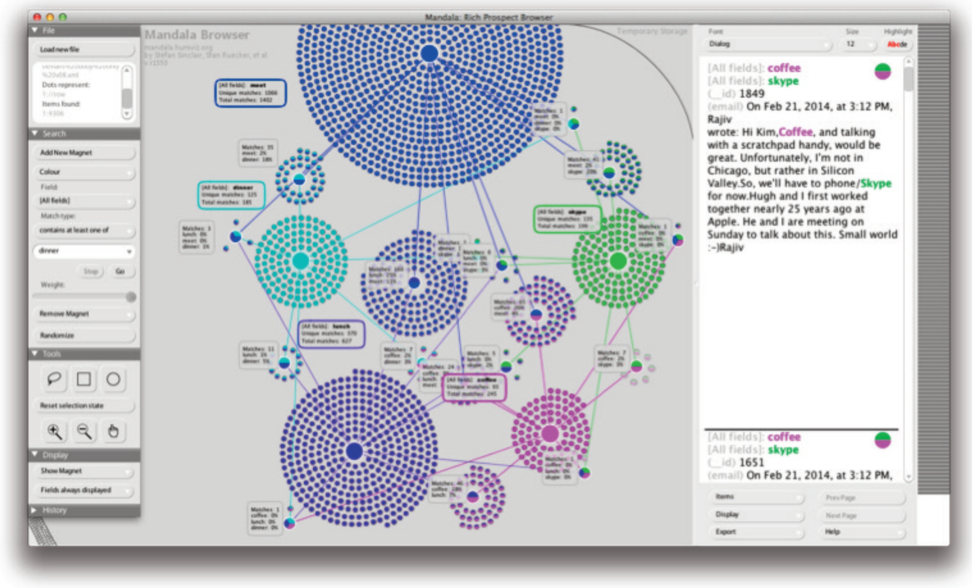
However, in experimental settings the ability to switch among interface alternatives is a productive advantage in at least two situations. In one case, it supports the comparative testing of similar affordances, where the interfaces offer the same functions in different ways. In the second instance, the ability to quickly switch in a new interface allows for introducing and studying new affordances. From this perspective, the primary pair of concerns that need to be kept separate are really the human-computer interface and everything else. In terms of development projects, especially at a large scale, one strategy is the design and use of service-oriented architecture (SOA) or service systems (e.g., Kontogiannis, Lewis, Smith, Litoiu, Muller, Schuster, & Stroulia, 2007), which involve a similar concept of flexibility, but typically require a fairly significant investment.

Thinking by defining the API

Even in an experimental setting, the design and development of an API does represent an additional investment of resources beyond what is arguably essential for an individual project. It requires the programmers, who may be used to thinking of separation in terms of objects, to think in terms of layers. It necessitates a shift in attention from the primary objective of the project toward a component that may or may not prove valuable, but will in any case introduce additional complexity.

If an API is nonetheless desirable, expending effort on deciding what should be included and what should be left aside, whether for a future iteration or for the foreseeable future, is worthwhile. The alternative is to produce an experimental prototype that is not flexible in the sense that it allows for interface variations. In the Mandala Browser project, for example, we had an existing prototype (Figure 1) where the code between the human-computer interface, the data, and the processing were tightly coupled.

Figure 1: The existing Mandala Browser interface, before rewriting the code to add an API. This version shows an analysis of email messages (email address redacted).



[All fields]: coffee
[All fields]: skype
(_id) 1849
(email) On Feb 21, 2014, at 3:12 PM, Rajiv
wrote: Hi Kim,Coffee, and talking with a scratchpad handy, would be great. Unfortunately, I'm not in Chicago, but rather in Silicon Valley.So, we'll have to phone/Skype for now.Hugh and I first worked together nearly 25 years ago at Apple. He and I are meeting on Sunday to talk about this. Small world :-)Rajiv

[All fields]: coffee
[All fields]: skype
(_id) 1651
(email) On Feb 21, 2014, at 3:12 PM,

An API for the Mandala Browser would not necessarily have included all the functionality that the tightly coupled prototype eventually had, such as the ability to produce print-quality screenshots. However, many of the core functions could be made available.

The full set of functions in the existing Mandala Browser

From top to bottom on the left side of the screen:

- Opening a file and reading the XML;
- Changing the colour of magnets;
- Managing the data and structure in such a manner that it can be made searchable;
- Shifting the strength of a magnet to modify locations of subsets of dots;
- Selecting a random search pattern;
- Choosing multiple dots for text display on the right-hand side;
- Zooming in on the magnet display and moving it around;

- Toggling what data is displayed next to the magnet;
- Toggling what text fields are displayed on the right from the XML; and
- Tracing the history of the searches.

From the central display:

- Clicking a dot toggles its text for reading on the right; and
- Double-clicking a magnet toggles all text for its associated dots.

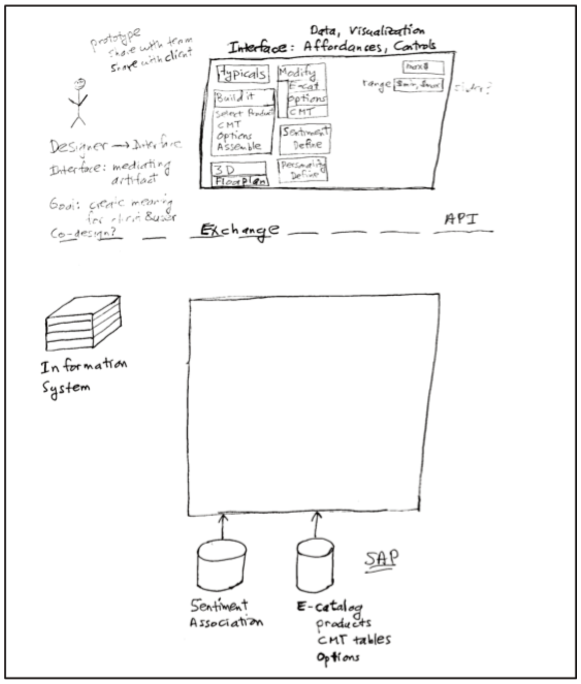
From the right side of the screen:

- Displaying reading text;
- Indicating the magnet associated with the text;
- Displaying an interactive microtext column;
- Toggling full or truncated display of text items;
- Toggling display of text with field names or without;
- Exporting the current magnet configuration;
- Exporting a print resolution screenshot (300 dpi); and
- Providing help.

In addition, thinking in terms of future iterations may suggest some functions that were never added, such as user annotation of searches. Date ranges, or in fact any numeric ranges, although not currently handled, would also be a useful addition.

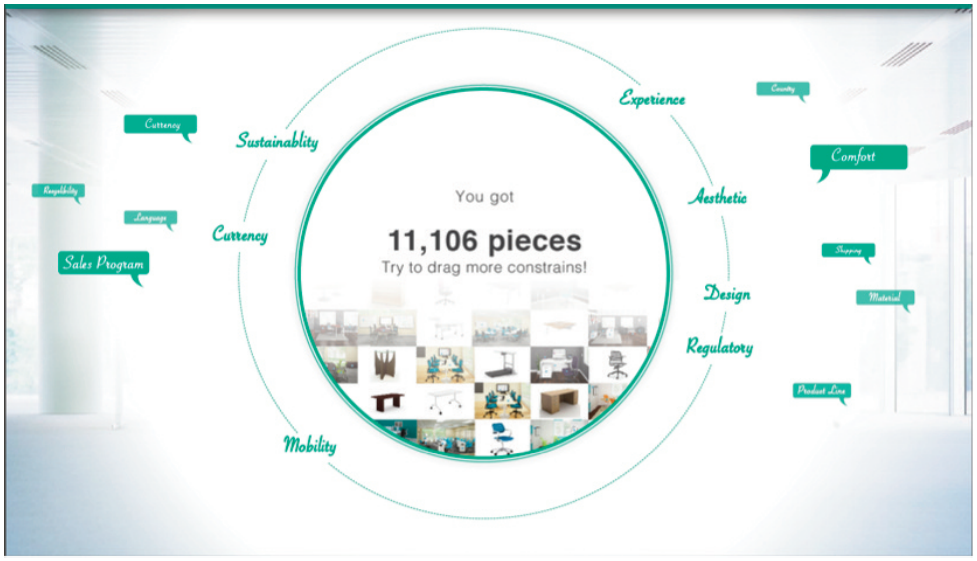
An immediate issue in this design context is what processing should be done below the API, and what should be delegated to the interface. For example, if a list can be sorted in various ways by the user, should the API return the list already sorted, or should the interface code be responsible for managing its own sorting? Similarly, if a snippet of text has been located using search terms, should the API return the snippet with the search terms already identified, or should the interface carry out what is basically a second search to find and highlight them throughout the document? Figure 2 shows an early prototype API function drawing as we began to address these issues. In general, we had three potential types of functionality in mind: first was the existing purpose of working with XML-encoded primary and secondary documents in the humanities; second was identifying and configuring products in a large manufacturing company; and third was the analysis of conversation, whether in the form of transcripts or correspondence.

Figure 2: Interface, API, and everything else



The sketch, developed in the context of our second use case, shows some more specific requirements, such as the possibility of doing sentiment analysis and association, as well as integration with other data stores such as SAP™ (a common enterprise data management system). Since the proposed design context was commercial, potentially dealing with tens of thousands of items, we decided to focus on a rich-prospect representation of the available metadata. We relegated the contents to a central space that would contain a number until the user applied sufficient filtering to allow visibility of individual items (see Figure 3).

Figure 3: A focus on the variety of types of metadata that could appear in this alternative to the current Mandala Browser interface is suggested by the words shown around the central display area.



To support this proposed interface, we developed the following preliminary list of API calls, which we realized would be subject to iterative refinement, as well as possible future additions:

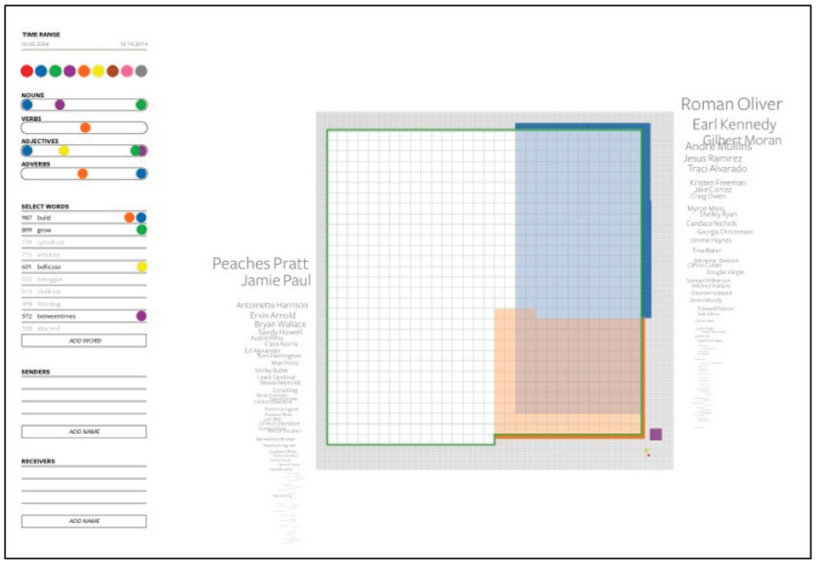
Example function calls

- getProduct(productLine, productName, productFeatureList(pColor = color, pMaterial = material, pTexture = texture), productOptionsList(Options), pCostRange = CostRange);
- getProduct(productFeaturesList);
- getProduct(productOptions);
- getProduct(productStyleNumber);
- getProduct(productStyleNumber, productFeatureList, productOptions);
- getCategories(); // returns the categories of products defined in the backend; for populating category display visualizations in the interface;
- getCategorySubTypes(); //list of product category subtype; e.g., under chairs: adjustable arm, mobile …;
- getProductCombinations(productLineList, productFeatureMatrix, productOptionsMatrix);
- getProduct(categoryType, categorySubType, pColor);
- getProduct(categoryType, categorySubType, pMaterial);
- getProductTechnicalSpecs(ProductLine, productName, productTechnicalSpecs);

- getProductMass(productLine, productName, productStyleNumber);
- getFeatureList(productLine, productName);
- getOptionList(productLine, productName);
- getProduct(CategoryType, CategorySubType, costRange);
- getMediaTypes(); // returns the types of media in the backend; e.g., pictures, 2-D drawing, 3-D, video …;
- ConstrainVisualReturnType(); e.g. specify only pictures or 2-D drawings;
- ConstrainTextReturnType();
- ConstrainNumericReturnType();
- ConstrainDataType(); e.g., manufacturing, released only … .

The new API calls are useful for the new design because they allow the user to specify a broad range of product attributes and features, as well as the form of the visuals that will be returned instead of the coloured dots. They represent, however, only a small part of the functionality of the original system, selected pieces of which would need to be accommodated as appropriate with corresponding calls.

Figure 4: Topics of interest are identified algorithmically, and then displayed as coloured blocks of documents in the grid on the right.

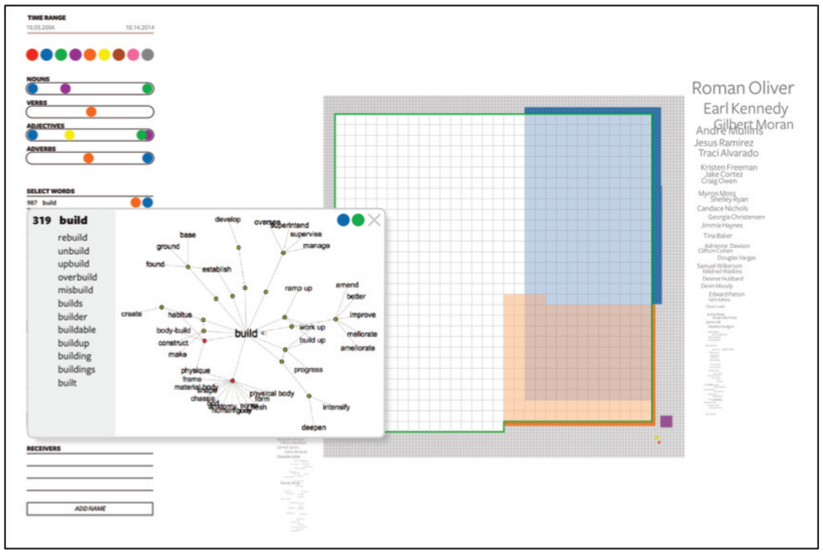


However, further consideration reveals additional potential in the proposed API approach. With some slight expansion, the system could provide even more flexibility for the user. What if, for instance, we were reading other kinds of materials, for example where time was a factor, or dependency, or interchange? In order to address these questions, we turned to a discussion of our third example (See Figures 4-9), where we were considering the use of the Mandala Browser in the context of understanding some form of conversation – perhaps, for instance, an email exchange. Accordingly, we anticipated that we would need to draw on the text of individual correspondences, the metadata associated with those correspondences (e.g., date, senders, and receivers), and also dictionary and thesauri data. To identify ideas, themes, relationships, and patterns within a large collection of texts, the functionality to cross-reference words and phrases with parts of speech, lemma, and lexemes is useful. To understand the particularities of the necessary data involved, we developed a five-step scenario.

Step 1. On initiation (Figure 4), the visualization system algorithmically identifies "topics of interest" (aspects of which are represented by coloured tokens in the facets column) and displays them with correspondingly coloured, and sometimes overlapping, shapes on a grid at the right. Colourless or unselected squares on the grid represent nonspecific documents that fall outside of the identified topics. A "topic of interest" collection might, for example, be an exceptionally large number of certain words or combinations of words, or it might be a large number, or alternatively perhaps a complete absence, of certain words or parts of speech in the collection as a whole, or just in the correspondence of specific senders, and so on.

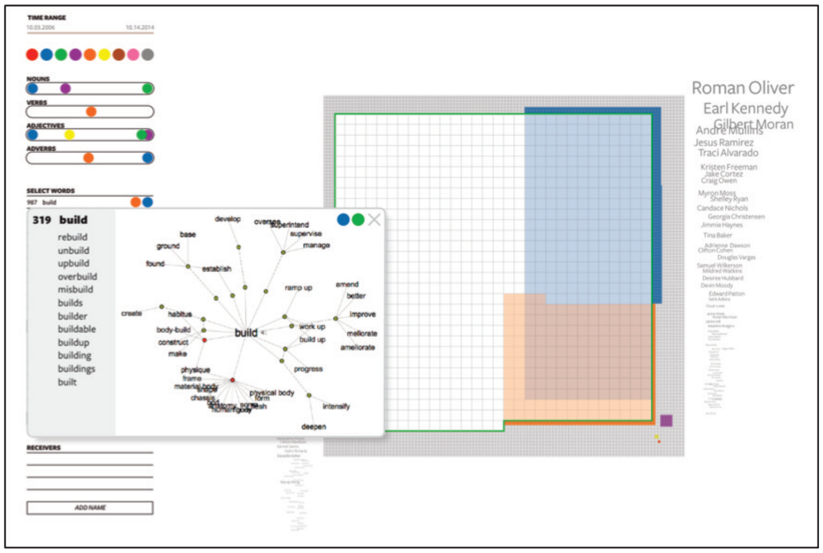
At this stage, the interface is accessing the data of the correspondences, both the comprising text and what can be considered the metadata regarding them – e.g., the date the correspondence was sent, the sender, and the receiver. In the "parts of speech," the system also cross-references dictionary/thesauri data to identify which words are nouns, verbs, adjectives, and adverbs.

Figure 5: The reader is fine-tuning the display.



The visualization to the right of the facet panel displays these topics as coloured squares (corresponding to the colour of the tokens) on an expanding and contracting grid – expanding to best display selected topics, contracting to minimize topics not being examined (while still keeping them visible for context) – and uncoloured squares that represent the remaining, unspecified documents of the full data set. Topics can overlap and create a subgroup, which may be of interest in itself.

Figure 6: The names of document senders appear on the left; recipients on the right.

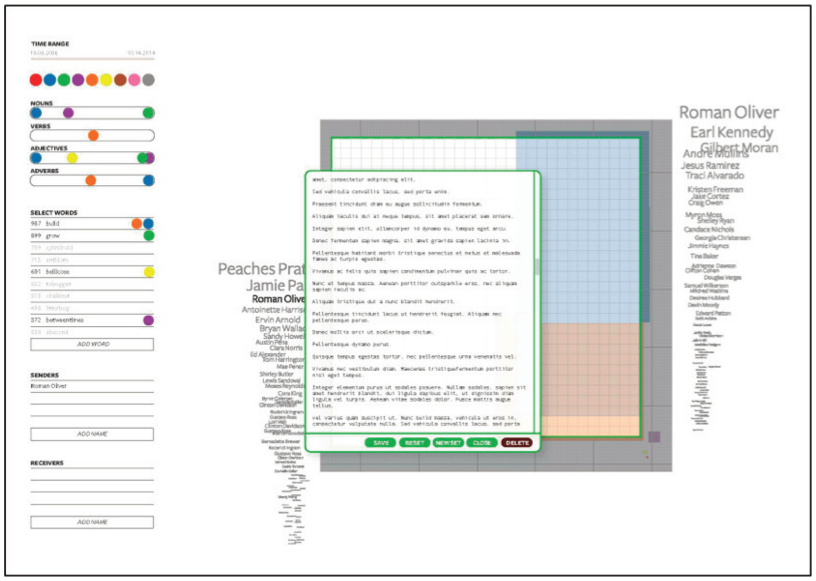


Step 2. After the system identifies topics and marks them with coloured tokens, users can adjust token placement, making the topic squares on the grid larger or smaller and affecting overlap. At this stage users also have the opportunity to fine-tune the specific meaning of specific words by selecting or deselecting lexemes and thesauri offerings in the "meaning" panel (see Figure 5). This section directly accesses dictionary and thesauri data to then cross-reference against the textual data of the correspondences, again affecting the size and overlap of squares on the visualization grid.

Step 3. The group of names to the left of the grid represents "senders" of correspondences; the group on the right, "receivers." The size and proximity to the top of the list represents a greater number of correspondences sent or received, calculated from the metadata of the collection of correspondences. Names can be dragged to the "senders" or "receivers" list on the facet panel to limit the entire data set to just the correspondences of those senders or receivers (potentially shrinking the size of several of the topic squares on the grid), or to an assigned a topic icon (to potentially shrink the size of just that topic square on the grid) (see Figure 6).

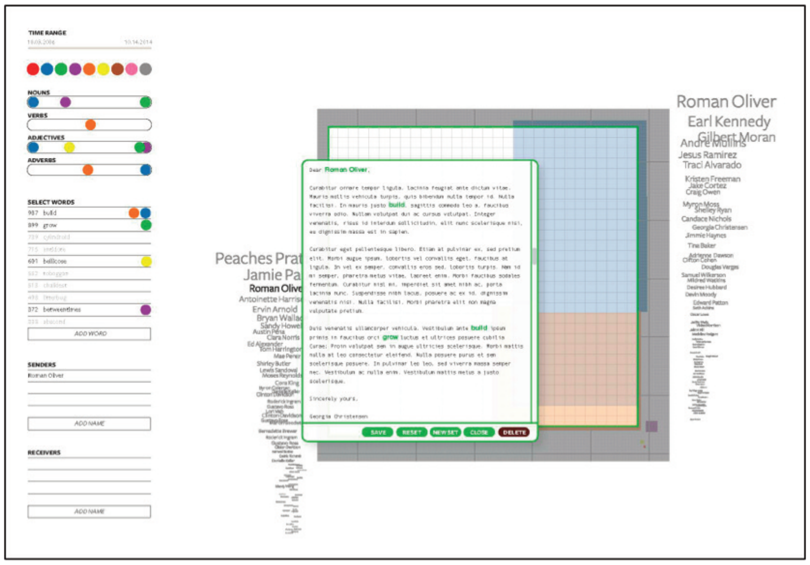
Step 4. Selecting an entire topic square on the grid expands it to reveal information about that collection of data, such as why the system initially identified it as "interesting," the total number of individual documents, if and how it has been altered by the user, etc. The document represents a unique subset of all other data. Among options to reset to system generated facets, or to close or delete this set, there is also the option to save this selection of correspondences somewhere else, and the option to use just this selection as a subset of the original data and open it in a new visualization window (see Figure 7).

Figure 7: Selecting an entire topic square yields a screen that shows its features.



Step 5. Selecting just one square on the grid provides a random example of a document that occurs within that topic's data subset. Conversely, selecting a single document outside of any topic's data subset provides a random example of a document that eludes all of the topics' established facets. Data accessed in this part of the visualization includes the text of the correspondence itself and metadata – including the sender(s), the receiver(s), and the date it was sent – highlighting everything specifically pointed to in this data subset. The system draws on the select words and phrases identified either in the facet panel or the meaning panel and highlights them too (see Figure 8).

Figure 8: A random sample of a document either from within a subset or from outside any given subset.



The outcome of this process was a rethinking of the API from a more general perspective. Items like the sender, receiver, date, and so on could be handled through the generic XML parsing and searching functions. However, we had added some new files such as the dictionary and thesaurus, and some new processes such as topic modelling. In addition, the product-centric naming of the API calls in our second example simply did not make sense when the interface was interacting with an email collection.

The two perspectives of programming and design

It had become clear to us at this stage that producing an API that was useful for multiple interfaces was going to prove challenging, in particular for cases where the underlying collection contents were significantly different. In order to address the design goal of having a common back end for a variety of human-computer interfaces, it would be useful to figure out what can be shared across the common API and what should be customized, and whether or not there is a sufficient proportion of common calls to make the exercise worthwhile.

From the programming perspective, design changes to an interface can sometimes come to seem like a form of scope creep. This is particularly true in cases where it is debatable whether some experimental design feature that requires significant programming effort may actually produce sufficient value in terms of new knowledge. Another possible frustration is that experimental designs often seek not just to provide new functionality, but also to represent functionality in new ways.

One communication strategy that both perspectives can share is the clear expression of the research questions that are being addressed with each feature, and with the combination of features. Positioning the discussion at this higher level of abstraction has the potential to help accommodate the concerns of both perspectives.

Conclusion

The inclusion of an API in an experimental human-computer interface project can involve a significant investment of resources, so the advantages must be weighed against the disadvantages. One of the advantages is that the underlying assets can be readily reused to support other kinds of interface solutions designed by the team, potentially reducing expenditures of time, energy, and funding on future iterations of the project or in other projects with similar requirements. Another advantage is that the design of the API can help to clarify thinking across the team about the kinds of affordances the system will privilege and the kinds it will leave aside. Finally, an API can potentially make the assets accessible to other teams on other projects, following the open source model.

The disadvantages, however, include not only the time and cost, but also the distraction from the primary purpose of the project, which resides in the reification of ideas that can help address a research question. Another disadvantage is that the skills to develop a good API are not the same as the skills required for the other typical aspects of a research project, so they either need to be found or developed. Finally, it would appear that an API is, at least in this context, nearly always a work in progress, which means that it needs to be included not as a single stage, but rather as an evolving part of a project plan. In this respect, it involves a commitment beyond the addition of a typical "feature."

Websites

Flickr, http://flickr.com/

Google Maps, https://www.google.ca/maps?source=tldso

Mandala Browser, http://mandala.humviz.org/

Twitter, https://twitter.com/

References

Bloch, J. (2005). How to design a good API and why it matters [keynote]. Library-Centric Software Design (LCSD) workshop at *Object-Oriented Programming, Systems, Languages and Applications* (*OOPSLA*), San Diego, California, October 16-20. URL: http://lcsd05.cs.tamu.edu/#keynote [May 31, 2015].

de Souza, C.R.B., Redmiles, D., Cheng, L., Millen, D., & Patterson, J. (2004). Sometimes you need to see through walls - A field study of application programming interfaces. *Proc. CSCW (2004),* 63-71.

Farooq, Umer, & Zirkler, Dieter. (2010). API peer reviews: A method for evaluating usability of application programming interfaces. In Proceedings of the 2010 ACM conference on Computer supported cooperative work (CSCW '10). New York, NY: ACM. URL: http://dl.acm.org/citation.cfm?id=1718957 [May 31, 2015].

Ganapathy, Vinod, Seshia, Sanjit A., Jha, Somesh, Reps, Thomas W. , & Bryant, Randal E.. (2005). Automatic discovery of API-level exploits. In Proceedings of the 27th international conference on Software engineering (ICSE '05) (pp. 312-321). New York, NY: ACM. URL: http://dl.acm.org/citation.cfm?id=1062518 [May 31, 2015].

Gerken, Jens, Jetter, Hans-Christian, Zöllner, Michael, Mader, Martin, & Reiterer, Harald. (2011). The concept maps method as a tool to evaluate the usability of APIs. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '11)* (pp. 3373-3382). New York, NY: ACM. URL: http://dl.acm.org/citation.cfm?id=1979445 [May 31, 2015].

Jacobson, Daniel. (2014). Top 10 lessons learned from the Netflix API. URL: http://www.slideshare.net/danieljacobson/top-10-lessons-learned-from-the-netflix-api-oscon-2014 [May 31, 2015].

Jiau, Hewijin Christine, & Yang, Feng-Pu. (2012). Facing up to the inequality of crowdsourced API documentation. SIGSOFT Software Engineering Notes, *37*(1), 1-9. URL: http://dl.acm.org/citation.cfm?id=2088892 [May 31, 2015].

Kontogiannis, Kostas, Lewis, Grace A., Smith, Dennis B., Litoiu, Marin, Muller, Hausi, Schuster, Stefan, & Stroulia, Eleni. (2007). The landscape of service-oriented systems: A research perspective. In *Proceedings of the International Workshop on Systems Development in SOA Environments (SDSOA '07)*. Washington, DC: IEEE Computer Society. URL: http://dx.doi.org/10.1109/SDSOA.2007.12 [May 31, 2015].

Mansilla, Neil. (2012). What are some examples of some successful, API-driven companies? *Quora*. URL: http://www.quora.com/What-are-some-examples-of-some-successful-API-driven-companies [May 31, 2015].

Pree, W. (1991). *Object-oriented versus conventional construction of user interface prototyping tools* [doctoral thesis]. Linz, AT: University of Linz. URL: http://www.softwareresearch.net/fileadmin/src/docs/publications/C002.pdf [May 31, 2015].

Ruecker, Stan, Radzikowska, Milena, & Sinclair, Stéfan. (2011). *Visual interface design for digital cultural heritage: A guide to rich-prospect browsing.* Farnham, UK: Ashgate Publishing. URL: http://www.ashgate.com/isbn/9781409404224 [May 31, 2015].

Tung, Sho-Huan Simon. (1992). Merging interactive, modular, and object-oriented programming [PhD dissertation]. Bloomington, IN: Indiana University, Computer Science. URL: http://www.cs.indiana.edu/ftp/techreports/TR349.pdf [May 31, 2015].

Watson, Robert, Stamnes, Mark, Jeannot-Schroeder, Jacob, & Spyridakis, Jan H. (2013). API documentation and software community values: a survey of open-source API documentation. In Proceedings of the 31st ACM international conference on Design of communication (SIGDOC '13) (pp. 165-174). New York, NY: ACM. URL: http://dl.acm.org/citation.cfm?id=2507076 [May 31, 2015].