# RiTa: Creativity Support for Computational Literature

**Daniel C. Howe**
Brown University
Computer Science Dept.
dhowe@cs.brown.edu

**ABSTRACT**
The RiTa Toolkit for Computation Literature is a suite of open-source components, tutorials, and examples, providing support for a range of tasks related to the practice of creative writing in programmable media. Designed both as a toolkit for practicing writers and as an end-to-end solution for digital writing courses (the focus of this paper), RiTa covers a range of computational tasks related to literary practice, including text analysis, generation, display and animation, text-to-speech, text-mining, and access to external resources such as WordNet. In courses taught at Brown University, students from a wide range of backgrounds (creative writers, digital artists, media theorists, linguists and programmers, etc.) have been able to quickly achieve facility with the RiTa components, to gain an understanding of core language processing tasks, and to quickly progress on to their own creative language projects.

**Author Keywords**
Computer education, Computational literature, Digital writing, Software libraries, Creativity support tools

**ACM Classification Keywords**
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

**General Terms**
Design, Human Factors, Languages

**INTRODUCTION**
While significant research in the Creativity Support community has focused on tools to aid creative practice, there has been less work addressing artists and designers interested in directly employing procedural methods and thinking [6,10]. Areas where such practice has received increasing interest in recent years include procedural product design, generative architecture, algorithmic musical composition, and computational literature (the focus of this paper), to name just a few. Teachers of introductory art and design courses that leverage computational techniques are often faced with the challenge of setting up practical programming tools for student assignments and projects. In these new and rapidly evolving fields, this task can be a difficult one. Students often enter courses with vastly different backgrounds and skill sets, and their creative projects tend to integrate a variety of programming tasks. A typical approach to this problem has been to employ multiple programming environments, with each providing support for some specific task of interest. For example, an introductory course in computational (or 'digital') literature might use Perl for text parsing and web-crawling, Apple's built-in 'talk' facility for text-to-speech, Flash for text display and animation, Max/MSP for audio support, and one of several research-oriented natural language packages for statistical analysis. By relying on the built-in features of these languages and platforms, the instructor can avoid developing a software infrastructure on their own.

One of several unfortunate consequences of this strategy is that significant time must be devoted to teaching the specifics of each new environment. This increases the delay before students are able to move on to more substantive topics, whether related to software engineering, natural language processing, or the creative practice of computational literature itself. Further, students cannot build on previously learned material in subsequent assignments. This lack of scaffolding is especially problematic when student projects tend to span a variety of 'core' tasks and thus require multiple environments to be bridged in a final project, often a formidable task. For example, a somewhat typical student project that involves extracting text from the web, altering it in some way, and visually displaying the results, accompanied by text-to-speech, might use most or even all of the environments mentioned above. It seems clear that with such shortcomings, a fresh approach is warranted.

In this paper we present a novel, creativity-oriented approach that directly addresses the above challenges and provides a flexible means for organizing the practical component of an introductory computational literature course. We describe RiTa, a creativity support library developed by the author in conjunction with a series of courses taught at Brown University from 2007-2009. RiTa is implemented in Java, optionally integrates with the Processing language environment, and runs on common platforms including Windows, OS X, Linux, and Unix. It is freely available under an open-source Creative Commons license at http://www.rednoise.org/rita/.


*C&C'09,* October 26–30, 2009, Berkeley, California, USA.

## THE PROGRAMMING ENVIRONMENT

An important first step in designing creativity supporting software for computational arts practice is the choice of an appropriate programming environment[1] [1]. A number of considerations, discussed elsewhere [5], influenced our choice of Processing [8] (and Java) for the context of digital literature. First, it was important that the environment provide a relatively shallow learning curve, so that novice programmers could receive immediate rewards for their efforts. Second, it should support rapid prototyping and short develop/test cycles. Third, it should be widely used so that questions, examples, and projects can be easily located on the web. Fourth, it should facilitate both structured (object-oriented) and ad-hoc programming styles. Fifth, it should provide end-to-end support for tasks that, though perhaps not central to the practice of computational literature, are often necessary for fully realizing a work (these include simple, yet robust access to sound, network, and graphics libraries.) Sixth, all library functions should run in (perceptual) real-time; there should be no need for offline processing. Seventh, student programs should be easily publishable and include source code to facilitate knowledge-sharing. Finally, all programs should be executable in a web-browser environment to eliminate any dependencies on hardware, operating system, and configuration, which can waste valuable time in a workshop setting.

## DESIGN CRITERIA

Once a base environment in which to implement the toolkit was selected, we identified several criteria which we felt would be important in the design and implementation of RiTa, somewhat generally following the methodologies laid out in earlier work [3,9].

*Ease-of-Use.* The primary purpose of the toolkit was to allow students to effectively implement their own creative language works; the more time students spent learning the toolkit, the less useful it would be.

*Consistency*. The toolkit should use consistent naming, syntax, functions, and design patterns. Therefore if one object was created via the traditional call to 'new()', another should not be created via a static or factory creation pattern.

*Extensibility*. The toolkit should easily accommodate new component implementations, whether for replication of existing functionality (in exercises or assignments), or for the addition of new (for the needs of specific projects).

*Documentation*. The toolkit, its components, and its implementation should be carefully and thoroughly documented and updated. All naming conventions should be carefully chosen and consistently used.

*Design-By-Interface.* Even when exposed in basic usage, all core objects should be implementations of interfaces and typed internally as such, thus allowing students to build and easily test their own implementations in exercises or assignments.

*Small/Light.* To enable download and execution in web browsers, the library should be as efficient as possible in its use of resources, most importantly browser memory. The most recent core version of RiTa (not including the WordNet database or the TTS voices] was ~2 MB.

*Modularity*. The interaction between different components of the toolkit should be minimal via well-defined interfaces. In particular, it should be possible to complete individual projects using small parts of the toolkit without concern for how they interact with the rest. This allows students to learn the library incrementally over one or more semesters.

*Share-ability*. Students programs should be easily exportable as web applets, including the generation of HTML pages and web-browsable source code.

*Performance*. Library functions should be fast enough that students can use the toolkit for interactive projects with functions returning in 'perceptual' real-time.

*Transparency*. The library and its underlying support APIs (Java, Processing, etc) should all be freely available with easily browsable and well-documented source code.

*Platform Agnostic*. The library should contain no operating system-specific behavior, allowing it be used in all major browsers and on all common platforms.

## CORE OBJECTS

The RiTa toolkit is implemented as a Java library comprised of seven independent packages. The core object collection is comprised of approximately 20 classes within the rita.* package, all of which follow similar naming and usage conventions. The rest of the packages provide support for these core objects, but are not directly accessed under typical usage. Each core object (described briefly below) defines the basic properties, methods and support structures for a specific task.

*RiText*: The basic utility object for strings of text and associated features. Contains a variety of utility methods for typography and display, animation, text-to-speech, and audio playback.

*RiAnalyzer*: Analyzes phrases, annotating each contained word, and the phrase itself, with a range of (customizable) feature data. Default features include word-boundaries, part-of-speech, stresses, syllables, and phonemes.

*RiMarkov*: Performs analysis and text generation via Markov chains (aka *n*-grams) with options to process single

[1] 'Environment' in this context refers to the combination of programming language, libraries, development support (e.g., IDEs), and associated tools for writing, compiling, debugging, running, and publishing programs.

characters, words, sentences, and arbitrary regular expressions.

*RiGrammar*: Implementation of a probabilistic context-free grammar (with specific literary extensions) to perform text generation from user-specified grammars.

*RiLexicon*: The built-in, user-customizable lexicon equipped with implementations of a variety of matching algorithms (min-edit distance, soundex, anagrams, alliteration, rhymes, looks-like, etc.) based on combinations of letters, syllables, pos, and phonemes.

*RiTokenizer*: A simple tokenizer for word and sentence boundaries with regular expression support for customization.

*RiStemmer*: A simple stemmer (based on the Porter algorithm) for extracting roots from words by removing prefixes and suffixes.

*RiPluralizer*: A simple rule-based pluralizer for nouns. Uses a combination of letter-based pluralization rules and a lookup table of exceptions for irregular nouns, e.g., 'appendix' → 'appendices'.

*RiSearcher*: A utility object for obtaining unigram, bigram, and weighted-bigram counts for words and phrases via online search engines.

*RiWordNet*: An intuitive interface to the WordNet ontology providing definitions, glosses, and a range of *-onyms* (hypernyms, hyponyms, synonyms, antonyms, meronyms, etc.) Can be transparently bundled into web-based, browser-executable programs.

*RiChunker*: A simple and lightweight implementation of a phrase-chunker for non-recursive syntactic elements (e.g., noun-phrases, verb-phrases, etc.)

*RiKWICker*: An implementation of a simple KeyWord-In-Context (KWIC) model for efficient indexing and lookup of words-in-phrases within documents.

*RiPosTagger:* A light-weight transformation-based part-of-speech tagger based on an optimized version of the Brill algorithm.

*RiSpeech*: Provides basic cross-platform text-to-speech facilities with control over a range of parameters including voice-selection, pitch, speed, rate, etc.

*RiSample*: Simple library-agnostic audio support for playback of *wav* and *aiff* samples and server-based streaming of *mp3*s.

*RiHtmlParser*: Provides various utility functions for fetching and parsing text data from the web using either the Document-Object-Model (DOM) or regular expressions.

*RiTextBehavior*: An extensible set of text-behaviors including a variety of interpolation algorithms for moves, fades, rotates, color-changes, etc.

*RiTaServer:* A client/server-based runtime for core objects that may have expensive initialization routines (e.g. building a large *n*-gram model,) or those that may benefit from persistent caching (e.g., the *RiSearcher* object described above).

Further detail on these objects is available in the online documentation at http://rednoise.org/rita/documentation/.

## DOCUMENTATION

RiTa is accompanied by extensive documentation that explains the functionality provided by the toolkit and describes how to use and extend it. In addition to descriptions, examples, tutorial and a comprehensive reference, the Project Gallery provides students with access to a wide range of existing projects (implemented with RiTa) by other students and artists, all with linked source code. Students can access this archive either for inspiration on projects or for assistance in addressing particular issues. The projects featured demonstrate proper documentation strategies, a particularly important element for those working with rapidly evolving technologies. Finally, students may, with instructor approval, add their own projects to the gallery, a goal which has inspired students and encouraged participation in the larger community of practicing digital artists.


**Fig. 1. An interactive student work created with RiTa**

## IN THE CLASSROOM

We used RiTa as a basis for the assignments, discussions and student projects in multiple iterations of 'Electronic Writing' (LR0021) and 'Programming for Digital Arts & Literature' (CSCI1950) at Brown University. These courses were open to both undergrads and graduates and focused on the analysis and creation of works of computational literature, with the former sponsored by the Literary Arts Department and the latter sponsored jointly by the Computer Science Department and the Rhode Island School of Design's Digital+Media program. Students' backgrounds were highly varied with a diverse mix of creative writers, plastic and digital artists, computer scientists, and digital media theorists. Technical backgrounds varied from programmers with many years of experience to those with

zero. In fact, negotiating this diversity of experience in lectures and assignments was one of the primary challenges of the course, one that would have been difficult to overcome without a unified toolset for all students.

A useful property of the RiTa toolkit is that it can easily be used to create student assignments of varying difficulty and scope. In very simple exercises, we have students experiment with a single RiTa object, attempting to generate interesting literary outputs by adjusting its properties and/or supplying new inputs. Students are then asked to reflect on the potential of the technique represented by the object, (e.g., context-free grammars and the RiGrammar object) and to identify any limits that they encounter. The variety of existing objects provides a range of opportunities for creating such simple assignments. As students become more familiar with relevant concepts, they can be asked to make minor changes or extensions to an existing module, perhaps addressing the limiting elements they had previously experienced. A more challenging task is to re-implement some or all of the object's interface themselves, demonstrating not only that the student understands the important intellectual concepts, but that they can also implement them efficiently in code. Finally, students are asked to employ the object in question (their own implementation if they have done one) in a creative project of their own design. Here, RiTa provides some useful starting points, including the examples and project gallery described above. More importantly, student projects are conceived and developed with extensive feedback (largely in workshop-style critiques) from both class members and the instructor. This iterative process serves to ensure that projects are not only of sufficient technical and intellectual merit, but also can be feasibly implemented within the specified time frame (generally 2-6 weeks).

## EVALUATION

To assess the degree to which the RiTa tools enhanced students' learning experiences and supported creativity, we performed a pilot evaluation with students in the Spring '09 iteration of the PDAL course. We collected data using a combination of pre and post surveys (~25 questions each), 2 programming quizzes, analysis of student projects, informal student feedback, and qualitative observations. Our goals in this pilot evaluation were three-fold. First, we hoped to measure changes in students' attitudes toward programming after exposure to the tools. Second, we wished to evaluate improvements in their programming ability. Third, we hoped to gain some initial insight concerning the degree to which these tools supported students' creativity in the context of digital art practice.

### Attitudes Toward Programming

In order to measure participants' general attitudes toward programming we developed four questions, each of which were asked in both the pre and post surveys. The first question addressed students' general sense of confidence in their programming and showed significant increases between pre and post-test scores, according to a paired-samples *t*-test, $t(11) = 3.677$, $p<.001$, $d=1.109$. The 2nd question addressed students' assessment of their programming ability and also showed significant increases, with $t(14) = 2.687$, $p<.01$, $d=0.718$. The 3rd question addressed the frequency with which students could "express their creative ideas" via programming, and also showed a significant increase, $t(12)=2.5606$, $p<.001$, $d=.739$**.** The 4th question addressed students' feelings about programming and, while not statistically significant, did show a small increase between pre and post-test scores. While it would be premature to assert any causal relationship based on these findings, they do suggest a positive impact on students' attitudes toward programming, regardless of gender, department, school, or previous technical background.

### Knowledge and Self-efficacy

To measure students confidence, knowledge and self-efficacy in a more granular fashion, eight additional questions were asked regarding knowledge and confidence in specific programming constructs (from variables, loops and conditionals, to functions, object-orientation and the Java language). On all 8 of these dimensions, paired-samples *t*-tests indicated significant differences between pre and post surveys. While it is unclear that such differences would not have resulted under other circumstances, it does suggest that both confidence and perceived knowledge were increased over the semester-long exposure to the tools and techniques.

### Programming Ability

To compare students' self-assessments with their skills, programming ability was also tested via a short programming quiz administered before and after the course. The style of the problem, generally referred to as a Parsons' problem [7], presents each student with a paired superset of the lines of code required to solve a programming task.

```
return result;
return word;

String result = "";
String result;

if (word.charAt(i) == 'x')
if (word.charAt(i) != 'x')

for (int i = 0; i < word.length(); i++)
for (int i = 0; i < word.length;  i++)

result = result + word.charAt(i);
result = word.charAt(i);

String removeAllXs(String word)
String removeAllXs(word)
```

**Fig. 2. An example Parsons' problem (from the pre-test)**

The student's task is to select the correct line of code from each pair, and then place the selected lines in the correct

order to define a (Java) method that accomplishes the specified task. An example Parsons' problem (from our pre-test), is presented in Fig. 2 above.

Table I. Parsons' problem results

| *total-score (scale 0 – 9, n=7)* | *% of students in pre-test* | *% of students in post-test* |
|---|---|---|
| mostly incorrect (0- 4.5) | 0.71 | 0 |
| partially correct (5–8.5) | 0.29 | 0.57 |
| fully correct (9) | 0 | 0.43 |

Although Parsons' problems are by no means a perfect measure[2], the results of the programming quiz, as presented in Table I, show a significant gain in students' ability to both read and write code.

### Student Feedback

Though largely anecdotal, the student feedback collected thus far supports our sense that RiTa provides a creativity-enhancing experience for a majority of students. While some noted their appreciation of the fact that they could run all programs on their home computers or laptops (rather than specifically designated lab computers), others appreciated the ability to quickly engage in creative and personally motivating projects from the outset. Still others noted the ability to easily share work (and code) on the web as particularly advantageous to their experience. All but the least experienced students appeared to find the level of detail in the documentation adequate for learning to use the toolkit and appreciated the ease with which they could combine different components to build new projects. Further, access to a large community of digital artists working with the same tools (Processing) appeared to greatly accelerate the development of students' own projects, even though few existing Processing works had focused on natural language.

Although more evaluation is needed, especially as regards the degree of creativity support provided, our experience thus far suggests that RiTa provides an effective and positive experience for the majority of students. Perhaps most convincing of all is the breadth and depth of student work represented in the RiTa project gallery[3] and how well-received such work has been in the larger digital arts community[4]. A second study, using a control population and focusing specifically on creativity support, is planned for Spring 2010.

[2] For a discussion of the strengths and weaknesses of this approach, see [2].

[3] Available at: http://www.rednoise.org/rita/rita_gallery.htm

[4] Several student projects have been featured in prestigious computational literary journals and gallery exhibitions.

## CONCLUSIONS

RiTa provides an extensible end-to-end programming framework for assignments, projects, and class demonstrations in computational literature classes. It is well-documented, easy to learn, and simple to use. While there exist several frameworks for teaching natural language and computational linguistics, e.g., NLTK [1], none of these focus on the specific requirements of an arts context. Similarly, existing tools and strategies for teaching digital art rarely address the domain of language. Further still, none have focused on addressing the needs of a highly diverse, inter-departmental student population. It is our hope that RiTa will not only provide a useful resource for instructors teaching in such a context, but may also provide some insight into how best to support creativity and procedural literacy in a range of potential domains extending well beyond the traditional borders of computer science departments.

## ACKNOWLEDGMENTS

Special thanks to the Brown University Computer Science and Literary Arts departments and to the RISD Digital+Media program for their generous sponsorship of this work. Also to John Cayley, Bill Seaman, and Braxton Soderman for their insight into the design and development of the toolkit and their feedback on this paper. Most of all to our students for their continual feedback on RiTa and the inspiring work they created with it.

## REFERENCES

1. Bird, S and Loper, E. 2002. NLTK: The Natural Language Toolkit. In Proceedings of the ACL Workshop on Effective Tools and Methodologies for Teaching Natural Language Processing and Computational Linguistics.
2. Denny, P., Luxton-Reilly, A., and Simon, B. 2008. Evaluating a new exam question: Parsons problems. In Proceeding of the Fourth international Workshop on Computing Education Research (Sydney, Australia, September 06 - 07, 2008). ICER '08. ACM, New York, NY, 113-124.
3. M. Flanagan, D. Howe, and H. Nissenbaum. 2008. Embodying Values in Design: Theory and Practice. In Information Technology and Moral Philosophy. Jeroen van den Hoven and John Weckert (eds.) Cambridge: Cambridge University Press, 2008.
4. Hartman, C. O. 1996. Virtual Muse: Experiments in Computer Poetry. Wesleyan University Press.
5. Howe, D. and Soderman, A. B. 2009. The Aesthetics of Generative Literature: Lessons from a Digital Writing Workshop. Hyperrhiz Journal of New Media Cultures, 2009 (forthcoming).

6. Mateas, M. 2005. Procedural literacy: Educating the new media practitioner. In On the Horizon: Special Issue on Future Strategies for Simulations, Games and Interactive Media in Educational and Learning Contexts.
7. Parsons, D. and Haden, P. 2006. Parsons' programming puzzles: a fun and effective learning tool for first programming courses. In Proc. of ACE 2006, Hobart, Australia, 157-163, January 16 – 19.
8. Reas, C. and Fry, B. 2007. Processing: A Programming Handbook for Visual Designers and Artists. MIT Press.
9. Resnick, M., Myers, B., Nakakoji, K., Shneiderman, B. Pausch, R., Selker, T., Eisenberg, M. 2005. Design Principles for Tools to Support Creative Thinking. NSF Workshop Report on Creativity Support Tools, Washington, DC, 12-14 June, 2005, 37-52.
10. Wing, J. M. 2006. Computational thinking, Communications of the ACM, v.49 n.3, March.