Writing-to-Learn-to-Program: Examining the Need for a New Genre in Programming Pedagogy

Dr. Bryan A. Jones, Mississippi State University

Bryan A. Jones (S'00–M'00) received the B.S.E.E. and M.S. degrees in electrical engineering from Rice University, Houston, TX, in 1995 and 2002, respectively, and the Ph.D. degree in electrical engineering from Clemson University, Clemson, SC, in 2005. He is currently an Assistant Professor with the Mississippi State University, Mississippi State, MS.

From 1996 to 2000, he was a Hardware Design Engineer with Compaq, where he specialized in board layout for high-availability redundant array of independent disks (RAID) controllers. His current research interests include robotics, real-time control-system implementation, rapid prototyping for real-time systems, and modeling and analysis of mechatronic systems

Dr. M. Jean Mohammadi-Aragh, Mississippi State University

Dr. M. Jean Mohammadi-Aragh is an assistant research professor with a joint appointment in the Bagley College of Engineering dean's office and the Department of Electrical and Computer Engineering at Mississippi State University. Through her role in the Hearin Engineering First-year Experiences (EFX) Program, she is assessing the college's current first-year engineering efforts, conducting rigorous engineering education research to improve first-year experiences, and promoting the adoption of evidence-based instructional practices. In addition to research in first year engineering, Dr. Mohammadi-Aragh investigates technology-supported classroom learning and using scientific visualization to improve understanding of complex phenomena. She earned her Ph.D. (2013) in Engineering Education from Virginia Tech, and both her M.S. (2004) and B.S. (2002) in Computer Engineering from Mississippi State. In 2013, Dr. Mohammadi-Aragh was honored as a promising new engineering education researcher when she was selected as an ASEE Educational Research and Methods Division Apprentice Faculty.

Ms. Amy K Barton, Mississippi State University

Amy Barton is Technical Writing Instructor in the Shackouls Technical Communication Program at Mississippi State University. In 2013, she was inducted into the Academy of Distinguished Teachers for the Bagley College of Engineering. She is an active member of the Southeastern Section of ASEE. Her research focuses on incorporating writing to learn strategies into courses across the curriculum.

Dr. Donna Reese, Mississippi State University

Donna Reese has served as head of the Computer Science and Engineering Department at Mississippi State University since 2010. Prior to that she served for six years as associate dean in the Bagley College of Engineering. Her research interests are in recruitment and retention of underrepresented groups in computing and engineering fields.

Hejia Pan, Mississippi State University

Hejia Pan received the B.S. degree in automation from the University of Science and Technology of China, Hefei, China, in 2007 and the Ph.D. degree from Mississippi State University in 2011. He is currently with Mississippi State University, Mississippi State, MS, USA. His research interests include engineering education, literate programming and modern control theory.

Writing-to-Learn-to-Program: Examining the Need for a New Genre in Programming Pedagogy

We are a nation driven by code, dependent on programs for our everyday lives. Vital aspects of our national security depend on our ability to withstand daily cyber intrusions into Department of Defense assets; likewise, our economic security rests on the ability of banks and stock exchanges to securely move millions of dollars electronically. Our nation's infrastructure depends on the smart grid and thousands of other process control systems that manage our water supplies, run our factories, and operate our cars. The social lives of millions of Americans rely on social networking, smart phones and tablets, and a high-speed internet backbone. Serious, sometimes fatal bugs such as Toyota's $1.2 billion dollar penalty[1] for unintended acceleration[2], Shellshock[3,4], and Heartbleed[5] all demonstrate the need for proficient programmers.

Despite the constantly increasing need for qualified engineers with strong programming abilities, the difficulty of teaching introductory programming still stands as a barrier to many STEM disciplines. A multi-institutional, multi-national experiment conducted by the McCracken group[6] reported only a little over 20% of students were able to solve the types of programing problems expected by their instructors. It is essential to properly prepare the people who power our security, our economy, and our lives.

In this paper, we present a new genre in computer science education that is aimed at improving student learning and application of programming concepts. Based on the well-acknowledged effectiveness of the writing across the curriculum (WAC) and Writing-to-Learn (WTL) movements, this paper employs the intermingling of writing activities with coding, which has the potential to dramatically impact the programming learning process. We term this approach Writing-to-Learn-to-Program (WTLTP). Following our discussion of the WTLTP genre, we present technologies that support the genre and discuss our future efforts to investigate the effectiveness of WTLTP in the classroom environment. This paper contributes both to WTL literature by incorporating WTL principles into a new domain and to computer science education literature by proposing a new approach to introductory computer science courses (CS1).

1. From writing-to-learn (WTL) to writing-to-learn-to-program (WTLTP)

WTL strategies arose from the writing across the curriculum (WAC) movement, which can be traced back to the 19th century in the U.S. It describes programs that emphasize the connection between writing and learning, but the term also refers to the pedagogical theories that support this connection. In the following sections, the history and influence of WTL are discussed as the foundation for WTLTP.

1.1. History of WTL

David Russell's history[7] of the WAC movement traced the cultural changes in the U.S. that enabled the movement's growth. Until the 1960s, universities were focused on disciplinary rigor and purity, so writing instruction was considered the domain of the English department. However, social change made the college classroom increasingly diverse, underscoring the need to provide more comprehensive writing instruction to students of all backgrounds and disciplines. WTL strategies were useful because they prompt students to write for their own benefit, making course material more meaningful. WTL is based on the premise that students learn through the act of writing, particularly when the writing assignments are short, informal, and designed to promote reflection, analysis, synthesis, and deeper understanding of course material. Examples of such assignments include journals, commentary, and reflections.

WTL strategies also allow students to assert more control over the way they learn. Emig[8] explains this benefit: "One writes best as one learns best, at one's own pace." Butler and Winne[9] assert that individualized learning is critical to individual success: "the most effective learners and self-regulating." They define self-regulation as "a suite of powerful skills: setting goals for upgrading knowledge; deliberating about strategies to select those that balance progress toward goals against unwanted tasks; and, as steps are taken and the task evolves, monitoring the accumulating effects of their engagement." If students are to be active participants in their learning, writing assignments must prompt self-analysis and reflection. Emig describes the simple act of reading one's own writing as a valuable learning moment in which "information from the process is immediately and visibly available as that portion of the product already written." Reviewing a set of writings collected over time, then, creates an opportunity to extend the learning process. Both instructors and students benefit from the act of collecting artifacts because they represent the changes and growth that accompany learning. When integrated in a purposeful way appropriate to a given discipline, WTL deepens student understanding, improves student engagement, increases retention, and makes students active participants in the learning process[10,11].

1.2 WTL and computational thinking

To understand the value of WTL strategies to computer science education, it is useful to examine the thinking processes involved in learning to program and the problems novice programmers tend to encounter. Davies[12] defines "computational thinking" as a complex type of problem solving that requires creativity as well as "elegance and precision." According to Soloway[13], the problem solving process is inextricably linked with the act of communicating understanding: "learning to program amounts to learning how to construct mechanisms and how to construct explanations." George[14] distinguishes computer science learning from learning in "humanity-based subjects" as the difference between "declarative content," or facts, and content that involves "procedures, processes, algorithms and problem solving steps." To program, students must think computationally, which requires critical thinking, reflection, revision, and communication ability.

The challenge for many is that computational thinking takes place in the context of a new, complicated language. According to Davies, the challenge for students trying to learn a new way of thinking is the language of programming itself: "a frightening world of semicolons and curly braces whose secret meaning takes great pains to discover." With this type of distraction, Davies asserts that students cannot undertake the serious problem solving that lies behind the language. While programming does require a programming language, computational thinking can be supported by writing in a student's native language.

Research in computer science has demonstrated that students' ability to effectively explain the meanings of code "in plain English" is correlated with their ability to program. Murphy et al.[15] found that students who could not correctly explain sections of code also had difficulty developing their own code as the semester progressed. Corney et al.[16] found that this issue carried on into the data structures class. They hypothesize that developing students' ability to explain code in written English can improve their ability to develop code. Essentially, WTL principles can support students' development of computational thinking skills by allowing them to think in their native language, thereby reducing a major barrier for learning to program.

1.3 WTLTP: A new genre in computer science education

The difficulty of teaching and learning introductory programming concepts is well documented. As early as the mid 1980s, the work of Soloway[17] documented that only 14% of Yale's CS1 students could solve a simple programming problem correctly. This study has been repeated over the years with similar results. At our home institution, for example, 42% of the students who have completed a two-semester programming class cannot solve the following program: "In a programming language of your choice, prompt the user to enter integers one at a time, keeping a running sum of the integers. If -1 is entered, then exit and print the sum." Perkins and Martin describe the main deficit of novice programmers as "fragile knowledge…knowledge that is partial, hard to access, and often misused"[18]. According to Lahtinen, Ala-Mutka and Järvinen[19], while most programming students can comprehend "basic concepts," they struggle with "learning to apply them." There is a significant need to address the well-documented and persistent need to improve programming pedagogy.

To improve programming pedagogy, we seek to re-introduce Knuth's concept of literate programming in a significantly revised and improved manner. To quote Knuth, "Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."[20] The philosophy undergirding literate programming, succinctly stated, is that "you do not document programs (after the fact), but write documents that contain the programs."[21]

As educators, we believe that writing clarifies thinking, extends creative abilities, and enables effective communication. Our beliefs are backed by the aforementioned literature. Thus, our research goal is to explore how reflective writing intermingled with coding can enable students to more effectively develop requisite problem solving skills and learn how to program. We hypothesize that the interaction between writing and programming can significantly improve the development of a novice programmer's skills. Through writing, programmers can visualize their learning, allowing them to view the coding process as a series of logical, interrelated steps that connect an overall goal to all the potential ways of reaching it. In a broader context, this visualization could have far-reaching implications for professional programming. This fundamental shift in the programmer's thinking requires a change in the programming process, particularly the process of learning to program.

We therefore propose a radical redesign of how writing and programming can and should interact, by making writing an essential component of programming instruction and the coding process as a whole. Specifically, embedding WTL strategies into the coding process will transform the way students learn to program. While students often view writing as separate from the practical demands of their professions, WTL strategies, when integrated purposefully, blend writing and learning in a way that can support development in any discipline and particularly in the context of programming, WTLTP leverages WTL strategies and intermingles writing with coding to support learner processes in an effort to improve programming pedagogy.

2. Writing-to-learn-to-program (WTLTP)

WTLTP represents a new genre built on the ideals of literate programming, which has rarely been incorporated into programming pedagogy. The purpose of WTLTP (and of literate programming) is not simply to add comments to code, but to transform a student's thinking process, by making writing an integral step in every aspect of the programming endeavor. In addition to researching changes in student thinking and learning, we will explore the design of software and technologies that support WTLTP and programmer development in relevant learning environments.

2.1 WTLTP in the classroom

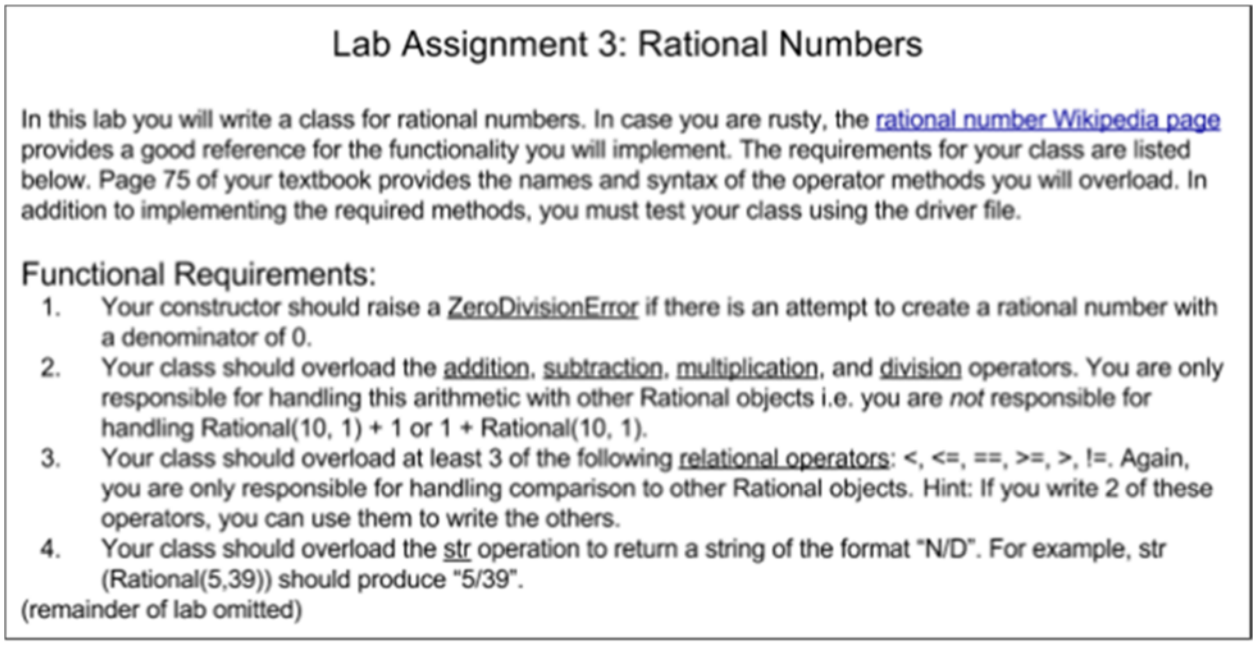
In a typical programming class, a lab assignment provides a problem specification; students are then graded on the correctness of their implementation. The bulk of instruction, therefore, focuses on teaching students to transform a problem into a specific implementation through a design-build-test cycle in which the design is accompanied by written artifacts such as a flowchart or pseudocode. The build process might also be accompanied by written artifacts (explain in English, for example), while the test process relies solely on the students' expertise rather than taking any written form. The design-to-build process is typically a linear, one-time process—students are expected to iterate the build-test cycle based on their own abilities. Moreover, the written artifacts remain isolated from the program; while the code (and, implicitly, the design) may evolve during the build-test cycle, the written artifacts remain unchanged. This process poorly supports the student's actual thinking process.

Therefore, we propose a method that matches and supports the ways students think about and solve programming problems, by intermingling writing throughout the design-test-build cycle. The WTLTP program is a document, which can and must contain all writing artifacts produced throughout the process. These artifacts prompt students to describe their thinking processes and reflect on their learning throughout the cycle. Specifically, the process begins as the students craft a description of their design, capturing the overall flow of the program. Next, students write reflectively, considering how this design should be captured in specific coding constructs (loops, functions, etc.). The build step begins with students evaluating which specific statements to use (which is the best library function to call, or how to index an array), then coding that particular portion of the program. The coding informs the overall design; perhaps the constraints of the language as the implementation proceeds will change the design, prompting students to rewrite and rethink that portion of the document. Next, students execute the program, then record and reflect on the observations captured by this experiment: did the program follow the mental model the student had formed of the program? If not, how was that model incorrect? What then should be changed? If these written reflections change the design or build aspects, those can be updated as a part of making changes to the code.

As a concrete example, consider a typical programming assignment taken from CSE 1384, Intermediate Computer Programming, offered at our home institution. In this lab, students write a class to support basic operations on rational numbers, which are represented as two integers N/D (Figure 1). A set of requirements is given, and students must then solve the problem with little guidance on the thinking process that should accompany the problem. In contrast, the screenshots in Figures 2–5 illustrate the integration of writing throughout the process of crafting a solution to this problem. In a beginning lab, the literate programming paradigm carefully guides students through the writing and thinking process; later labs have less guidance, requiring more creative, independent writing and thinking.

2.2 Existing software supporting WTLTP processes

Knuth's original literate programming system, consisting of the WEB (for the Pascal language) and CWEB (for C) applications, provide this needed ability, albeit with several significant drawbacks[22]. First, CWEB's input is not source code, but a document containing fragments of code mixed with troff/nroff or TeX/LaTeX and CWEB markup, requiring a steep learning curve and resulting in difficult-to-read documents for the uninitiated. CWEB then transforms this input into source code, stripping out much of the markup and formatting and rearranging the order, which produces source code that cannot be understood apart from laboriously referring to the CWEB document it came from. This makes use of a traditional IDE, along with the many tools it offers (debuggers, syntax highlighting, automatic refactoring, version control, static analysis, etc.), difficult if not impossible. Second, the typeset output of CWEB (typically a PDF document produced by compiling the literate programming input) is likewise difficult to edit. It cannot be directly edited; instead, the TeX/LaTeX fragment that produced it must be located by hand in the source literate program, then edited, then recompiled to a PDF document to check that the edits produced the desired typeset result. Although SyncTeX provides a mapping from a PDF paragraph to the underlying TeX/LaTeX that generated it[23], CWEB does not extend this mapping to



Lab Assignment 3: Rational Numbers

In this lab you will write a class for rational numbers. In case you are rusty, the rational number Wikipedia page provides a good reference for the functionality you will implement. The requirements for your class are listed below. Page 75 of your textbook provides the names and syntax of the operator methods you will overload. In addition to implementing the required methods, you must test your class using the driver file.

Functional Requirements:

1. Your constructor should raise a ZeroDivisionError if there is an attempt to create a rational number with a denominator of 0.
2. Your class should overload the addition, subtraction, multiplication, and division operators. You are only responsible for handling this arithmetic with other Rational objects i.e. you are *not* responsible for handling Rational(10, 1) + 1 or 1 + Rational(10, 1).
3. Your class should overload at least 3 of the following relational operators: <, <=, ==, >=, >, !=. Again, you are only responsible for handling comparison to other Rational objects. Hint: If you write 2 of these operators, you can use them to write the others.
4. Your class should overload the str operation to return a string of the format "N/D". For example, str (Rational(5,39)) should produce "5/39".

(remainder of lab omitted)

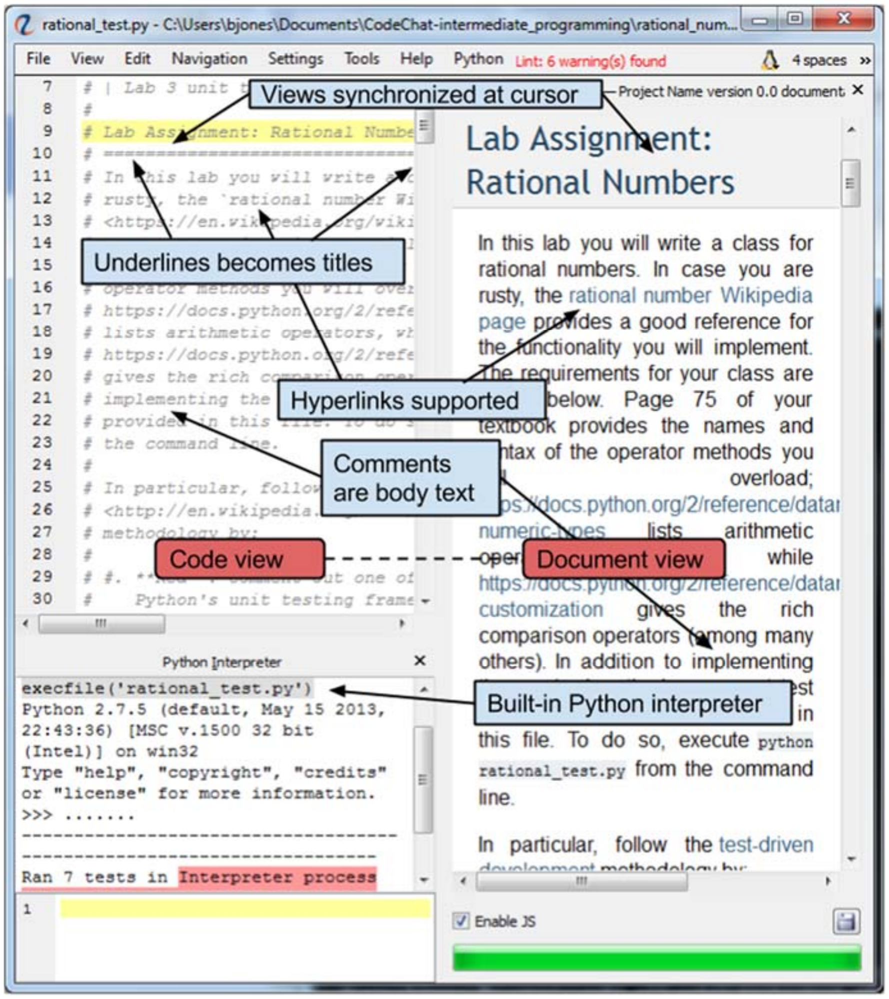
Figure 1. A CSE 1384 Intermediate Computer Programming Lab typifies the traditional approach, in which document and program exist in strictly separated areas.

the literate programming source. In addition, though several excellent GUIs embed this synchronization in a TeX/LaTeX environment (TeXWorks, for example, supports Windows/Linux/Mac), none provide this for CWEB, imposing a high cost on editing the typeset output.

Although many other literate programming packages exist as reviewed by Schulte[24], Pieterse, V. et. al.[25], most share these same weaknesses: they take a literate program as input and produce source code as output, creating relatively difficult-to-read code that cannot be directly edited because it will be overwritten by the next document-to-code transformation. Likewise, these packages take a literate program as input and produce a typeset document as output, creating beautiful documents that are time-consuming to edit.[26]

While literate programming tools have not entered the mainstream, their variants have; documentation generators, such as Doxygen and Javadoc, boast a huge user base and produce vast numbers of web pages that document large libraries and applications, such as the Java API and the KDE window manager. These tools provide an excellent method for documenting the externals of a program—its application programming interface (API), typically. However, from a literate programming perspective, these tools lack the ability to explain the inner workings of a function or method, instead restricting themselves to documenting how to call a function but not why the function works the way it does. Therefore, these tools likewise lack the ability for students to explain in writing what they are doing and why they are doing it.

Literate programming techniques have been previously employed in the classroom, with most activity occurring in the 1990s. Hurst[27] primarily investigates the tools needed to grade homework submissions, reporting that "the general quality of student submission has risen as a result of using literate programming" but offering little quantitative data in support of this assertion. Childs, Dunn, and Lively[28] report "significant benefit from the use of literate programming," providing detailed analysis that substantiates this finding. However, students reported frustration using the required literate programming tools (Emacs, TeX, and WEB). Shum and Cook[29] employed a literate programming tool, reporting that students using the system wrote comments that described the algorithm used by the code; students using a traditional programming methodology did not write these comments. Unfortunately, students reported that debugging the code produced by the tool was very difficult. Again, while the literate programming approach demonstrably improves novices' programming ability, the barriers raised by the current set of tools make these benefits difficult to realize. These barriers make the ability students most need—the ability


rational_test.py - C:\Users\bjones\Documents\CodeChat-intermediate_programming\rational_num...
File View Edit Navigation Settings Tools Help Python Lint: 6 warning(s) found 4 spaces »
Views synchronized at cursor
Underlines becomes titles
Hyperlinks supported
Comments are body text
Code view
Document view
Built-in Python interpreter
Project Name version 0.0 document
Python Interpreter
Enable JS

```
# | Lab 3 unit t
#
# Lab Assignment: Rational Numbe
# ================================
# In this lab you will write a c
# rusty, the `rational number Wi
# <https://en.wikipedia.org/wiki

# operator methods you will over
# https://docs.python.org/2/refe
# lists arithmetic operators, wh
# https://docs.python.org/2/refe
# gives the rich comparison oper
# implementing the
# provided in this file. To do s
# the command line.
#
# In particular, follow
# <http://en.wikipedia.
# methodology by:
#
# #. **Red
#     Python's unit testing frame
```

```
execfile('rational_test.py')
Python 2.7.5 (default, May 15 2013,
22:43:36) [MSC v.1500 32 bit
(Intel)] on win32
Type "help", "copyright", "credits"
or "license" for more information.
>>> .......
-----------------------------------
-----------------------------------
Ran 7 tests in Interpreter process
```

Lab Assignment: Rational Numbers

In this lab you will write a class for rational numbers. In case you are rusty, the rational number Wikipedia page provides a good reference for the functionality you will implement. The requirements for your class are below. Page 75 of your textbook provides the names and ntax of the operator methods you overload; ps://docs.python.org/2/reference/datar numeric-types lists arithmetic oper while https://docs.python.org/2/reference/datar customization gives the rich comparison operators (among many others). In addition to implementing st in this file. To do so, execute `python rational_test.py` from the command line.

In particular, follow the test-driven development methodology by:

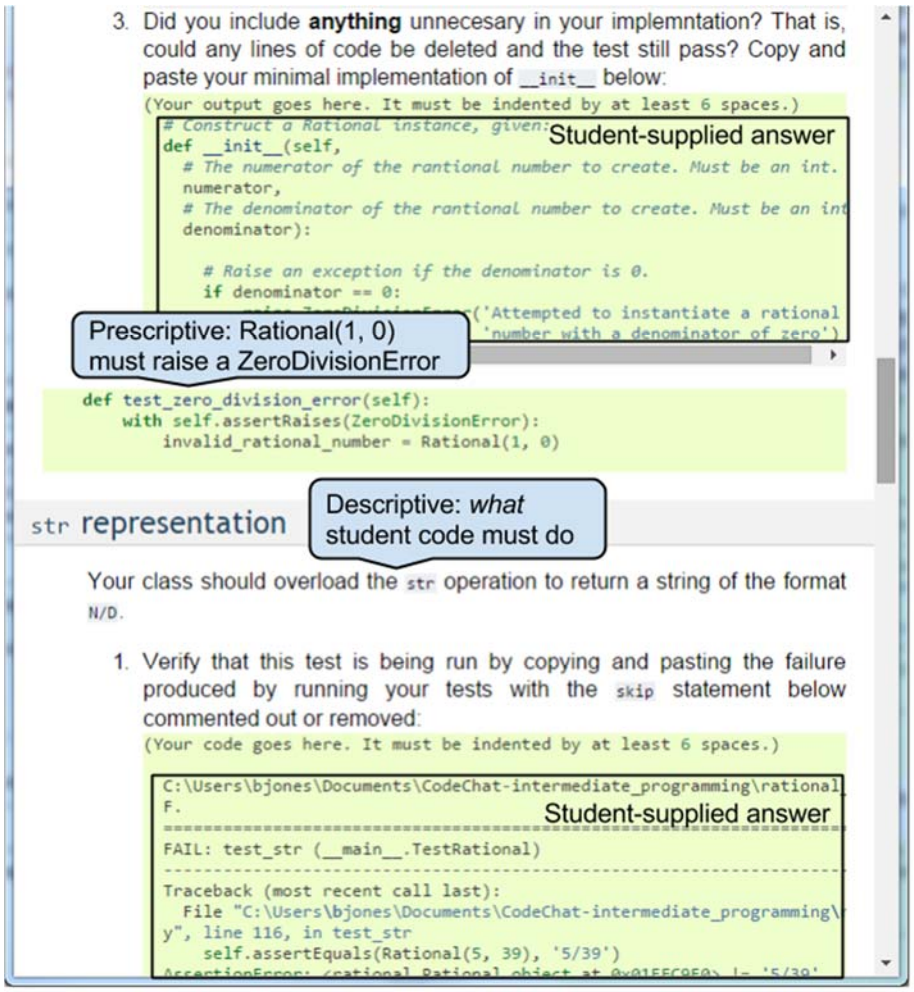
Figure 2. The CodeChat user interface, showing the code view on the left synchronized to the document view on the right.

to write before, during, and after they code—difficult if not impossible to learn.

2.3 CodeChat: Improved software to support WTLTP

With the goal of alleviating the weaknesses with current software described in section 2.2, we have developed a literate programming system named CodeChat. Figures 2–6 illustrate CodeChat's implementation in its present form.

The series of screenshots (except Figure 5) show a WTLTP lab on rational numbers for CSE 1384, Intermediate Programming, which is taught in the Python programming language. Figure 2 can be compared to the current lab shown in Figure 1. The WTLTP lab consists of a series of unit tests that guide a student through the implementation of a rational numbers class. The Python source code that the student will edit is presented on the left side of Figure 2, while on the right side, this source code has been transformed into a document (a web page), providing a visual reminder to students that a program is a document and should be treated as such. (The current


Student-supplied answer
Prescriptive: Rational(1, 0) must raise a ZeroDivisionError
Descriptive: *what* student code must do
Student-supplied answer

3. Did you include **anything** unnecesary in your implemntation? That is, could any lines of code be deleted and the test still pass? Copy and paste your minimal implementation of `__init__` below:

```
(Your output goes here. It must be indented by at least 6 spaces.)
# Construct a Rational instance, given:
def __init__(self,
  # The numerator of the rantional number to create. Must be an int.
  numerator,
  # The denominator of the rantional number to create. Must be an int
  denominator):

    # Raise an exception if the denominator is 0.
    if denominator == 0:
                          ('Attempted to instantiate a rational
                           'number with a denominator of zero')
```

```
def test_zero_division_error(self):
    with self.assertRaises(ZeroDivisionError):
        invalid_rational_number = Rational(1, 0)
```

`str` representation

Your class should overload the `str` operation to return a string of the format `N/D`.

1. Verify that this test is being run by copying and pasting the failure produced by running your tests with the `skip` statement below commented out or removed:

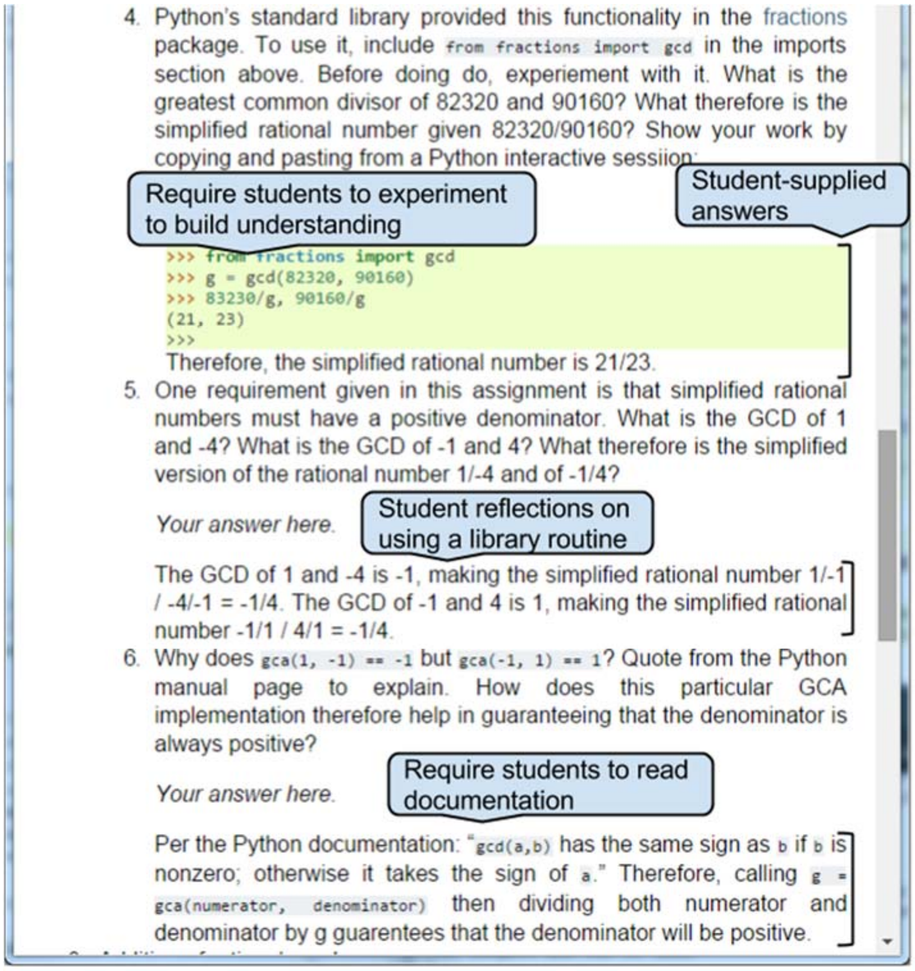
```
(Your code goes here. It must be indented by at least 6 spaces.)
C:\Users\bjones\Documents\CodeChat-intermediate_programming\rational
F.
======================================================================
FAIL: test_str (__main__.TestRational)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "C:\Users\bjones\Documents\CodeChat-intermediate_programming\
y", line 116, in test_str
    self.assertEquals(Rational(5, 39), '5/39')
```

Figure 3. The use of a program as a document which contains both descriptive and prescriptive portions.

implementation does not allow editing from the document view). The included GUI synchronizes between source code and web output automatically, so that movement or mouse clicks in the source code editor shows the corresponding HTML text and vice versa. CodeChat interprets the comments of the source code using reStructuredText (ReST), whose primary goal is to "define and implement a markup syntax … that is readable and simple, yet powerful enough for non-trivial use" as described on reStructuredText homepage.

Figure 3 demonstrates the use of a program as document that contains both descriptive and prescriptive (i.e., executable) portions. Here the unit test passes only if a divide-by-zero exception is raised by student code. This helps focus novice programmers on solving one problem at a time. It requires them to fill in a portion of the document with the results of their successful test (so that students will not simply skip this step to their own detriment) and the code they used, while simultaneously providing an opportunity to reflect.

Figure 4 shows a set of embedded reflection questions requiring student responses. These questions help a student think about how best to implement rational number simplification before


Require students to experiment to build understanding
Student-supplied answers
Student reflections on using a library routine
Require students to read documentation

4. Python's standard library provided this functionality in the fractions package. To use it, include `from fractions import gcd` in the imports section above. Before doing do, experiement with it. What is the greatest common divisor of 82320 and 90160? What therefore is the simplified rational number given 82320/90160? Show your work by copying and pasting from a Python interactive sessiion:

```
>>> from fractions import gcd
>>> g = gcd(82320, 90160)
>>> 83230/g, 90160/g
(21, 23)
>>>
```

Therefore, the simplified rational number is 21/23.

5. One requirement given in this assignment is that simplified rational numbers must have a positive denominator. What is the GCD of 1 and -4? What is the GCD of -1 and 4? What therefore is the simplified version of the rational number 1/-4 and of -1/4?

Your answer here.

The GCD of 1 and -4 is -1, making the simplified rational number 1/-1 / -4/-1 = -1/4. The GCD of -1 and 4 is 1, making the simplified rational number -1/1 / 4/1 = -1/4.

6. Why does `gca(1, -1) == -1` but `gca(-1, 1) == 1`? Quote from the Python manual page to explain. How does this particular GCA implementation therefore help in guaranteeing that the denominator is always positive?

Your answer here.

Per the Python documentation: "`gcd(a,b)` has the same sign as `b` if `b` is nonzero; otherwise it takes the sign of `a`." Therefore, calling `g = gca(numerator, denominator)` then dividing both numerator and denominator by g guarentees that the denominator will be positive.

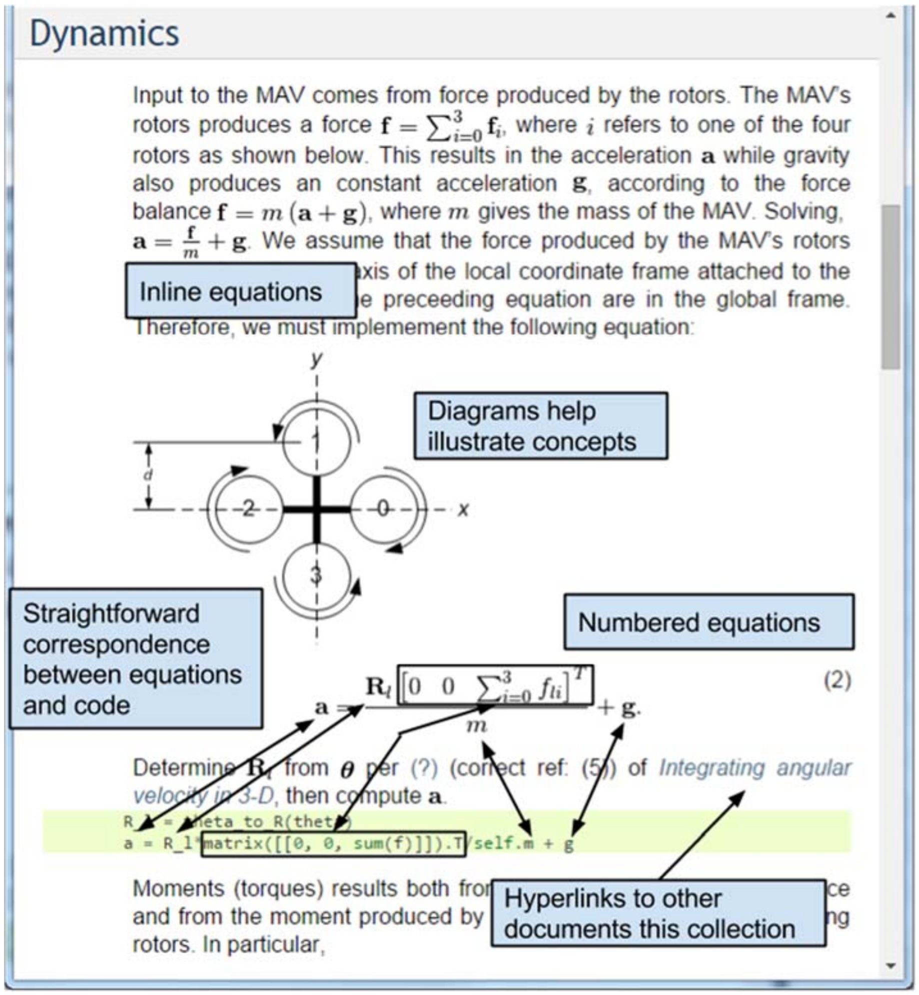
Figure 4. Embedded reflection questions require student responses.

writing the code and require them to gain a deeper understanding of the library routine (gcd) they will use. This helps students produce code that they understand, rather than code that "just sorta works" without deep understanding.

The program-as-document approach to literate programming in Figure 5 shows that intermingling images and equations with the code that implements them provides a natural, intuitive explanation of the operation of a quad-copter (a four-bladed rotary-wing aircraft). This approach allows students to think through the complex mathematics they must implement, or instructors to better convey the connection between theory and its implementation in a program.

3. Conclusions and future research direction

In this conceptual paper, we introduce a new genre in computer science pedagogy based on intermingling writing activities with coding. We have proposed a radical redesign of how writing and programming can and should interact. We contribute to WTL literature by incorporating WTL principles into a new domain. We build on Knuth's initial ideas for literate programming


Inline equations
y
d
1
2
0
3
x
Diagrams help illustrate concepts
Straightforward correspondence between equations and code
Numbered equations
Hyperlinks to other documents this collection

Dynamics

Input to the MAV comes from force produced by the rotors. The MAV's rotors produces a force $\mathbf{f} = \sum_{i=0}^{3} \mathbf{f}_i$, where i refers to one of the four rotors as shown below. This results in the acceleration $\mathbf{a}$ while gravity also produces an constant acceleration $\mathbf{g}$, according to the force balance $\mathbf{f} = m\left(\mathbf{a} + \mathbf{g}\right)$, where m gives the mass of the MAV. Solving, $\mathbf{a} = \frac{\mathbf{f}}{m} + \mathbf{g}$. We assume that the force produced by the MAV's rotors xis of the local coordinate frame attached to the e preceeding equation are in the global frame. Therefore, we must implemement the following equation:

$$\mathbf{a} = \frac{\mathbf{R}_l \boxed{\begin{bmatrix} 0 & 0 & \sum_{i=0}^{3} f_{li} \end{bmatrix}^T}}{m} + \mathbf{g}. \qquad (2)$$

Determine $\mathbf{R}_l$ from $\boldsymbol{\theta}$ per (?) (correct ref: (5)) of *Integrating angular velocity in 3-D*, then compute $\mathbf{a}$.

```
R_l = theta_to_R(theta)
a = R_l*matrix([[0, 0, sum(f)]]).T/self.m + g
```

Moments (torques) results both fro ce and from the moment produced by ng rotors. In particular,

Figure 5. Intermingling images and equations with code demonstrates the program-as-document approach to literate programming.

by proposing the integration of those ideas into technology supporting learning and into classroom learning environments. We have established the background and need for the new WTLTP genre.

This conceptual paper is the precursor to our upcoming research investigation into the use of technology to facilitate WTLTP in the technology-rich environment of an introductory programming course. The purpose of our future research direction is to thoroughly investigate how WTLTP can help students learn to program. We focus on understanding the impact of WTLTP instruction on students' programming development in comparison to students educated by traditional programming pedagogy. We also plan to investigate how WTLTP may impact students' development as writers. Finally, we have planned data collection that will offer insight into "best practices" for effectively integrating WTLTP in classrooms. All of our research is driven by the overarching research question: How can intermingled writing assignments affect the development of a novice's programming skills? Our research plan will allow us to consider 1) next steps for how technology can support intermingled WTL, 2) how instructors can support intermingled WTL, and 3) challenges to effective use of intermingled WTL in classroom contexts.

References

[1] Woodyard, C & Johnson, K. (2014, Mar. 20). Toyota to pay $1.2B to settle criminal probe. USA Today, Money section.

[2] Barr, M. (2013, Oct. 26). *An update on Toyota and unintended acceleration*. [web log]. Retrieved from http://embeddedgurus.com/barr-code/2013/10/an-update-on-toyota-and-unintended-acceleration/.

[3] National Institute of Standards and Technology National Vulnerability Database, Vulnerability Summary for CVE-2014-6271. (2104, Dec. 2). Retrieved from: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-6271 .

[4] National Institute of Standards and Technology National Vulnerability Database, Vulnerability Summary for CVE-2014-7169. (2014, Dec. 2). Retrieved from: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7169.

[5] National Institute of Standards and Technology National Vulnerability Database, Vulnerability Summary for CVE-2014-0160. (2014, Dec. 2). Retrieved from: https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0160.

[6] McKracken, M. et al. (December 2001) *A Multi-national, multi-institutional study of assessment of programming skills of first-year CS Students*, SIGCSE Bulletin, vol. 33, no. 4, pp. 125-180.

[7] Russell, D. R. (1991). *Writing in the academic disciplines*, 1870-1990: A curricular history. Carbondale, IL: Southern Illinois UP.

[8] Emig, J. (1977). *Writing as a mode of learning*. College Composition and Communication, 28, 122-128.

[9] Butler, D. & Winne, P. (1995). *Feedback and self-regulated learning: A theoretical synthesis. Review of Educational Research*, 65, 245-281.

[10] Paretti, M. C. (2011). *Theories of Language and Content Together: The Case for Interdisciplinarity*. Across the Disciplines, 8(3).

[11] Paretti, M. C. (2009). *When the Teacher is the Audience: Assignment Design and Assessment in the Absence of "Real" Readers,* in Engaging Audience: Writing in an Age of New Literacies, A. Gonzalez, E. Weiser, and B. Fehler, Editors. 2009, NCTE Press: Urbana, IL. p. 165-185.

[12] Davies, S. (2008). *The effects of emphasizing computational thinking in an introductory programming course.* 38th ASEE/IEEE Frontiers in Education Conference, T2C-3—T2C-8.

[13] Soloway, E. (1986). *Learning to program=learning to construct mechanisms and explanations.* Communications of the ACM, 29(9), 850-8.

[14] George, S. E. *Learning and the reflective journal in computer science.* (2001). Proc. 25th Australasian Computer Science Conference (ACSC2002), Melbourne, Australia, 77-86.

[15] Murphy, L., R. McCauley and S. Fitzgerald (March 2012). *'Explain in Plain English' Questions: Implications for Teaching,* SIGCSE '12, pp. 385-389.

[16] Corney, M. et al. (March 2014). *'Explain in Plain English' Questions Revisited: Data Structures Problems,* SIGCSE '14, pp. 591-596.

[17] Soloway, E. (November 1983) *Cognitive strategies and looping constructs: An empirical study*, Communications of the ACM, Vol, 26, No 11, pp. 853-860.

[18] Perkins, D. & Martin, F. (1985). *Fragile knowledge and neglected strategies in novice programmers*. National Institute of Education Report. Cambridge, MA. Retrieved from http://faculty.salisbury.edu/~xswang/ Research/Papers/ debugging/ED295618.pdf.

[19] Lahtinen, E., Ala-Mutka, K. & Järvinen, H. M. (2005). *A study of the difficulties of novice programmers*. ACM SIGCSE Bulletin, 37(3), 14-18.

[20] Knuth, D. (1992). Literate Programming (1984). Literate Programming. CSLI. p. 99.

[21] Skaller, M. J. in a Charming Python interview. Retrieved from http://gnosis.cx/publish/programming/ charming_python_8.html.

[22] Thimbleby, H. (1986). *Experiences of 'Literate Programming' using CWEB (a variant of Knuth's WEB)*. The Computer Journal, 29(3), 201-211.

[23] Laurens, J. (2008). *Direct and reverse synchronization with SyncTEX*. TUGboat, 29(3). Retrieved from: http://www.tug.org/TUGboat/tb29-3/tb93laurens.pdf.

[24] Schulte, E. (January 2012). "*A Multi-Language Computing Environment for Literate Programming and Reproducible Research*" 46(3). Journal of Statistical Software.

[25] Pieterse, V., Derrick G. K., & Boake, A. (2004). *A case for contemporary literate programming*, in SAICSIT, 75, 2-9, Stellenbosch, Western Cape, South Africa.

[26] PyLit, available at http://pylit.berlios.de/index.html, represents the only tool known to the authors which also stores the literate programming source in the source file, lacks a GUI to synchronize between typeset output and source code.

[27] Hurst, A. J. (1996). *Literate programming as an aid to marking student assignments*. Proceedings of the 1st Australasian conference on Computer Science education, 280-286.

[28] Childs, B., Dunn, D., & Lively, W. (1995). *Teaching CS/1 Courses in a Literate Manner*. Proceedings of the TeX Users Group Conference, St. Petersburg, Florida, July 24-28, Volume 16, No. 3, p. 300-309.

[29] Shum, S. & Cook, C. (1994). *Using Literate Programming to Teach Good Programming Practices*. Proceedings of the twenty-fifth SIGCSE symposium on Computer Science education, 66-70.