

PROMOTING ACTIVE LEARNING IN COMPUTER SCIENCE USING MICROLABS

A Thesis
by
PHILIP ROSS MEZNAR

Submitted to the Graduate School
at Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

May 2013
Department of Computer Science

PROMOTING ACTIVE LEARNING IN COMPUTER SCIENCE USING
MICROLABS

A Thesis
by
PHILIP ROSS MEZNAR
May 2013

APPROVED BY:

Barry L. Kurtz
Chairperson, Thesis Committee

James B. Fenwick Jr.
Member, Thesis Committee

Rahman Tashakkori
Member, Thesis Committee

James T. Wilkes
Chairperson, Department of Computer Science

Edelma D. Huntley
Dean, Research and Graduate Studies

Copyright by Philip Ross Meznar 2013
All Rights Reserved

Abstract

PROMOTING ACTIVE LEARNING IN COMPUTER SCIENCE USING MICROLABS

Philip Ross Meznar
B.A., Bryan College
M.A., Appalachian State University

Chairperson: Barry L. Kurtz

Computer science education continues to grow in importance as the technology industry becomes increasingly prevalent on a global scale. In order to remain competitive, computer science education must continue to increase both the quality and quantity of graduates. In efforts to achieve such ends, the Wags system has been designed and developed to be used in conjunction with the Microlab Learning Cycle, an educational process founded in constructivist learning theory. Through continual testing and refinement, the Microlab Learning Cycle and accompanying system have been able to produce measurable improvements in student understanding and retention of important computer science concepts, while providing an active-learning classroom environment that students enjoy and find valuable.

Table of Contents

Abstract	iv
Foreword	1
1 Introduction	2
1.1 Importance of Technological Education	2
1.2 Active Learning in Computer Science Education	3
2 Background and Related Work	6
2.1 Constructivist Learning Theory and Jean Piaget.....	6
2.2 The Learning Cycle	8
2.3 Visual Programming Languages.....	12
2.4 Automated Grading Systems	13
2.5 Use of Tablets in the Classroom	15
2.6 Conclusion.....	16
3 Pedagogical Issues and Goals	18
3.1 Implementing the Learning Cycle.....	18
3.2 Types of Microlabs	19
3.3 The Microlab Learning Cycle	23
3.4 Designing a Microlab Framework	26
3.4.1 Goals.....	26
3.4.2 Usability	27
3.4.3 Creating Microlabs	29
3.4.4 Incorporating Guidance in Microlabs.....	33

3.4.5 Classroom Management.....	36
4 Software Design and Implementation	39
4.1 Design and Implementation Decisions.....	39
4.2 Client Side Structure.....	42
4.3 Server Side Structure	46
4.4 Database Structure	49
5 Classroom Testing and Outcomes	52
5.1 Early Testing in Fall 2011	51
5.2 Testing in Spring 2012.....	53
5.3 Testing in Fall 2012.....	55
6 Summary, Conclusions, and Future Development.....	62
6.1 Summary of the Microlab System (Wags).....	62
6.2 Successes and Issues.....	63
6.3 Future Work.....	68
References	72
A Wags Manual	74
Vita	85

Foreword

The research detailed in this thesis has been formatted according to the style used in Appalachian State University's Computer Science department for thesis submissions. This format is based on the ACM standard for published papers, with additional style guidelines concerning margins, font size, boldface, and headings. References and in-text citations follow ACM guidelines unaltered.

Chapter 1

Introduction

1.1 The Importance of Technological Education

In his 2013 State of the Union address, President Barack Obama called for a greater emphasis on science, technology, engineering, and math education (STEM) [12]. President Obama identified that explicit demands for ever improving technology indicate implicit demands for ever improving technological education. Computer science is a cornerstone of the technology industry, thus computer science education is a cornerstone of technological education. Therefore, despite the successes of computer science education in America today, improvements and advancements must be constantly sought after with vigor and intensity; as to become complacent in education is to become stagnant, and stagnancy leads to failure on a global scale.

Improvement and advancement in education must be strictly defined. Simple allocation of funds, while helpful, is no guarantee of an improvement in quality instruction. Rather, science must be built upon science. Educational practices must be built upon secure foundations of education and development theory. Thus, a renewed emphasis on technological education demands a new emphasis on education theory and its relation to the

classrooms students occupy today. A balance must be found between ideological allegiance to a strict interpretation of an educational theory and the practical application of that theory into the classroom.

The traditional lecture format of education has proven itself an adequate technique for education across grade levels and disciplines. Adequacy, however, can and should be improved upon. Traditional lectures often require an attention span that is not commonly found in a culture that promotes instant gratification and constant interaction. Thus, Active Learning practices are being encouraged across the nation – practices that increase student participation in the classroom; practices that do not demand passive attention but create active interaction [14]. While such practices do much to improve education, the mantra “increase student participation” does little to equip instructors in a practical manner. At an inter-disciplinary level such a mantra may be all that is possible; “one size fits all” only works when that size is widely encompassing and generic.

1.2 Active Learning in Computer Science Education

Thus, narrowing the focus to computer science education allows for practices to be delineated that strike a better balance between ideology and practicality. The ideological foundation and the development, testing, and outcomes of one such practice are the focus of this thesis.

Constructivist learning theories are the basis for much of the active learning emphasis seen today. Examination of constructivist theory in the specific context of computer science education provides opportunity for significant improvement in computer science education

practices. Constructivist learning theory, at a fundamental level, is concerned with the creation of knowledge. The theory holds that knowledge does not originate within the instructor and is not somehow “transferred” or passed down to the student via lecture or reading. Rather, the theory states that knowledge originates with the student, and it is the role of the instructor to help students create knowledge for themselves. The instructor accomplishes this through two means: equipping the students with necessary foundational information and exposing the students to environments which require the knowledge the student is supposed to create. These environments are often problems that the student is asked to reason through and solve. Each problem should be designed to encourage student exploration and interaction with the concept at hand, allowing the student to examine the problem from multiple angles without penalty.

Constructivists hold that the primary duty of the instructor is to equip the students to teach themselves. Therefore, specific tools and frameworks become an integral part of the classroom as they allow students to interact with the problem domain and guide the student towards knowledge creation. Consequently, the research presented in this thesis concerns constructivist learning theory insofar as it informs the design and development of tools and frameworks that incorporate sound active learning practices into the computer science classroom.

The development and performance of one such tool and corresponding framework is the main presentation of this thesis. The tool is called Wags, and the framework is known as the Microlab Learning Cycle – a scheme based on constructivist learning theory and designed for computer science education. The name Wags began as an acronym for Web automated grading system. As Wags continued to develop using the term as an acronym fell

out of favor as the acronym put unwarranted emphasis on only a portion of the systems functionality. The name Wags is still used for historical consistency. Foundational to Wags is the idea of a Microlab – a short, five to ten minute exercise that students can accomplish during class time, which allows multiple attempts at a solution, and enables students to interact with the concepts they are learning first hand.

Chapter 2

Background and Related Work

2.1 Constructivist Learning Theory and Jean Piaget

The belief that incorporating Microlabs into computer science education will improve student understanding and retention is founded on an education theory known as Constructivism.

Constructivism promotes active learning and hands on approaches, as constructivism resonates with the belief of “the only way of ‘knowing’ a thing is to have made it,” as espoused by 18th century philosopher Giambattista Vico [16]. Even before Vico and in a different cultural setting, a Confucian proverb expounds a similar belief: “I hear and I forget, I see and I remember. I do and I understand.” The constructivist philosophy holds that the act of a student acquiring knowledge is primarily an act of creation, rather than an act of transference (of knowledge from teacher to student, for example). Therefore, education centers around equipping students with necessary foundational information, and then exposing them to an unknown environment and allowing the student to create knowledge first-hand.

Two of the first constructivists are John Dewey and Maria Montessori. Dewey applied constructivist theories in the early 20th century by creating an active learning

environment in his laboratory school [8]. Maria Montessori is the progenitor of Montessori schools, which are still common today and stress childhood independence and personal psychological development, principles consistent with the constructivist learning theory [1]. Both constructivists emphasized hands-on activities as the way to guide a student to creating their own knowledge, abandoning more traditional lecture formats.

Jean Piaget, another historic constructivist theorist, held that the way a person learns changes over time. Piaget uses the word schema to describe a mental framework that develops as children interact with the world around them. As the child develops new schema they must somehow relate this new knowledge to what they held previously. Piaget states that there are two ways this can occur. If the new schema is consistent with previous schemata, then the child will “assimilate” the new schema without conflict – this is often a process of reinforcing and expanding a previously held schema. However, if the new schema causes dissonance with previously held beliefs, or this new schema does not relate to previous schema, the child must “accommodate” the new schema by revising or replacing the previously held beliefs or by introducing a completely new one [2].

Piaget identified stages of learning that provide a useful framework for the development of interactive educational activities. This framework allows for the development of activities that aim to be more than merely hands-on – they are designed to help guide students towards more advanced stages of intellectual development for a given subject. Piaget listed four main stages of development: Sensorimotor, Preoperational, Concrete Operational, and Formal Operational. The Sensorimotor and Preoperational stages are found in young children. The Concrete Operational stage is typically found in children around the ages of 7 to 12, but can persist into adulthood, and is characterized by reliance on

physical aides to bring about assimilation or accommodation of new schema. Formal Operational is the highest level of intellectual development, allowing for abstract thought, reflective thinking, and understanding of complex concepts without relying on the physical aides found in the Concrete Operational stage [3].

Given these stages, the job of the educator becomes guiding students into a formal operational understanding of the current topic. Thus, education becomes a process of creating knowledge through assimilation and accommodation brought about through techniques appropriate for the students' current stage of development on a given subject. This process aims to produce a formal operational understanding of a new subject in the students, though it is not uncommon for a student to be in the concrete operational stage of development for a new concept. Therefore, an approach that guides students to actively assimilating and accommodating new schema, designed to lead from a concrete to formal understanding, is an invaluable tool in education.

2.2 The Learning Cycle

Such an approach was developed by Robert Karplus, theoretical physicist-turned-educator. As an educator, Karplus studied the theories of various psychologists subscribing particularly to Jean Piaget's work. Thus, Karplus began working on a practical way to incorporate Piaget's theories into the classroom. The end result was the Learning Cycle, a systematic approach to teaching that Karplus believed incorporated the foundations of education uncovered by Piaget and reflected in Constructivist Theory [5]. To achieve this, the Learning Cycle is split into three phases: Exploration, Invention, and Application.

The Exploration phase exposes students to a new concept in an environment that leaves them free to explore [6]. The instructor only provides minimal guidance in this phase as the students are given the task of gathering new information through their own actions. During this time students should come across problems that they cannot overcome using pre-existing schema. Thus, the student works to expand an existing schema or develop a new schema. If the student is successful in completing the current task, then an existing schema is reinforced. If not successful, students become aware of the lack in their current patterns of reasoning and become prepared to adopt new strategies.

The Invention phase capitalizes on the students' recognition that new knowledge is required to formalize the knowledge gained during the exploration phase. If the student had developed a correct schema during the exploration phase, the schema should be assimilated in the invention phase. If the student had applied an incorrect schema, the realization that they have to introduce new reasoning results in replacing whatever schema was forming during the exploration phase (Piaget's accommodation terminology applies here). The invention phase includes a more traditional introduction to the subject by the instructor. While the exploration phase usually employs some concrete aides to help students construct new schema, the invention phase introduces formal definitions to help transition students into a formal operational understanding of the subject.

The Application phase is designed to further solidify the accommodation or assimilation that occurred in the Invention phase by exposing the students to a wider range of problems based on the concept being taught. Furthermore, this phase allows the students to once again create their own knowledge rather than be dependent on the instruction during the Invention phase. This application of the schema expanded or developed in the invention

phase is critical in improving knowledge retention – the knowledge they were given in the Invention phase turns into understanding in the Application phase. Of course, the instructor can provide individualized feedback for any students still struggling with the concept.

An example concept can be useful to illustrate the Learning Cycle process. Karplus often demonstrated his cycle using the problem “Mr Tall and Mr Short,” and it will be used here as well. In the Exploration phase the instructor will give a quick introduction to the problem, namely that Mr Short is six paperclips tall, but when measured in buttons is only four buttons tall. Mr Tall, in comparison, is six buttons tall. The instructor will then ask the students to predict how tall Mr Tall will be in paperclips. Once the students have recorded their answers and accompanying explanations, the instructor will pass out some concrete aides in the form of paperclips and buttons and ask the students to verify the given data. Subsequently, the students will be asked to measure Mr Tall and check the real answer against what they had predicted. Students who correctly answered nine paper clips will have their existing schema reinforced, even if they do not have a formal operational understanding of why they were correct. Students who answered incorrectly will be forced to recognize that their existing schema did not produce the correct answer, producing cognitive dissonance that they will seek to resolve in the Invention phase. The most common incorrect answer is eight paper clips, an additive strategy explained by “There are two more paperclips than buttons” or “Mr. Tall is two buttons taller so he will be two paperclips taller.”

The Invention phase would have the instructor explain to the students concepts like proportional scaling and multiplicative thinking. That is, the teacher will explain to the students why the answer they found through measurement makes sense. This stage may involve the instructor initially using the concrete aides to express their point, but at the end of

the phase concrete aides are hopefully no longer necessary as the students have progressed into the formal operational stage. Thought experiments are often useful in this transition, as the teacher asks questions that expose flaws in common incorrect schema (such as additive reasoning). For example, the instructor may ask about a Mr Tiny who is only two paper clips high and ask how tall he would be in buttons, or about a Mr Giant who is twice as tall as Mr Tiny (twelve paper clips) and asking how many buttons tall Mr Giant would be.

The Application phase would then consist of students once again tackling problems without teacher assistance designed to expand upon and solidify the newly introduced schema. In this example, the teacher could have questions with different scales than paperclips and buttons, or photos enlarged to different sizes. Students who understood the schema in the Invention phase would have that schema reinforced as they discover more and more correct answers, while students who mistakenly believed they understood the schema will soon have the inadequacy of their understanding revealed.

The Learning Cycle has been used in many classrooms and has been a tested method of teaching for decades. Using the Learning Cycle in computer science curriculum has particularly significant potential due to the nature of the discipline, as schema for a specific problem are often formalized into an algorithm or a specific data structure. Incorporating the Learning Cycle into lecture can be challenging as it requires three successive phases that greatly benefit from being close to each other temporally; splitting the phases among class times would diminish the effect as each phase leads naturally into the next. Thus, the students should be able to complete Exploration and Application exercises quickly, and must be given feedback as to whether or not their attempts were successful in order to produce reinforcement or dissonance where necessary. Visual Programming Languages (VPLs)

allow students to explore concepts quickly, while Automated Grading Systems (AGSs) can provide instantaneous feedback for multiple students simultaneously. Therefore, examining VPLs and AGSs can make incorporation of the Learning Cycle a feasible and beneficial inclusion in computer science education.

2.3 Visual Programming Languages

Visual Programming Languages (VPLs) are characterized by allowing students to construct programs through a graphical manipulation of program elements, in contrast to the traditional approach of typing the program. The two most commonly used examples of a VPL are Alice and Scratch.

These languages are often used in an educational setting, as a strength of a VPL is quick

exploration of a concept without relying on previous programming knowledge concerning syntax or the contents of standard libraries [11]. Both programming languages use a drag-and-drop interface to allow quick creation of programs or animations. Figure 4.1 shows an example of Scratch code, where a loop is constructed out of various pieces of code, displaying the graphical nature of program construction.



Figure 4.1 – Some Scratch code

Source: Google Image Search

Visual Programming Languages allow students to develop and test programs quickly. However, VPLs inherently lack the ability to impart syntactical knowledge to the student – a benefit as this allows for quick production of programs, but a weakness as it is impossible to teach computer science solely through the use of a VPL. Students tend to enjoy visual

programming languages due to their ease of use. Thus, VPLs are often used with the goal of attracting and retaining students who have not been introduced to computer science, which is true for many female students [7]. Visual Programming Languages allow students to explore a new concept quickly and without the frustration that can arise out of the difficulty of using correct syntax, making them a valuable resource for Microlab Learning Cycle implementation.

2.4 Automated Grading Systems

Student participation is a universal goal in a classroom, but can be difficult to fulfill due to time constraints. In order for students to benefit from doing an in-class exercise they must be able to get some sort of feedback concerning their success. In a traditional classroom, this typically consists of the instructor giving the students time to work on the problem and then going over the problem in front of the class. While useful, this approach falls short on a few levels. First, it can teach less industrious students simply to wait for the correct answer to be given rather than try to solve it themselves – that process of trying to solve the problem being critical in forming a receptivity to a new schema in the student. Second, students are not able to try the problem multiple times as they are immediately given the correct answer – the students will not naturally explore the concept, they will come up with some answer and wait. Third, this approach doesn't provide individual evaluation and guide the students toward the correct answer; it gives it to them – reflecting an educational mindset of transference and not creation on the part of the student.

More and more classrooms attempt to overcome these problems by using “clickers” in the classroom. Clickers is a generic term for a mechanism that allows a student to attempt to answer a multiple choice question immediately and an instructor to instantly review the results. This approach ensures that all students interact with the problem as the provided answers are recorded. However, the clicker approach is largely limited to multiple choice questions, making it difficult to get students to have a meaningful second attempt versus quickly starting to “play the odds.” Furthermore, while the teacher can use the information to let students know if they answered incorrectly, the students reasoning is unknown and providing individualized feedback is not possible. Nonetheless, clickers can be used for a wide variety of problems and are easy to incorporate into the classroom.

Automated Grading Systems (AGS) offer solutions that clickers cannot. Automated Grading System is an apt title – it is a system responsible for handling student submissions without instructor input. Extending an AGS to provide feedback to students individually is a key characteristic that provides more utility than can be accomplished through clickers, which inherently group students into categories depending on which multiple choice answer they chose. Furthermore, an AGS does not have to rely on multiple-choice questions. Ken Bowles of the University of California San Diego first used an AGS to grade programming quizzes in Pascal [4], and Bill Pugh of the University of Maryland developed a system that can use multiple methods of grading student programming assignments, including “how well [students] test their own code” [15].

When using an AGS, as with clickers, the instructor is able to track attempts motivating less industrious students to work on the problem as accountability is built into the system. Furthermore, the students can attempt a problem multiple times, as the AGS can

immediately inform them of success or failure. Finally, if the AGS's feedback can be fine-tuned, then guidance can be given whenever the student submits an incorrect solution, encouraging the student to keep trying until they complete it successfully.

While AGSs overcome many of the difficulties inherent with in-class exercises, they introduce some new challenges as well. First, an AGS requires more work to implement as they handle a wider variety of input and provide output to the student. Secondly, while an AGS can create individualized feedback students must still be able to access the feedback. That is, each student must have their own access to the AGS, which typically requires connection to the school computer network or the internet. Therefore, implementing an AGS in a classroom requires some sort of mobile lab. Laptop carts are not uncommon in Universities, but usually create a fair amount of logistic overhead in their distribution, powering-on, and collection of the laptops. Tablets are a potential alternative which require less in-class overhead.

2.5 Use of Tablets in the Classroom

Tablets have become increasingly common, largely since the release of Apple's iPad in 2010. By 2014 there will be an estimated 90 million tablet users in the US [13]. The popularity of the tablet makes it a viable inclusion in the classroom, as tablet prices fall and most students have been exposed to tablet use by the time they enter a college classroom. Therefore, the availability and accessibility of tablets rivals that of traditionally used laptops for mobile-lab classroom implementations.

As tablets are still somewhat new, their full educational potential is unknown but developing. Apple has released their iBooks Author software for developing textbook materials that are accessible on the iPad. These iBooks can have embedded video examples, links to exercises, and dynamic illustrations (for example, a video of a cell absorbing and releasing nutrients through its membrane). Rich web applications can be developed for the tablet, making use of the drag and drop nature to ease user navigation and use. In addition to the iBook, various educational apps have been created that can be used recreationally or incorporated into classwork.

Development of an Automated Grading System for a tablet combines the strengths of each. The Automated Grading System provides individualized feedback for each tablet, while classroom incorporation of tablets allows for quicker access to the AGS than achievable through a laptop. Furthermore, the physical characteristics of tablets provide some advantages in the classroom. They are easier to move from classroom to classroom, and more fit into the same physical space. Perhaps most importantly, tablets often have longer battery life, allowing for use in subsequent classrooms without requiring downtime to recharge.

2.6 Conclusion

The Constructivist theory holds that students learn better by doing than by observing. Robert Karplus developed the Learning Cycle in order to incorporate these constructivist beliefs into the classroom in a systematic way. The Learning Cycle is composed of the exploration, invention, and application phases which lead into each other and are designed to help guide

the student into a formal operational understanding of a new subject through assimilation and accommodation of new schema. Visual Programming Languages and Automated Grading Systems increase the viability of incorporating the Learning Cycle successfully into the classroom. Using tablets as the mechanism that accesses the Automated Grading System and displays the Visual Programming Language reduces the overhead incurred of using the Learning Cycle in a traditional classroom.

Therefore, a system that combines the benefits of an Automated Grading System and a Visual Programming Language and that is designed for use on a tablet creates an environment prime for incorporating the Learning Cycle in a traditional computer science classroom.

Chapter 3

Pedagogical Issues and Goals

3.1 Implementing the Learning Cycle

The Learning Cycle focuses on students interacting with a new concept in a manner that allows them to take control of their own learning through the process of exploring and applying a new concept. Microlabs are small exercises designed to be the environment wherein a student can explore the current concept [10]. Microlabs are designed to be completed during lecture time in around 5-10 minutes while increasing active class participation and learning. Students can work on a microlab individually or in pairs. Each attempt at completing a microlab is automatically evaluated and feedback is provided for the student, either informing them of their success or providing some guidance to lead the students to the correct solution. Three different types of microlabs have been created for inclusion in the Learning Cycle. Each type of microlab contributes to the Learning Cycle in a particular way. The three types of microlabs are logical, programming, and code magnet microlabs.

3.2 Types of Microlabs

Logical microlabs are highly visual microlabs that are concerned with the underlying ideas and concepts of a problem rather than the specific implementation and syntax of a programming solution. Logical microlabs enable the student to solve a given problem at a very intuitive level, often by using a

drag-and-drop interface. For example, a student may be asked to build a binary search tree with a given postorder traversal by rearranging given magnets on a screen and drawing lines which indicate a parent-child relationship between them, as seen in Figure 3.1. Many students find this task confusing when dealing with abstract notions of Nodes, Leaves, Parents, and Children. The logical microlab

attempts to provide the student with a concrete representation of the problem that later will be abstracted to an algorithm, which is a more formal operational understanding of the concept.

Once the student arrives at a proposed solution, he or she clicks the “Evaluate” button. If the solution is correct

the student is informed of their success. If the solution is incorrect the student is provided

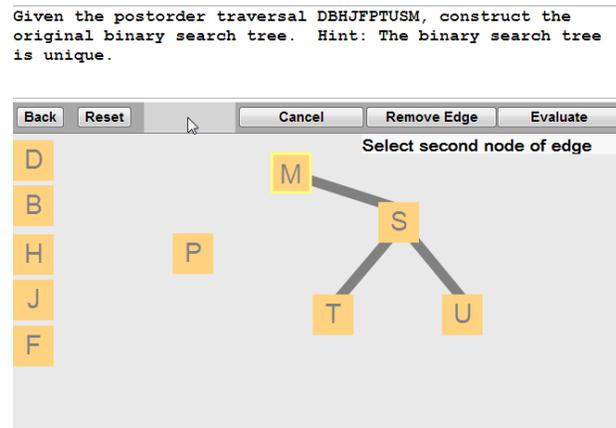


Figure 3.1 - A partially completed Logical Microlab

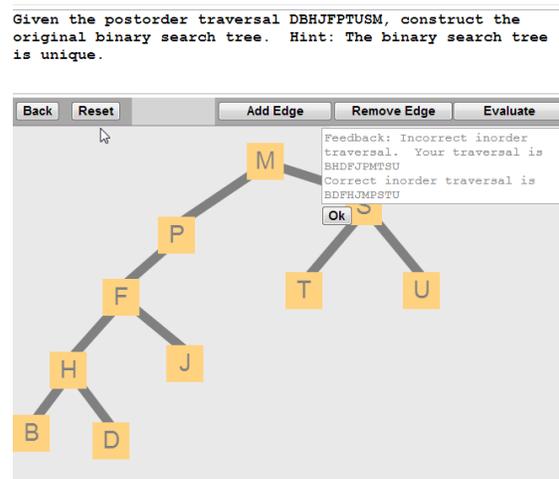
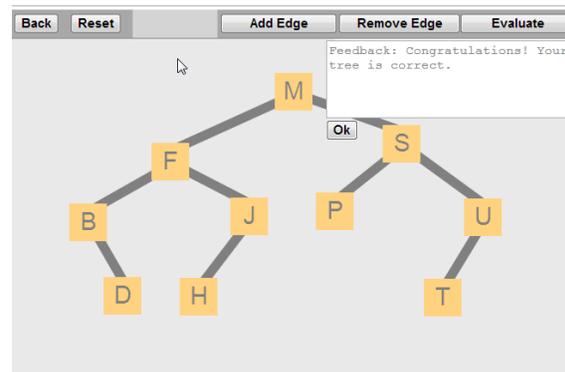


Figure 3.2 – An incomplete solution with guiding feedback

with guiding feedback, such as informing them how much of their tree is correct, or asking questions designed to guide the student to a correct solution. Figure 3.2 shows a student attempting to build a binary search tree from a postorder traversal. Upon submitting their solution, the student is informed that their tree has an incorrect inorder traversal. The system then lists the student's incorrect traversal and the correct traversal. Using the feedback from the system, the student re-arranges his tree and tries another submission. This time, the student is informed that he successfully completed the problem, shown in Figure 3.3.

Given the postorder traversal DBHJFPTUSM, construct the original binary search tree. Hint: The binary search tree is unique.



Figures 3.3 – A correct solution

To discourage systematic guess and checking, a limited number of full-credit attempts are usually given. Each subsequent submission beyond the full-credit attempts results in an incremental decrease of the grade for the exercise. This forces the student to engage with the problem, which increases the opportunity for cognitive dissonance or schema reinforcement.

Programming microlabs are the most challenging type of microlab, as the student is asked to type real code implementing a method or a portion of method that was introduced in class (usually by an accompanying logical microlab). The microlab testing environment hides all non-necessary implementation details, helping to focus the student on the task at hand as well as simplifying execution of the code from the student's perspective in order to minimize the time required for the exercise. Figure 3.4 shows the student's editor, limiting the editable portion of the program to the method that the student is implementing.

```
12
13  /* Your objective in this macro-exercise is to construct a binary search tree
14   * given only a passed in postOrder traversal of the tree.
15   */
16  public Node postOrderBuild(String postorder)
17  {
18      // if length = 0, tree = null
19
20      // find, create root node (always located at a certain position in
21      // post order traversal
22
23      // if length is 1, return current node (recursive base case).
24
25      // count the number of elements in the left subtree
26
27      // using the number of elements calculated to be in the left/right
28      // subtree, create two separate strings for left/right subtrees.
29
30      // set currentNode's left Node to recursive call on left subtree
31      // string.
32
33      // set currentNode's right Node to recursive call on right subtree
34      // string.
35
36      // return node.
37      return null;
38  }
~~
```

Figure 3.4 – The buildBST Programming Microlab outline as provided by the invention phase of the learning cycle

Detailed instructions are provided for each programming microlab, either as an easily accessible description as part of the microlab testing framework, or in-code comments that build upon what the instructor went over formally in class. Each submission is automatically compiled, run, and graded with a mark of either successful completion or not. When an incorrect solution is submitted, guiding feedback is given similar to what is found in a logical microlab.

As programming microlabs require typing real code, implementation of a programming microlab on a tablet device can be rather challenging, particularly when a virtual keyboard covers part of the screen. In order to overcome this limitation, another type of microlab was created, known as the code magnet microlab. Code magnet microlabs ask the student to construct real code without requiring the time of typing on a tablet. Building a

code magnet microlab out of a programming microlab is a straightforward and relatively easy process.

Code magnet microlabs use a drag-and-drop interface similar to what is found in logical microlabs. The draggable units are referred to as “magnets” which hold snippets of real program code, as seen in Figure 3.5. The student must then

arrange the magnets in the proper sequence and with correct nesting in order to construct a working method for the specific exercise. Figure 3.6 shows what a completed method looks like in code magnets. A code magnet microlab can usually be completed in substantially less time than the equivalent programming microlab.

Code magnet microlabs come with many possible implementation strategies that vary the difficulty of the exercise. For example, the instructor can decide whether or not to include unnecessary magnets, to include comments within magnets to help guide the student, or even

```
char rootValue = postorder.charAt(postorder.length()-1);  
  
int countLeft = 0;  
  
for(int j = 0; j < postorder.length(); j++) {  
}  
  
if(postorder.length()==0) {  
}
```

Figure 3.5 – Some magnets used in the BuildBST exercise

```
public Node postOrderBuild(String postorder) {  
  
    if(postorder.length()==0) {  
        return null;  
    }  
  
    char rootValue = postorder.charAt(postorder.length()-1);  
  
    Node root = new Node(rootValue);  
  
    if(postorder.length()==1) {  
        return root;  
    }  
  
    int countLeft = 0;  
  
    for(int j = 0; j < postorder.length(); j++) {  
        if(postorder.charAt(j) < rootValue) {  
            countLeft++;  
        }  
    }  
}
```

Figure 3.6 – buildBST implemented as a code magnet microlab

to implement the entire exercise in pseudo-code rather than asking the students to identify correct syntax. As such, code magnets microlabs are an extremely flexible and platform independent form of microlabs.

Microlabs are designed to supplement and increase success on out-of-class programming exercises; they are not designed to replace such assignments. Most subject material covered with microlabs has a natural extension that can be implemented in an out-of-class programming assignment. For example, after completing all the buildBST exercises with the Microlab Learning Cycle which use a postorder traversal, a natural assignment would have the students write a program that builds a BST from a preorder traversal. This would require the students to incorporate their knowledge of boundary cases in tree building, and write their own testing methods to verify the correctness of their implementation.

3.3 The Microlab Learning Cycle

Microlabs were designed with Karplus's Learning Cycle in mind. Thus, instructors are encouraged to follow the Microlab Learning Cycle when employing microlabs in their classroom.

In the exploration phase of the Microlab Learning Cycle, the teacher very briefly introduces a new problem or concept. The students then explore this problem with a logical microlab, and attempt to come up with a solution. This phase seeks to introduce the student to the topic in such a way that the student is confident in their ability to succeed, by the use of concrete representations of ideas, as well as encouraging the student to assimilate the problem into an existing schema when possible. If the student has no appropriate existing

schema he or she may succeed by guess and check, but this has opened the door for introducing a new schema that does not require guessing during the invention phase.

Students are typically willing to return to lecture mode if they are allowed to finish the exercise before the next class period and still receive full credit. As students who have not yet finished the assignment are both those who would benefit more from lecture and those who are least likely to want to divert their attention from the exercise, it is important to make the due date of the exercise clear. Returning the students to a lecture at such a point capitalizes on the cognitive dissonance engendered by the inability to find a correct solution in some cases, making the students more attentive and more likely to go back to the problem outside of class to complete it successfully.

This time of lecture is the invention phase in the Microlab Learning Cycle. This phase is the most similar to the traditional classroom structure. At this time the instructor should use the experience the students gained experimentally and intuitively, and turn it into a more formal form of knowledge via notation, representation, or an algorithm. This often involves the instructors helping direct the students into formulating a pseudocode algorithm of the process they just worked through conceptually. For example, the algorithm for building a binary search tree from the postorder traversal might be expressed in the following manner:

```
node buildBST(postorder)
    base case: return null if postorder is empty
    create root, using the last value in postorder
    base case: return root if postorder has only one character
    count the number of nodes in the left subtree (those < root)
    create the postorder traversal of the left subtree
    build the left subtree using recursion
```

```
create the postorder traversal of the right subtree
build the right subtree using recursion
return the root
```

In the application phase of the Microlab Learning Cycle the students turn back to another microlab, generally a programming or code magnet microlab. Both types of microlabs require the student to build up a method that implements the solution discussed in the invention phase in code. This gives the students an opportunity to assimilate the schema developed in the invention phase while providing exposure to an actual implementation in code.

When the instructor designs a class session using the Microlab Learning Cycle, deciding between code magnet and programming microlabs is important. Programming microlabs currently have support for F# and Prolog, while code magnet microlabs do not. Programming microlabs also provide greater flexibility of implementation and more of a focus on syntax. Code magnet microlabs are capable of being used on tablet hardware – they’ve been tested on 7” Google Nexus tablets and were well received even on such a small screen size. Finally, code magnet microlabs are quicker to complete in class, as well as providing students with hands-on experience reading and understanding code – an aspect of the computer science industry that sometimes goes underrepresented in computer science curriculum.

Qualitative feedback on microlabs shows that students generally enjoy the Microlab Learning Cycle approach, finding it both valuable and interesting. However, the collection of quantitative data is equally if not more important. Thus, testing long-term retention of material covered using the Microlab Learning Cycle is critical for further advancement of computer science curriculum. In-class tests provide a perfect opportunity for this type of

study. For example, to test the long term retention of the buildBST material the instructor could include the test question “Given the postorder traversal 1 4 7 6 3 13 14 10 8 draw the original binary search tree.” Notice that the problem domain has been changed from a tree with character nodes to a tree with integer nodes; transference of the schema to similar problems is important.

3.4 Designing a Microlab Framework

3.4.1 Goals

A framework has been designed that helps in the creation and dissemination of microlabs in order to make the incorporation of the Microlab Learning Cycle in computer science education as simple as possible. For historical reasons this framework is known as Wags, which stands for Web Automated Grading System. Both students and teachers use Wags for interacting with all types of Microlabs. For the teacher, this consists of assigning microlabs, optionally creating microlabs, and gathering grades on the exercises. Students use the framework to complete each exercise, be it a logical, programming, or code magnet microlab.

Framework design is critical, as it is possible for the deployment environment to obstruct the usefulness of the Microlab Learning Cycle if implemented poorly. That is, the efficacy of microlabs could be limited by the framework if it proves a hindrance to the student or teacher. Therefore, steps were taken to make Wags as user-friendly as possible, resulting in an iterative update/deploy cycle as the framework was revised to more closely align with user wants and needs. In order to ensure this over-arching goal, Wags was designed with certain requirements in mind.

The first requirement is that the framework be easy to navigate and thematically consistent. Second, the framework must allow for on-site creation of individualized microlabs to give the instructor greater control over the classroom. Third, microlabs must have a core structure that encourages and facilitates the incorporation of guiding feedback for the students. And fourth, the framework allows the instructor to do all necessary classroom management and administrative work within the framework itself, including the registering of students, assigning of exercises, and reviewing of submissions.

3.4.2 Usability

For the student, Wags is used for completing three different types of microlabs. Thus, the framework was split into three main sections. Providing a consistency for the user among the three sections of Wags while respecting the differences among microlab types was a necessary goal. An inherent difference was quickly made apparent between programming microlabs and logical/code magnet microlabs, as programming microlabs would implicitly be unsuitable to a tablet environment while logical and code magnet microlabs were designed primarily for such an environment.

Upon navigating to the Wags website, each user is asked to sign in. From this homepage they have the option of going to the programming, magnets, or logical page, as seen in Figure 3.7. A student can navigate to a different page by using the navigation bar that remains at the top of the screen during the

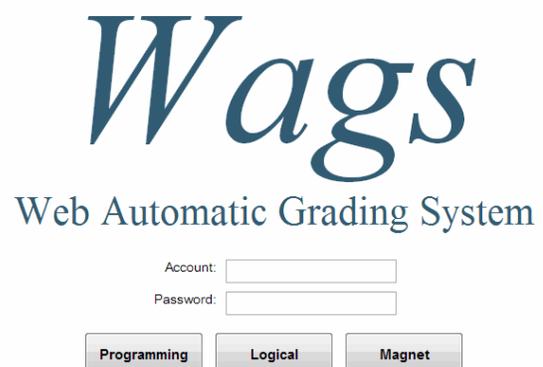


Figure 3.7 – The Wags sign in screen

Wags session. This navigation bar is also used to notify the user of successes and failures of attempts, as well as for any other necessary information. Thus, the navigation bar is one way of providing consistency among the different areas of the Wags framework, as shown in Figure 3.8.



Figure 3.8 – The WAGS Navigation Bar

In addition to separating access to each type of microlab, Wags provides the student with the option to review previous work done on each type of microlab. This functionality is inherent in programming microlabs where each file the student works on is saved periodically and can be accessed at any time. For logical and code magnet microlabs, this ability is provided through the Logical Index Page or Code Magnet Index Page. These pages display a list of currently assigned exercises for the student – these are the pages the student gets directed to upon clicking the “Magnet” or “Logical” button on the homepage. These pages provide the student with a “Review Past Problems” button. Upon clicking this button, the list of exercises changes from those currently assigned to all of those that have been assigned in the past, as seen in Figure 3.9. Completing exercises from this list will not update any submission information, so a student cannot complete work after the due

Logical Microlabs

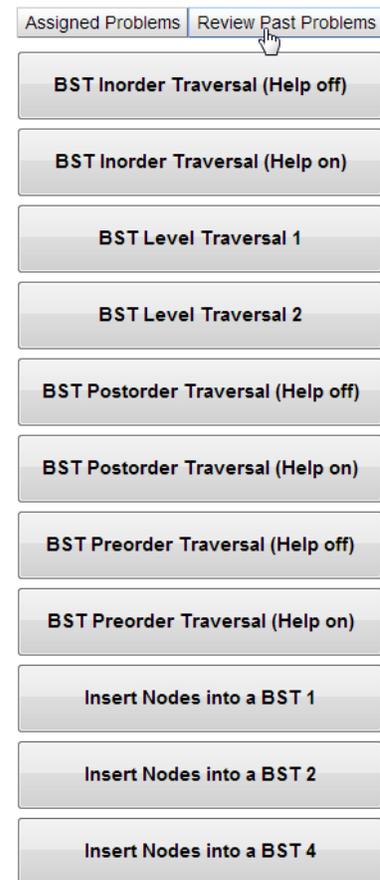


Figure 3.9 – A list of previously assigned exercises the student can revisit

date, but it will allow the student to work through each exercise as many times as they would like; students have found this to be useful when studying for an exam.

3.4.3 Creating Microlabs

Many microlabs are made readily available to any instructor using the Wags framework. However, having pre-developed microlabs for every possible instance and use is not feasible. Thus, Wags allows the instructor to create their own code magnet and programming microlabs, or create variations of pre-existing logical microlabs. This functionality allows Wags to be individualized for each class, and does not attempt to force any instructor into spending a set amount of time on any one aspect of the course material.

Programming microlabs are the simplest to create. An in-depth tutorial on this process is included in the Wags user manual in the supplementary material. In brief, the development of a programming microlab is little more than the writing of three classes, each with a different role. The first class is the Student class, which students will edit during the exercise. The second class is the Solution class, which provides the correct answers for the test cases contained in the Test class. The final class is the Test class, which contains the `main` method and runs the evaluation of the Student class. Additional Helper classes can also be uploaded if the exercise requires use of classes not included in the Java library. For example, the `buildBST` lab required a Node class with methods such as `getLeft`, `setLeft`, and so forth.

The Solution class can be implemented in a few different ways. Perhaps the quickest way to create a working exercise is to hard-code in the solutions to the test cases which the student should find. For example, if the student is assigned to create a method that sums all

of the integers of an array, the instructor could just hardcode in the test cases and the correct answers and not be required to implement the method in the Solution class. However, actually implementing the method has many benefits. It allows for hardcoded test cases for limiting cases, along with a series of randomly generated test cases. Furthermore, it is much easier to expand upon classes with the correct implementation than those with hardcoded answers. Nonetheless, either approach works within the Wags framework.

The Test class typically initializes a variable of the Student type and another of the Solution type. Multiple tests with varying input, including boundary cases, are then run. After each test, the result from the Solution class method is compared with that of the Student class method. If all results are equivalent, the Student has correctly implemented the



The image shows a web form titled "Uploading" with a light blue background. It contains several input fields and buttons. The fields are: "Exercise Name:" with a text input; "Solution:" with a "Choose File" button and "No file chosen" text; "Skeleton:" with a "Choose File" button and "No file chosen" text; "Testing Class:" with a "Choose File" button and "No file chosen" text; "Format: DD monthName YYYY HH:MM:SS" with a text input; "Opens at:" with a text input; and "Closes at:" with a text input. At the bottom left is an "Add Exercise" button. A mouse cursor is pointing at the button.

Figure 3.10 – The form used to upload a manually created programming exercise

method. After these classes are written and tested in the developer’s favorite environment the exercise can be uploaded using the Wags system. The instructor should navigate to the Editor page for programming microlabs and select the “Exercises” tab on the right side of the screen. The instructor can then upload the exercise using the form seen in Figure 3.10.

In addition to the aforementioned process for creating microlabs, the instructor must ensure that the test class signals to the system when a student has correctly implemented the method. While this is typically done by returning a boolean value within a Java program, Wags requires notification to the system and not just the testing program. Thus, a slightly more robust method must be employed, a method that deters any attempts at falsifying a

correct submission. The technique Wags employs is referred to as “nonce validation,” and is necessary for both programming and code magnet microlabs.

Nonce validation is a simple concept. Each Test class must accept an argument from the command line. This argument is created individually for each submission and is randomly created by the Wags server. The server records this value (referred to as a “nonce”), and when the program is finished running it checks the output for this value. If the nonce was printed, the student successfully completed the exercise. The server then removes the nonce from the output, and returns the rest of the output to the client for display. Thus, students are unaware of the presence of a nonce value at all, and even if they were made aware they would be unable to predict or report it from their Student class. Implementation of nonce validation is straightforward, as it simply requires the Test class to print the nonce after all print statements in the program if the student completed the exercise successfully.

Code magnets are created very similarly to programming microlabs, as they also suggest the use of a Test and Solution class. The Student class is built out of the Solution class as magnets are created corresponding to the statements in the Solution class. This conversion is done on the Problem Creation Page, as seen in Figure 3.11 on the next page. More extensive instruction for creating code magnet microlabs can also be found in the appendix. In short, the microlab is developed using a front-end that automates the inclusion of all necessary syntax for code magnet microlabs to function. This abstracts away the actual implementation behind a mode magnet microlab, leaving the instructor with the simple process of choosing which magnets to create and in what order they are displayed on the left side of the screen.

Figure 3.11 – The form used to create Code Magnet Microlabs

The left side of the Problem Creation Page handles all input necessary for creating the microlab, while allowing the instructor to choose an exercise to edit or use as a basis for a modified version. The right side of the Problem Creation Page shows the output with necessary syntax that will be placed into the database and used to create the code magnet microlab. The right side of the page is editable to allow for quick fixing of typos or syntax errors.

Each created code magnet microlab is placed into a group specifically for the instructor who developed the microlab. The instructors are then free to edit and update this microlab as they see fit. When altering a pre-existing microlab, the instructors must save the changes into their own group, and cannot modify the public microlab. A public repository of microlabs is in development which would allow for exercises to graduate from private to publicly available, allowing a scalability of exercises beyond what is possible for a limited number of devoted developers.

Logical microlabs are significantly more complex to develop than code magnet or programming microlabs, as different logical microlabs require their own particular representation of information. For example, a radix sort exercise cannot use the same display

as a binary tree exercise. This requires development of many different displays.

Furthermore, logical microlabs are evaluated on the client, as opposed to both programming and code magnet microlabs which are evaluated on the server. Thus, a new type of logical microlab cannot be created from the ground up without access to client code and requiring recompilation and deployment.

Nonetheless, logical microlabs do exist on the server in a sense. They are stored in the database and are fetched by the client who subsequently constructs the actual problem. Thus, allowing instructors to create new logical microlabs as an act of adding to the database is a feasible goal, but limits new logical microlabs to being varieties of currently existing problem types. For example, while instructors could create new binary search trees for students to traverse, they would be unable to implement some new sort of traversal to have the student attempt. Furthermore, as there are substantial differences among logical microlabs, different types of logical microlabs require different interfaces for even altering pre-existing logical microlabs.

The mechanism that allows for these alterations is currently in development. Each logical microlab type will have a slightly different implementation of the same base mechanism. All of these implementations will be found on the Admin Page under Problem Creation, and will have more thorough instructions as to their use in the accompanying Wags Manual as their development continues.

3.4.4 Incorporating Guidance in Microlabs

The purpose of a microlab is to increase active learning in the classroom, causing improved understanding and retention of material through a series of short exercises that can be used

during lecture time. The traditional limitation with assigning exercises in the middle of lecture is the inability of the instructor to interact with each student individually to provide guidance and feedback as they attempt to complete a problem. Microlabs solve this problem by asking each student to solve exercises individually or in pairs while providing each student with individualized feedback when the student submits an incorrect solution, as well as automatically keeping track of student attempts and successes for the teacher's records.

Guiding the student towards a correct solution is a crucial aspect of the exploration phase of the Learning Cycle, and by extension, the Microlab Learning Cycle. Each time a student fails to correctly complete an exercise he/she should be given a piece of information that helps identify the problem and guides them towards the solution. While the most effective way to do this differs with each subject, there are a few common techniques that are usually helpful. These techniques are most important for programming and code magnet microlabs, as the microlab framework allows for instructors to create their own versions of these microlabs. As logical microlabs are substantially more complex to develop all logical microlabs are provided with the microlab framework, and each type of logical microlab already provides guiding feedback.

Programming and code magnet microlabs are exercises that involve the development of program code. This code is then sent to the server hosting Wags where the code is compiled, run, and tested. When developing a new exercise the instructor is responsible for writing a testing method for the exercise. Generally, each exercise consists of the student being provided with some input and working with that input to produce some output. This output is generated through a series of steps. Thus, helpful feedback for the student typically

includes the data the program was provided, the desired output from the program, and information detailing success or failure at different steps in the algorithm.

For example, when attempting the “BuildBST from Postorder Traversal” code magnet microlab, the student is informed of the postorder supplied to their method, the postorder traversal of the tree their method built, as well as the preorder traversal of their tree and the preorder traversal of the correct solution, as seen in Figure 3.12.

Visual representations of concepts within the microlab are often quite useful for the students, as provided in the code magnet microlab exercise for building a binary search tree as a series of insertions of different nodes. This exercise not only provides the students with the input to their method as a traversal, but also shows a graphical representation of the tree. Then, as the students are asked to add a node, the testing method also prints the tree the student’s method built. This output is shown in Figure 3.13.

```

For postorder: ACBEGFD
You generated: GFD
Correct preorder: DBACFEG
Your preorder:   DFG

For postorder: DGFQTMJ
You generated: TMJ
Correct preorder: JFDGMTQ
Your preorder:   JMT

For postorder: BAHJDPMZTK
You generated: ZTK
Correct preorder: KDABJHTMPZ
Your preorder:   KTZ

For postorder: DFBINLZRKG
You generated: ZRKG
Correct preorder: GBFDKIRLNZ
Your preorder:   GKRZ

Incorrect

```

Figure 3.12 – Sample output from an incorrect attempt

```

Test [0]: Pre-insertion preOrder Traversal -> XHGAK
Test [0]: Pre-insertion inOrder Traversal -> AGHKX
Test [0]: Pre-insertion postOrder Traversal -> AGKHX
Tree Depiction:
      X
     / \
    H   K
   / \
  G   A
 /
A

Random Character to Insert: R
Your tree after insertBST method called with : R
      X
     / \
    H   K
   / \
  G   R
 /
A

```

Figure 3.13 – Output with more visual representations of data

In all cases, students greatly benefit from being provided information in a way that is familiar to them due to their exposure to the logical microlabs previously. For example, for the Build BST from postorder exercises, students are provided with the postorder traversal just like they were given in a logical microlab. Then, if the output from their application phase microlab is incorrect, they often will sketch out a tree using the techniques developed from the logical microlab. Upon verifying that their method produced an incorrect result, students will often go through their program sequentially testing it against their understanding gained from the logical microlab.

As students will attempt to work out a problem by hand, code magnet and programming microlabs are particularly well equipped mechanisms for teaching students about the importance of boundary cases, and writing code that handles them appropriately. For example, for the buildBST problem a tree with a single node or an empty tree represents a boundary condition. Therefore, each testing method should take special care to create boundary tests. This in turn creates ready candidates for alternative and incorrect magnets for code magnet microlabs. Repeated exposure to boundary case tests and magnets that fail to incorporate boundary cases correctly lead to a student cultivating a mindset aware of limiting cases of algorithms even beyond the scope of the microlab framework, especially when used in conjunction with external programming assignments.

3.4.5 Classroom Management

While the student perspective of the Wags framework is mainly characterized by working on various microlabs, the instructor requires more from the system. Therefore, Wags aims to

allow instructors to efficiently register students, assign exercises, and review student attempts on various microlabs.

Wags accounts are split into sections. Each section relates to an academic class, such as “Introduction to Computer Science” or “Data Structures.” Each section has a single administrator and multiple students. Sections are created by the Root account which decides on a section name (for internal reference) and sets up an administrator for the section. The administrator can then log in to their account and add students to the section, reset passwords for the students, assign exercises for the students to complete, and review assigned exercises. All administrative work occurs on the programming microlab page. Extra administrative tabs are displayed on the page when the system detects an administrator’s account. The instructor’s first step in setting up their Wags section for use in lecture is registering students.

In order to guarantee unique account names, Wags suggests using student e-mail addresses as usernames. A comma-separated value (CSV) file is used to register multiple students simultaneously. The file should have a separate student on each line, with each line having the form of “Lastname, Firstname, Username” for the student. Microsoft Excel allows for easy exporting of CSV files. Once the file is prepared, it can be uploaded on the “Student” tab of the programming microlab screen. Upon successfully creating the accounts a green notification message will appear in the navigation bar informing the instructor of their success. Each student account is initially created with the password “password.” The system prompts the student to change their password when they first log in as it detects the word “password.” Each password is saved using a MD5 hash in the Wags database.

The instructor can assign exercises to the students using the “Exercises” tab of the programming microlab screen. This tab is vertically split into five areas for different

administrative tasks. The top area is labeled “Uploading” and is responsible for uploading programming microlabs and making them available to the section. The next three areas are split between programming, logical, and code magnet microlabs. Each area allows for assigning and reviewing exercises. Programming microlabs are chosen from a drop-down list of previously uploaded exercises. Logical microlab are split into subjects (such as binary trees) and groups for each subject (such as “traversals” or “build BST”) in order to simplify navigation among exercises. Magnet microlab actions are separated by subject. Logical and code magnet microlabs are assigned by clicking the check box corresponding to each exercise. Upon clicking the “Set” button for logical or code magnet microlabs, each selected exercise is made visible on the corresponding Index Page.

All three areas allow for reviewing an individual microlab. This is useful during class time to monitor progress on the current assignment. Given this information, the instructor can move on to the next stage in the Learning Cycle or devote more time to the current problem as indicated by student success.

Username	Correct	NumAttempts
al[REDACTED]@appstate.edu	Yes	1
ar[REDACTED]@appstate.edu	No	5
be[REDACTED]@appstate.edu	No	0
br[REDACTED]@appstate.edu	Yes	5
ca[REDACTED]@appstate.edu	Yes	4
he[REDACTED]@appstate.edu	Yes	2
hd[REDACTED]@appstate.edu	Yes	1
hd[REDACTED]@appstate.edu	No	0
hu[REDACTED]s@appstate.edu	Yes	7

Figure 3.14 – A section of the real-time review table available to instructors

A sample review output is shown in Figure 3.14. The information for this chart is updated every time a student submits an attempt, allowing the teacher to continuously monitor completion in real time. The final area allows for a collective report of all assigned Microlabs. A CSV file is available for download containing information on student submissions for all exercises by clicking the “Review” button in the Comprehensive Review area. Microsoft Excel can import the CSV file and will automatically create a spreadsheet containing all of the corresponding information.

Chapter 4

Software Design and Implementation

4.1 Design and Implementation Decisions

Wags is designed for use during a classroom lecture. The ease or difficulty of incorporating Wags into the classroom depended largely on making correct choices about how Wags should be implemented. Given that Wags provides access to microlabs, which are supposed to be used during lecture time, Wags had to be developed for an environment that is accessible within a non-lab classroom. Laptops are an obvious option, but given the time constraints of a microlab, the overhead of passing out, booting up, logging in and logging out of a laptop would prove prohibitive. Depending on students using their own laptops is similarly problematic. Tablets overcome many of these shortcomings. Due to their smaller size, carting tablets and passing them out takes much less time. Tablets hold a charge longer than laptops allowing them to be turned on and set up before students arrive, further speeding up the process.

Deciding upon tablets as a target device narrowed the field of implementation choices. As different schools may use tablets with different operating systems and access to different App Stores, developing Wags as a tablet application would create too many limitations concerning accessibility. Thus, the only reasonable solution seemed to develop

Wags as a rich web application (RWA) with access to an external database that stores accounts, grades, exercises, and so on. Then, any device with a commonly-used browser should be able to access and use Wags, regardless of underlying operating system.

When approaching the development of a sizable application, investing time in the beginning in order to ensure making the right decisions about what tools to use can save time throughout the lifetime of the project. For Wags, this meant making decisions about what type of database to use, what framework should be used for application development, how the application would be hosted, and how different aspects of the application would interact with one another (communication between client and server, and between server and database, for example).

Many of these questions were easily answered given what was already in use at Appalachian State University, where Wags was developed. The computer science department had a machine running Linux, Apache, MySQL, and PHP for web hosting, commonly known as a LAMP setup. Familiarity with the LAMP structure was a further advantage, as well as all components of the LAMP software bundle being free and open source. Additionally, Wags uses the PHP Data Object (PDO) library to control access to the database. Thus, making decisions about the server side implementation of Wags was a straightforward process.

As Wags is a rich web application, the client side was going to be sizably more complex than a standard website. Wags depends upon a wide variety of Ajax calls and multiple interactive user interfaces, and needs to perform correctly on various browsers in order to support truly cross-platform tablet use. Ajax is a term used for a set of web

technologies that allows client and server communication to occur in the background and can update only portions of a page in order to provide the least disruption to the end user.

Multiple frameworks exist for easing the development of creating a rich web application, finding the correct one became a primary task.

Initial development was done using JavaFX, but this was quickly abandoned since this environment is not supported on most tablets. Google Web Toolkit (GWT) ended up meeting all the requirements and being the choice for the Wags development environment. GWT had a series of advantages making it the choice above all other frameworks. Perhaps the most critical advantage was the Java-to-Javascript compiler that runs at the heart of GWT. Given that Appalachian State's computer science curriculum teaches Java to a far greater extent than any dedicated web-development language, allowing development to occur in Java meant an overall faster development time. Cross-browser issues were eased as the GWT compiler cross-compiles multiple permutations of the JavaScript code optimized for various mainline browsers, abstracting away many browser quirks.

Wags relies heavily on Ajax calls and GWT simplifies this with the RequestBuilder class which encapsulates PHP get and post requests, allowing for much cleaner client-server interaction. Furthermore, GWT gives the client side code potential for more organization, not only because writing in Java allows object-oriented design principles to be used, but also because GWT comes with a user-interface utility known as UiBinder. UiBinder separates the presentation from the control logic by allowing declarative programming of an interface using XML, with automatic association with a corresponding Java file responsible for the logic of the interface. Finally, GWT has plugins available for Eclipse, an integrated development environment that is invaluable in writing code. Developing in Eclipse also

meant that Wags could use some plugins for version control, along with an SVN repository supplied by Google.

Thus, the basic implementation decisions for Wags were made. GWT would be used for client-side code development, which would run as JavaScript across multiple browsers.

The server code would be written in PHP, with an underlying MySQL database accessed via PHP::PDO. The entire application would be hosted using Apache on a Linux machine.

The project would be developed in Eclipse, using an SVN repository for version control as provided by Google Code Hosting. This high level view is summarized in Figure 4.1.

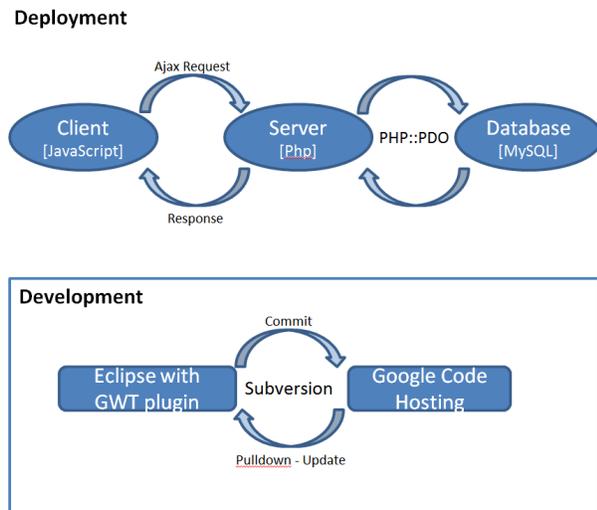


Figure 4.1 - The underlying structure for Wags during deployment and development

4.2 Client Side Structure

As Wags is written in Java, much of the organization is expressed through the package hierarchy employed within the project. The hierarchy was designed to minimize cross-package imports, while allowing sub-packages to import from parent packages freely. This resulted in an intuitive top-down organization, with higher level packages containing classes which are often subclassed in subpackages. Given that Wags is not designed to be included

in another project, but rather as a standalone application that exists primarily as compiled code, the traditional naming conventions that guarantee uniqueness were abandoned in favor of a scheme that provides more clarity as to the package contents.

Thus, the topmost package of the Wags project client side code is simply called *wags*. This package contains the user-interface classes that are responsible for the unified theme among the Wags application and thus are shared on all pages. This package also holds the entry point for the GWT application, and the Proxy class, which handles all non-form interaction between the client and the server. Form interaction relegates as much interaction control as possible to the Proxy class, although some interaction resides within subpackage classes. Thus, Wags adheres to a Proxy design pattern whenever possible, letting the high level Proxy class create all RequestBuilder classes, as well as holding the classes that parse server responses. The general process is the following sequence:

1. A lower class will call a static Proxy method, supplying the parameters for the request as well as either whatever widget needs to be modified or the calling class itself if it implements an interface responsible for callback methods.
2. The Proxy class then sends the request and parses the response, editing the widget or panel appropriately after the response from the server is received. This usually means creating a WEStatus object, which may contain further objects constructed from the response itself, or whatever message or variables the server sent down in a modified JSON form.
3. At this point, the program either waits for a new event on the client (such as clicking a button), or uses a callback method to return control to the previous class.

In addition to the mentioned classes, the *wags* package contains three sub-packages: *wags.magnet*, *wags.programming*, and *wags.logical*. Both the *wags.magnet* and *wags.programming* packages serve only as wrappers to subsequent sub-packages, *wags.magnet.view* and *wags.programming.view*. This occurs as both code magnet microlabs and programming microlabs are stored, compiled, and executed on the server. Given that the *wags* high level package is responsible for client-server interaction, the remaining code specific to code magnet microlabs or programming microlabs are concerned primarily with displaying the exercise and allowing students to modify code before sending it back up to the server. GWT provides a utility called UiBinder to allow declarative writing of user interfaces. This utility splits a page into two parts – the presentation of the page (an XML file) and the logic that accompanies the presentation (a Java class). This allows for cleaner code and easy updates of either the view or the logic without requiring both to be re-written. UiBinder is used heavily in Wags; the entire *wags.programming.view* package uses it as does the majority of *wags.magnet.view*.

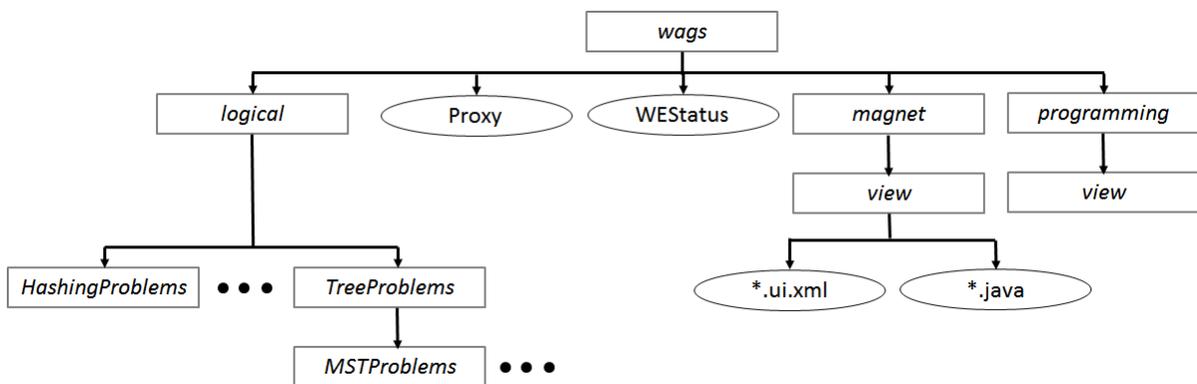


Figure 4.2 – The abbreviated Wags package hierarchy

The structure of Wags is reminiscent of the Model-View-Controller design. The most straightforward expression of the “View” are the UiBinder XML files, although a case could be made to include the corresponding Java files in the same category. The Proxy class and the Commands on the server (expanded on in the next session) relate to the “Controller,” as they control data flow and interaction and are relatively disconnected from the presentation of the information. In addition to Commands the server has a Class directory, full of PHP Objects that extend the WAGS Model class. The Models and corresponding information in the database then correspond to the “Model.” A visual comparison is shown in Figure 4.3.

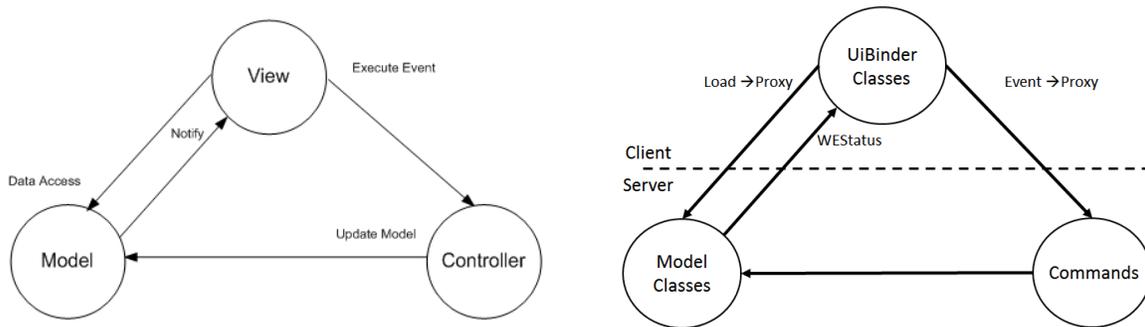


Figure 4.3 – A general MVC design (left) and the corresponding Wags elements (right)

The *wags.logic* package is more complex than the other two sub-packages, as logical microlabs are run and evaluated on the client itself. Logical microlabs are typically composed of three parts: a Problem class, which represents the specific exercise; a DisplayManager class, which handles the representation of the problem on the screen and how students can interact with it, and an Evaluation class, which contains the logic to check the student’s solution. Problems, DisplayManagers, and Evaluations are combined differently depending on the specific exercise. For example, a TreeProblem may use a TreeDisplayManager or an MSTDisplayManager. Some TreeProblems use the Evaluation_BSTTraversal evaluation, or they may need to implement the more specific

Evaluation_PostOrderBST. Separating the logic into these three roles allows for an optimal combination of code-reuse without sacrificing clarity.

The *wags.logic* sub-package structure is organized around the notion of a problem type, rather than by Evaluation, DisplayManager, or Problem. This approach minimizes import statements, and subscribes to the general philosophy of allowing imports from parent packages, but not sibling packages. Thus it is only natural that the high level *wags.logic* package contains the base Problem, DisplayManager, and Evaluation classes, among other commonly used classes that are not always extended (such as Node). In addition to these classes, the *wags.logic* package contains sub-packages for each noticeably distinct problem type. Currently, there are 6 immediate subpackages, with the *wags.logic.TreeProblems* package containing three more subpackages.

Wags is a rich web application, and the client carries the complexity that such a title connotes. However, by employing a sensible, intuitive package structure, delegating server interaction to the Proxy class, and separating the presentation, logic, and data, the Wags project client code becomes manageable and intuitive.

4.3 Server Side Structure

The server side of Wags exists primarily in a nested directory structure. The highest level of the structure holds the necessary cascading style sheet (CSS) rules and entry-point into the Wags code. The only other file in the highest level is the “Server.php” class, which is the target of all Proxy communication. The Server class then uses a CommandFactory class to construct whatever command the client requested, as well as then calling “execute()” on that

command. The purpose of the server is instantiating a command class that responds to the client-initiated requests. All other classes on the server support this role.

The second level of the server is the “class” subdirectory. Within this directory is a series of PHP classes that represent entities within the database. Each class in this directory extends the Model.php class, which has built in methods for assigning and retrieving universal database row characteristics, such as the id of the entity, the time it was added to the database, and when it was most recently updated. Model.php also contains “save()” and “delete()” methods, allowing for easy updating and cleaning of the underlying database. In addition to reflecting the columns of the corresponding class in the database, nearly all PHP objects in the class directory come with a series of static functions that further interact in the database. Thus, the users of Model classes never write SQL code directly, but rather call methods in the corresponding PHP object to accomplish the same end.

The lowest level of the server is the “command” directory. This directory contains the logic for every command that the client could request of the server. The program flow follows a stepwise process: the client sends a request, then Server.php catches it and builds the appropriate command from the CommandFactory, and then the command avails itself of the static methods in each class to create any necessary objects, run necessary logic, and return the requested information to the client. While this general scheme is rather simple, actual implementation can be involved. For instance, compiling and executing student code requires the use of multiple server-side classes. Important roles are distributed among the necessary classes, including gathering student code, associating the correct exercise files (the Test and Solution class) with the current student code, compiling the code safely so students cannot abuse the host machine, and forking and monitoring an execution thread so potential

infinite loops can be shut down before consuming too many server resources. However, this distributed scheme makes

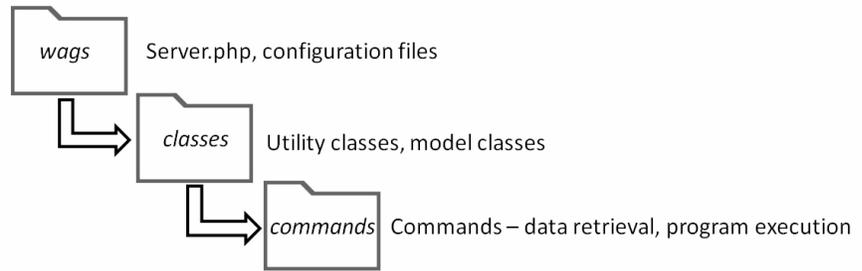


Figure 4.4 – Server side directory structure

it simple to incorporate new capabilities of Wags that rely on compilation and execution; while the overhead for running programming microlabs was high, sharing that functionality with magnet microlabs was a much easier process. The server directory structure is shown in Figure 4.4.

The server interacts with three areas of the Wags client code: forms, the Proxy class, and the WESstatus class. As both forms and the RequestBuilder class used by Proxy mimic get and post requests, the server is most interested in responding to the client in a way that is accessible to the WESstatus class. In order to ensure easy interaction between the server and WESstatus, a modified form of JSON is used as declared in the class/JSON.php file on the server. In short, the server wraps a JSON response with some additional information which is used by WESstatus to monitor the success of the request. WESstatus then parses the response into a “status” and a “content” – the content normally being some sort of message to be displayed to the user, but sometimes representing a client side object which is then created. Using the modified JSON and WESstatus classes allows both the client and the server to interact with each other in a standardized way, allowing quick development of client-server interaction code.

In short, the Wags server is mainly concerned with two things: running programs and data flow either into or out of the database. A series of classes is used to accomplish these

two tasks in an efficient and intuitive way. As so much of Wags is concerned with storing and retrieving information from the database, database organization is extremely important to the efficacy of Wags and the ease of the development process.

4.4 Database Structure

Wags uses a MySQL database and PHPMysqlAdmin for most database administration. Each table in the database fills one of three roles. A table can represent some entity used either on the server or even the client (like a logical microlab or a User). A table can be used for organizing associations (like index tables). A table can store information used strictly for running programs (such as the text to create files for different exercises). In addition to the three roles, each table relates to one of four “subjects” – a logical microlab, a programming microlab, a code magnet microlab, or a “section.” Section is the term used to represent an academic class employing Wags. Each section has an administrator account (used by the instructor), a series of regular users (the students), and a series of exercises (logical, code magnet, and/or programming). Every table in the database can be seen as fulfilling a role for a subject, although there is sometimes more than one table fulfilling the same role for a single subject. For example, both the LMGroup and the LMSubject table is an organizational table for the LogicalMicrolab subject. However, if a proposed addition to the database cannot be seen in terms of a single role and subject, the table should either not be included or re-organized so it only fills one role for a single subject.

This rule was not always followed in the development of Wags. For a while, the “section” group was used as an organizational tool for both the Section and the

LogicalMicrolab (the server representation of a logical microlab) subjects, resulting in a series of problems and iterative changes to the database in an attempt to uncouple this association. As Wags was developed over multiple semesters by student developers, different practices were often put into use as they were learned about, and not necessarily from the beginning of Wags development. Thus, the database was originally created very organically – growing as things were needed. Recently, the database has undergone revision in order to organize and “prune” some of that organic growth, resulting in a database that adheres much more closely to standard etiquette and good practice.

As with most things, logical microlabs presented a particular difficulty for database storage. Programming and code magnet exercises can exist primarily as a list of attributes that can be stored directly into the database, such as “title,” “description,” and so on while linking to files necessary for the exercise to be run – files which can be stored in their entirety in the database as text. However, logical microlabs depend on a Problem class, an Evaluation class, a DisplayManager class, and additional classes which cannot be stored in the database as they exist entirely on the client. Thus, additional logic is necessary on the client in order to map an index stored in the database to the correct form of each class on the client for the construction of the Problem as required by Wags for display.

Each key is saved in the database as an int, which is then used by the client to retrieve the appropriate class. Only a single table is used in the database for every type of logical microlab, forcing the inclusion of an extra column used to inform the client which parameters are necessary for the specific exercise. For example, some exercises require a list of positions of nodes to correctly display the exercise, while other exercises do not. Attempting to access all attributes regardless of microlab type would result in exceptions for certain

microlabs which do not use a certain attribute. Therefore, the LogicalMicrolab table uses a “genre” column to indicate to the client which parameters need to be used for the microlab being returned from the server (the word “group” has other meanings for LogicalMicrolabs, so the less common term “genre” is used here). While this requires extra work on the client, it stops the proliferation of LogicalMicrolab tables and objects on the server.

The database is organized to facilitate use on the server. This is achieved by separating each table into a subject and a role. Each “entity” table for a subject has a corresponding class on the server, allowing all entity tables to be edited completely through the “save()” and “delete()” methods of the Model.php class. Changes in the rows occur by setting the variable of the object which corresponds to the correct column, and then re-saving the object. File tables are used during execution and compilation of programs.

Organizational tables are primarily used in static methods of the server-side classes, in multi-table queries.

All database interaction occurs through the PHP::PDO library, and all PHP::PDO library functions are hidden within static methods of the corresponding PHP class. Therefore, the majority of Wags developers can operate with a complete lack of knowledge of PHP::PDO or database interaction, instead simply calling the correct method within the PHP class and never writing a line of SQL. This allows the database to be exactly what it was meant to be: an easy to use database that allows creating, reading, updating, and deleting of entries in order to add functionality to Wags while only minimally adding to complexity.

Chapter 5

Classroom Testing and Outcomes

5.1 Early Testing in Fall 2011

Early development of Wags focused exclusively on logical and programming microlabs. Given the difficulty of inputting text on a tablet, testing Wags in the classroom relied on laptop or desktop computers at this stage. Preliminary Wags testing was concerned with usability and finding bugs that may result from classroom use with typical students. Two Appalachian State data structures classes participated in testing Wags at this level in the fall semester of 2011. One group tested Wags near the beginning of the semester, the other tested Wags near the end of the semester.

As the goal of early testing was usability, little quantitative data is available on these first uses of Wags. However, important qualitative data was obtained even in the first use of Wags. It quickly became apparent that students enjoyed logical microlabs and found them useful, but were critical of Programming Microlabs. A number of initial bugs were found that harmed programming microlabs' reception. Furthermore, many students cited difficulties such as lack of syntax highlighting, code completion, and other features they had become accustomed to in an integrated development environment (IDE) such as Eclipse.

The vast majority of bugs were removed by the second round of testing in the data structures course. While students seem to find Programming Microlabs less troublesome, the consensus remained that an online development environment could not compete with a full-fledged IDE.

5.2 Testing in Spring 2012

More rigorous testing took place in the following semester. Usability was still a main concern as the programming microlab interface continued to improve and logical microlabs had been updated. This testing occurred in data structures courses at two different universities – Appalachian State University and UNC Greensboro. Students in each course were assigned multiple logical and programming microlabs. Student completion data was recorded quantitatively, while student attitudes were recorded qualitatively using an attitude survey administered by

an external evaluator, SageFox Consulting. Thus, this round of testing recorded both students' response to Wags as a tool and their ability to use it successfully.

Table 5.1 – Testing in Spring 2012

Exercise Subject	% Correct (logical)	% Correct (program)
Traversals	87.2	92.8
BST Insert	89.3	82.1
Build BST	76.6	70.2
Build General Binary Tree	70.2	38.3
Radix Sort	-	73.1
Heaps	84.2	-
Minimum Spanning Trees	84.2	-

Both classes used Wags when studying binary trees – a common data structure in computer science. The logical microlabs on heaps and minimum spanning trees were only tested at Appalachian State. A summary of the quantitative information gathered from this round of testing is in Table 5.1.

In short, students tended to successfully complete logical microlabs more than programming microlabs. Particularly difficult concepts, such as building a general binary tree from two traversals, saw an even greater disparity between logical and programming microlab completion rates. This is to be expected as programming microlabs are more difficult than the corresponding logical microlabs [9].

The qualitative information was gathered via an anonymous survey. The survey had open ended questions as well as statements with which the student could agree or disagree on a 5 point scale (strongly disagree to strongly agree). To summarize the results, the students found the logical microlabs more useful and accessible than programming microlabs (88% found logical microlabs easy to follow, versus 69% for programming microlabs), as well as finding the logical microlabs more enjoyable. 76% of students believed they could apply what they had learned, though only slightly more than half believed using microlabs is more effective than a traditional lecture.

The open ended feedback generally praised the logical microlabs or suggested improvements for programming microlabs. The comments on liking the “interactivity; visual representation of data structures” and thinking “The programming Microlabs text editor is pretty basic” and “The instructions were a little difficult to understand” are representative of the feedback as a whole. The overall impression given by this round of testing was that

students enjoyed and thought that logical microlabs were useful, but that programming microlabs may not be the ideal candidate for an application phase exercise. It is worth noting that in this second round of testing still used laptops or a laboratory classroom, further suggesting that implementing programming microlabs on a tablet would be unacceptable.

During this time SUNY at Stonybrook conducted a semester of traditional lecturing in a data structures course to serve as a control group in order to gather data on student information retention rates with an experimental group using microlabs planned for the next semester.

5.3 Testing in Fall 2012

Given the general attitude towards programming microlabs expressed by students in all rounds of testing to this point, development began on a new type of microlab that could replace programming microlabs in the application phase of the Microlab Learning Cycle. Code magnet microlabs were the result, and were developed primarily over Summer 2012, with ongoing revisions and updates. Thus, testing at Appalachian State during Fall 2012 was primarily concerned with code magnet microlabs and tablet usage. Meanwhile, SUNY at Stonybrook used logical microlabs in a data structures course as the corresponding experimental group to the control group established in the previous semester.

The experimental group at SUNY was composed of 85 students who used logical microlabs while covering three subjects – binary tree traversals, building a binary search tree from a postorder traversal, and building a general binary tree given an in-order traversal and one additional traversal. Exam questions on these three subjects were used in both the Spring

2012 and Fall 2012 courses to check student knowledge retention. Table 5.2 shows an average of a 9% performance increase among the three subjects that used microlabs.

Table 5.2 – Testing results at Suny in Fall 2012

Semester	Traversals	Build BST	Build GBT
Spring 2012	88%	75%	89%
Fall 2012	97%	85%	97%
Gain/Loss	+9%	+10%	+8%

Testing at Appalachian State during Fall 2012 occurred in seven different classes. As code magnet microlabs were a main focus of testing, they were used in all seven courses: four CS1 courses, two CS2 courses, and a data structures course. Additionally, the data structures course used logical microlabs, allowing for the first implementation of the Microlab Learning Cycle using code magnet microlabs in place of programming microlabs.

Tablets were used exclusively in this classroom testing at Appalachian State. Thus, a main objective of this semester of testing was tablet usability. A 7” tablet was used as a “worst case” device as usability would only improve with larger screens. The tablet primarily being used was the Google Nexus 7, which costs around \$200 per unit (a few students used their personally owned iPads). Additionally, a tablet cart was purchased, allowing easy transportation of up to 40 tablets of varying size. The cart also serves as a charging station for the tablets when they were not in use. Occasionally during the Fall 2012 semester up to five classes would use the tablets on the same day, with four of the five classes immediately following each other. Even with such a length of continuous operation, the tablets were able to be used in each class without requiring recharging.

Both logical and code magnet microlabs use a drag-and-drop interface, with occasional pointing and clicking. When students were first introduced to code magnet

microlabs on a 7” tablet it was not uncommon for students to express frustration at the drag-and-drop interface working contrary to their expectations. However, these comments ceased completely after the second use of tablets, and when context highlighting was introduced to help students with moving objects on the screen comments largely ceased even during a student’s first use. Upgrading to a 9” or 10” tablet would only further alleviate the issue.

Like laptops, tablets rely on a Wi-Fi connection to run Wags. One day of testing had to be cancelled due to the network failing – however, this occurrence affected many classes that were not reliant of tablets nor laptops, and should not be seen as a shortcoming to using tablets to implement the Microlab Learning Cycle. In summary, students, teachers, and administrators alike found tablets a suitable environment for using microlabs within a non-laboratory classroom.

Table 5.3 – Testing results at ASU in Fall 2012

Topic	# sections	# students	% correct
Arrays: find max	3	57	82.5
Arrays: find average	3	50	98.0
Arrays: index of max	3	53	77.4
Arrays: swap min max	3	39	71.8
ArrayLists:Ins/Remove	2	36	72.2
ArrayList: Add	2	38	97.4
ArrayList: ArithSeries	2	33	84.8
Strings DoubleLetter	4	76	97.4
Strings: DoubleWord	4	73	98.6
Strings: Count Occur.	4	72	98.6
Strings: Palendrome	1	15	73.3
HashMaps: Insert	2	39	97.4
HashMaps: Remove	2	36	97.4
for: sumOddNumbers	1	16	87.5
for: isPerfectSquare	1	14	64.3
TOTAL		647	89.7

Given the extensive use of code magnet microlabs during the semester, an abundance of quantitative data was gathered. The amount of code magnet microlabs used differed

depending on instructor in the CS1 courses. The most prolific user of Wags resulted in 13 different microlabs activities; with the other instructors used nine microlabs across two course sections with the same instructor and a third instructor used six microlabs. In total, 647 microlabs were attempted among CS1 students with an overall completion rate of 89.7%. A more detailed breakdown can be found in table 5.3.

Each CS2 class used seven microlabs. Given absences, a total of 226 microlabs were attempted among the two classes, with an overall success rate of 81.4%. This is not surprising as CS2 activities were substantially more difficult than those for CS1, and CS2 students had less of an opportunity to acclimate to the Wags testing environment. Furthermore, the first exercise using code magnet microlabs had a 56.7% completion rate, significantly lower than every successive assignment. A more detailed breakdown of CS2 lab completion can be found in Table 5.4.

Table 5.4 – Testing results in CS2

CS2 magnets topic	# students	% correct
Stacks: Balance Checker	30	56.7
Stacks: Postfix Helper	32	96.9
Stacks: Postfix Evaluation	28	71.4
Big Integer: Add	38	71.1
Big Integer: Digit Multiply	33	90.9
Big Integer: Multiply	32	90.6
Queues: Josephus	33	90.9
TOTAL	226	81.4

The single CS3 (data structures) course used 30 logical microlabs and 7 code magnet microlabs. The data structures course was the first to use code magnet microlabs and provided some qualitative feedback concerning both the interface and use of code magnet microlabs. Some of the implemented suggestions include increasing the area that scrolls the screen when dragged, organizing magnets in a logical succession (though not one that reveals the answer), and more advanced debugging output which includes the constructed program

as text with line numbers in order to better use compiler output. Among logical microlabs the students had an average score of 95.7%, and 93.6% among code magnet microlabs. A more detailed breakdown can be found in Tables 5.5 and 5.6 as seen below.

Table 5.5 – Logical microlab testing results in a CS3 course at ASU

CS3 logical labs	# Labs	Max	Avg
Insert into a BST	2	20	20.0
Build a BST from postorder	3	30	29.6
Build GBT from traversals	3	30	29.6
Hashing - linear probe	2	20	20.0
Hashing - quadratic probe	2	20	19.8
Insert into a heap	4	40	36.5
Delete from a heap	2	20	17.9
Build a heap	4	40	37.1
Heapsort	2	20	16.7
Min Span Tree - Prim	3	30	30.0
Min Span Tree - Kruskal	3	30	30.0
TOTAL	30	300	287.2 95.7%

Table 5.6 – Magnet microlab testing results in a CS3 course at ASU

CS3 magnet labs	Max	Avg
Linear search - iterative	10	10.0
Linear search - recursive	10	9.8
Binary search - iterative	10	7.3
Binary search - recursive	10	8.8
Insert nodes into a BST	10	9.9
Build BST from Postorder	10	9.9
Build GBT from traversals	10	9.9
TOTAL	70	65.5 93.6%

Qualitative feedback was also gathered by SageFox Consulting for the Fall 2012 semester. Feedback was obtained through a series of focus groups wherein an external evaluator met with students in each class with the teacher and Wags administrators absent, allowing for more in-depth and honest responses than possible when obtained through a

survey. Overall, the feedback about the Wags system and code magnet microlabs was extremely positive. The resulting evaluation is shown unaltered:

Students who had been exposed to the greatest number of Microlabs both found them the most useful and liked them the best. They had much fewer complaints about having to log in, the interface, or how much time they took to complete. Students not using Microlabs as often found them less useful. This suggests that the more instructors utilize Microlabs, the more beneficial they are to students while requiring less and less time in class.

Students who were exposed to the logical Microlabs noted that it provided them more opportunities for practice. Many students commented on wanting to use Microlabs to help study for tests. Positive aspects about using Microlabs, including that they were hands-on, involved problem-solving without some of the problems inherent in coding such as typing and syntax errors, give the form of coding while allowing you to see the bigger picture, and provide feedback. Students commented that Microlabs "helped me figure out the logic behind the code without having to worry about syntax", "reinforce concepts that we are learning – doing a lot of programming so easy to use prefabricated code to see how it could be done", and "helps you recognize bugs in other people's code".

Multiple students suggested that more challenging Microlabs were needed and that one way to create these is to provide more "magnets" to include some which are not used. Students also saw a benefit to not being allowed, whether by their professor or the actual program, to guessing solutions multiple times as some did report guessing and checking until they arrived at the solution.

Interestingly, portions of this feedback conflict with earlier responses given by the data structures students, whose initial feedback included comments suggesting removing unneeded magnets and alleviating the grading system which penalized multiple attempts.

Upon incorporating those suggestions, the above feedback was given, which asks for more difficult problems and stricter grading! Many feedback-oriented changes have already been made, including an easy-to-use review functionality that allows students to revisit completed exercises, and another attempt at finding an appropriate number of superfluous magnets and a grading scheme that allows for multiple attempts without being so easily abused. In general, there is a substantial difference in the number of attempts between students who try to think through the problem and those that try random combinations of magnets, making schemes that punish guessers while pardoning other students plausible.

Data on student retention of course concepts was also gathered in the Fall 2012 data structures course. When tested on class material, exam questions on subjects covered by microlabs averaged 10% higher than exam questions on other subjects throughout the semester. More results are summarized in Table 5.7.

Table 5.7 – CS3 retention on exam subjects

Subject	Exam	Max	Avg
Hashing (2), Write iterative binary search code	mid 1	15	11.8
Insert BST, Build BST & GBT, insert heap, heapify, heapsort	mid 2	30	22.7
Traversals, heapsort, hash tables, MST, build binary trees	final	43	38.2
TOTAL		88	72.7

Chapter 6

Summary, Conclusions, and Future

Development

6.1 Summary of the Microlab System (Wags)

Work on Wags began during the summer of 2011. In the two years since, Wags has been used in four universities and in over 15 courses with thousands of recorded attempts on varying types of Microlabs. All recorded instances of student retention of concepts shows at least a 10% increase in scores on exam questions on materials covered using the Wags system. With the introduction of code magnet microlabs Wags has been received favorably by students at all levels. Recent feedback indicates that students want to use Wags more frequently and to use it as a study tool even when exercises are not currently assigned.

Wags is based on The Learning Cycle, an educational framework developed by Robert Karplus based on the work of Jean Piaget and constructivist learning theory. The Microlab Learning Cycle consists of logical microlabs being used to introduce students to a new concept in the exploration phase, a traditional lecture formalizing the concept in the Invention phase, and code magnet or programming microlabs to solidify understanding in the application phase. Wags was developed using GWT with the tablet environment in mind

to facilitate use in a traditional, non-laboratory classroom. Students and teachers have reacted positively to tablets, finding them a capable deployment device even when using a 7” screen. Throughout the two years of development multiple issues were uncovered and solutions were found. The iterative process of testing and improving the Wags system allowed for solutions to be implemented and refined continuously, a main cause of the general approval students and teachers have for the current implementation of Wags.

6.2 Success and Issues

The issues Wags faced can be classified into two broad categories: inherent issues due to the nature of the work, and implementation issues which can be resolved by careful development and design. Wags faced three primary inherent issues: “development stigma,” effects of user age, and correct instructional use of the system.

Development stigma is a term for how students react to an application that they are aware is in development or perceive as non-professional (such as a system being developed by a student). In such a scenario students are very quick to believe any problems they encounter have to do with the application and not their own effort. This can be a deterrent to student learning – they don’t realize they have an incorrect answer as they think the application is incorrect, and thus don’t challenge themselves to try to find a solution. The best cure for development stigma seems to be repeated exposure to the application and seeing other students completing assignments without similar complaints, and crafting a good first impression of the application in the student’s mind. The issue of development stigma was

most evident when testing programming microlabs; since their replacement with code magnet microlabs this issue has been less prevalent.

The second inherent issue Wags faces concerns usability and interaction with 7” tablets. Such a small screen can be difficult for older, non-traditional students to read and the drag and drop interface can be foreign and unwelcoming. Thankfully, Wags also works on larger tablets and desk-tops, but use of tablets in a classroom where laptops and desktops are not available can create problems for some older students. While the code magnet microlab interface has been refined to ease some of these difficulties with some success, the simplest solution is to have on hand a few alternative devices with larger screens for students with vision or dexterity impairments.

The third inherent issue is making sure the Wags system is used appropriately for an instruction environment. While Wags provides a suite of microlabs that are ready to be used without any alteration from the teacher, microlabs have never been intended to replace traditional programming assignments. That is, the Microlab Learning Cycle is the core of Wags, and the Microlab Learning Cycle is designed to be used in the classroom, not as homework. Replacing traditional homework with microlabs can result in too little of an emphasis on the importance of syntax and restricts student exposure to different development environments and program design decisions (many of which are inherently made by the available magnets). The solution to this issue is simply making sure teachers are informed of the correct use of Wags and of the dangers of relying too heavily on this new environment at the expense of traditional techniques, such as out-of-class program assignments.

Wags also faced some implementation issues. Mainly, these issues are screen economy and portability among different browsers. Use of a 7” tablet was critical to the development of the Wags system as it was designed to succeed in a “worst-case” environment, having decided that using Wags on a phone or anything smaller than 7” was simply implausible. This created many challenges that had to be addressed. Given the drag and drop nature of logical and code magnet microlabs, great care had to be taken to ensure the correct elements of the screen were being moved. A few approaches were found that alleviated much of the difficulty. First, creating a space specifically reserved for scrolling the screen made use of a touch screen much easier – students were much less likely to move the screen when they meant to move a magnet and vice versa. Surprisingly, increasing the size of the designated space was often the correct choice even when it meant removing space from magnets.

The second approach that helped with screen economy was using context-sensitive highlighting. When moving a magnet across the screen each magnet it passes over is highlighted, indicating that releasing the magnet at this time would result in placing the chosen magnet into the highlighted magnet. This improves the efficacy of a finger as a pointing tool. Furthermore, magnets highlight different colors depending on whether or not they can accept the magnet being dragged, as seen in Figure 6.1. Even when using intuitive colors (e.g., highlighting red for an illegal move) it is critically important to inform the

```
public class Student{  
    public int findSum(int[] intArray) {  
        int sum = 0;  
        for (int i = 0; i < intArray.length; i++) {  
        }  
        return sum;  
    }  
}
```

Figure 6.1 – A red highlight indicates an invalid magnet placement

students of this highlighting mechanic, especially in their first introduction. Using highlighting, rather than their finger, to determine where the magnet will drop becomes second nature, and informing students of such a mechanic early greatly reduced complaints about small screens.

Portability among different browsers was another implementation issue. While GWT does an admirable job at cross-compiling Javascript code for major browsers some idiosyncrasies still remain. The approach taken at Appalachian State University was to focus on correct implementation on the Google Chrome browser, which comes standard on Nexus 7 tablets, while repeatedly testing functionality on other popular browsers (such as Firefox, Safari, and Internet Explorer). Functional issues were quickly prioritized to be resolved, while for aesthetic issues the appearance on Google Chrome took precedence.

While Wags is an overall successful application, there are a few areas in which it seems to perform particularly well. The first is student knowledge retention. Wags continues to show the effectiveness of the Microlab Learning Cycle. This indicates that the Wags environment does a successful job of keeping itself from inhibiting the natural learning process of the Learning Cycle and constructivist learning theory. While students enjoy dragging and dropping magnets or the interactive nature of logical microlabs, some of this enjoyment may diminish as the tablets continue to become less novel and students become more familiar with the Wags system. Thankfully, regardless of how students view Wags in the future, the basis in sound educational theory will continue to help students retain knowledge more successfully.

Another success of Wags is in introducing more code reading into computer science curriculum. Reading code someone else wrote is an extremely common occurrence in the industry, but is moderately uncommon in computer science education. When code is read, it is often skimmed over by students unless it directly correlates to a current assignment, or is expressed in pseudocode. Code magnet microlabs expose the students to reading real code and trying to see the intent behind it in both magnet form and as text constructed from the student's magnet arrangement. An example is shown in Figure 6.2. Code magnet microlabs expose students to approaches to problems that the students may not have thought of originally, just as they would see in code used in their vocation. Furthermore, with the incorporation of purposefully incorrect magnets, students learn to approach all code critically and with a debugging mindset, rather than blindly accepting whatever code they read – all valuable skills in the workforce, and a niche in computer science education that Wags fills very naturally.

```
1 public class Student{
2
3
4     public int findSum(int[] intArray) {
5         int sum = 0;
6
7         for (int i = 0; i < intArray.length; i++) {
8             sum += intArray[i];
9         }
10
11         return sum;
12     }
```

Figure 6.2 – Code constructed from a student's submission and used for debugging

Finally, while Wags was designed with the Microlab Learning Cycle in mind and will be of most benefits to the students when used in accordance with the Microlab Learning Cycle, the modularity and flexibility of the system allows it to be easily adapted for additional uses. All microlab types are very valuable as review for upcoming exams, and Wags now provides functionality with that specifically in mind. Furthermore, code magnet microlabs can be used in place of quizzes or as a post-assignment assessment. Even if a teacher cannot devote the necessary time for the Microlab Learning Cycle for a certain

subject, making microlabs available for students to work on optionally provides additional resources for student learning. The potential for use goes even beyond the classroom, with the mode magnet microlab design showing particular potential as an assessment tool.

Wags was developed through a continuous cycle of testing, getting feedback, and updating. This process allowed Wags to become a useful, student and teacher friendly tool in computer science education. Furthermore, this process allowed early identification of less fruitful development paths (such as programming microlabs) as well as additional uses for Wags which may have otherwise gone unnoticed (such as the value provided in Review). Nonetheless, as this cycle continues more potential is found for Wags to grow and improve.

6.3 Future Work

Wags has benefited greatly from continuous and thorough feedback from students. Such feedback resulted in many cosmetic changes, usability changes, and the shift from programming to code magnet microlabs. While striving to improve, the Wags student experience is widely looked on with favor, accomplishing one of the main design goals of the system: make sure the implementation (Wags) doesn't get in the way of the process (the Microlab Learning Cycle).

Despite the many times Wags has been used, more feedback from instructors is needed. Thus, much of the future work is oriented around making Wags as streamlined and welcoming an experience for instructors as possible. This means re-working the administrative user interfaces, enabling easy modification of logical microlabs, and

establishment of a Microlab Repository to further expand the library of available microlabs by creating an open source collection of microlabs with established usability and quality.

Currently, administrative operation of Wags occurs in a few areas throughout the application. Most administrative duties are bound to tabs only administrators can access on the Editor page. However, in order to create new magnet exercises the instructor has to navigate into a magnet problem and access the creation interface from there. Thus, administrative duties are unnecessarily bound to two Wags areas – the Editor and Code Magnet Microlab pages. With logical microlab modification currently in development, a shift is being made to create a completely separate page for administrators with no dependence on any other area of the Wags system. This will not only be a change of location but also implementation of the administrative interface. In short, many implementation changes center on visibility of information on the screen and a movement into easier to use widgets. With these changes, the Wags administrator experience can become as intuitive as the student experience.

A mechanism currently exists which allows instructors to quickly and easily create their own code magnet microlabs, and a mechanism is in development to allow instructors to modify existing logical microlabs to meet their particular needs. Both of these mechanisms allow educators to have access to their own newly created microlabs, but restrict access to those microlabs from all others in order to ensure quality microlabs on important subjects. Incorporating a process to upgrade personal microlabs to public access would allow for community sourced microlab expansion that dynamically meets the needs of instructors. This repository could have incentives for quality microlab creation. Mainly, this repository would be a scheme of changing permissions of administrators and privacy of microlabs

within the hosting database, incurring little overhead expense. As Wags use continues to grow, creating an external repository would be beneficial for distributed implementations of the Wags system on various servers.

As the user and instructor experience become even more user-friendly and even less of a distraction from the Microlab Learning Cycle, an additional area for future work becomes apparent. Currently, students are able to work in groups as they please, though such work usually consists of multiple students working on a single account on a single device. Implementing Real-time Collaboration, as seen on Google Drive documents, would make group work a much more interactive experience to the individual students. Allowing multiple students to work simultaneously on a single exercise on multiple tablets would garner the benefits of group work without sacrificing any of the creation of knowledge aspect of the Microlab Learning Cycle, a danger that occurs often in group work where one student has physical control of the exercise. However, such a change would require substantial rework of the Wags environment and would be difficult to implement.

As students have been exposed to code magnets and become more familiar with the interface they have expressed interest in being given more difficult code magnet microlabs and more control over the creation of the solution. In order to facilitate this, development is nearing completion on an “Advanced Code Magnet Microlab” structure, which allows students to create their own magnets from a limited pool of options. During these advanced problems students will be able to use a “Magnet Maker” to construct some magnets themselves. Such magnets are typically control structures, allowing students to specify what type of comparisons should be evaluated.

For example, instead of offering two different “If” magnets with different comparisons (perhaps less than versus less than or equal to), the student would be told they could create a single “If” magnet. When constructing this magnet, he or she would have multiple options for the type of comparison. This gives the student more control over the creation of the solution as well as enhancing difficulty through providing more options without encroaching on screen space like multiple magnets of the same type would. Each type of control structure would have a limited number of magnets the student could create, in efforts to guide students into a correct and understandable solution. If students create the maximum amount of magnets they can delete some of the created magnets in order to make new ones. Thus, the bank is not a limit on the number of times magnets can be created, but on the maximum number of created magnets (per type) that can exist at one time.

Wags has an established history of real benefit to computer science education. Growing on this foundation allows for even greater gains to take place. Moving Wags from the private to the public sphere with a Microlab Repository, streamlining the instructor experience, and creating robust group work opportunities only further brightens Wags’ already promising future. Such a future is, however, only a means. A means to an end of improved computer science education rooted in student understanding and retention, which will itself lead to a brighter future for the technological industry and the world as a whole.

References

- [1] “AMI School Standards”. Association Montessori Internationale-USA (AMI-USA). Retrieved March 20th, 2013. <http://www.amiusa.org/products-page/featured-products/ami-school-standards-2/>
- [2] Awwad, A. and Aqeel, A. 2013. Piaget’s Theory of Learning. *Interdisciplinary Journal of Contemporary Research in Business*, 4 (9). 106-129. Retrieved March 16, 2013, from Appalachian State University. http://appencore.wncln.org/iii/encore/articles/C_Spiaget_Orighresult?lang=eng&suite=ncapp
- [3] Benaroch, R. 2012. Piaget Stages of Development. Nov 6, 2012. Retrieved March 15th, 2013, from WebMD. <http://children.webmd.com/piaget-stages-of-development>
- [4] Bowles, K. 1978. A CS1 course based on Stand-Alone Microcomputers, *SIGCSE Bulletin*, 9 (1). 125-127
- [5] Fuller, R. 2003. “Don’t Tell Me, I’ll Find Out” Robert Karplus – A Science Education Pioneer. *Journal of Science Education & Technology*, (12) 4. 359-369. Retrieved March 16, 2013, from Appalachian State University. <http://ehis.ebscohost.com/ehost/detail?sid=34aabe0f-d2f6-4bdb-894f-495f4431b809%40sessionmgr15&vid=1&hid=5&bdata=JnNpdGU9ZWZWhvc3QtbGl2ZQ%3d%3d#db=ehh&AN=16859028>
- [6] Karplus, R. 1977. Science Teaching and the Development of Reasoning. *Journal of Research in Science Teaching*, 14 (2). 169-175.
- [7] Kelleher, C. 2006. Motivating Programming: using storytelling to make computer programming attractive to middle school girls. *Carnegie Mellon*. November 2006. Retrieved from Carnegie Mellon on April 24th, 2013. http://www.cs.cmu.edu/~caitlin/kelleherThesis_CSD.pdf
- [8] Kliebard, H. 1987. Constructing a History of American Curriculum, in *Handbook on Research in Curriculum*. 157-184.
- [9] Kurtz, B., et. al. 2013. Evaluation of Microlabs in Computer Science Education. Technical Symposium on Computer Science Education (SIGCSE), Denver, CO.

- [10] Kurtz, B, et. al. 2012. Developing Microlabs Using Google Web Toolkit, Technical Symposium on Computer Science Education (SIGCSE), Raleigh, NC
- [11] Maloney, J., et. al. 2010. The Scratch Programming Language and Environment. *ACM Transactions on Computing Education*. Retrieved April 24, 2013, from Massachusetts Institute of Technology.
<http://web.media.mit.edu/~jmaloney/papers/ScratchLangAndEnvironment.pdf>
- [12] Obama, Barack. The White House, Office of the Press Secretary. (2013). *Remarks by the president in state of union address* Washington, D.C.: Retrieved April 10th, 2013 from <http://www.whitehouse.gov/state-of-the-union-2012>
- [13] Schonfeld, E. 2011. Estimate: 90 Million U.S. Tablet Users By 2013; iPads Drop to 68% Share. *TechCrunch*. Retrieved March 15, 2013.
<http://techcrunch.com/2011/11/21/estimate-90-million-u-s-tablet-users-by-2014-ipads-drop-to-68-share/>
- [14] Schwartz, D. L., and Bransford, J. D. 1988. A Time for Telling. *Cognition and Instruction*, 16 (4). 475-5223
- [15] The Marmoset Project. Retrieved March 20, 2013, from The Marmoset Project, The University of Maryland. <http://marmoset.cs.umd.edu/>
- [16] Ultanir, E. 2012. An Epistemological Glance at the Constructivist Approach: Constructivist Learning in Dewey, Piaget, and Montessori. *International Journal of Instruction*, 5 (2). 195-212. Retrieved March 15, 2013, from Appalachian State University: <http://ehis.ebscohost.com/ehost/detail?sid=7840aeb4-987a-41de-bf64-740595c91251%40sessionmgr104&vid=1&hid=115&bdata=JnNpdGU9ZWhvc3QtbGl2ZQ%3d%3d#db=ehh&AN=79310661>

Using the Web-Based Automatic Grading System

Written By:

Philip Meznar
meznarp@email.appstate.edu

Supervisors:

Barry L Kurtz
Rahman Tashakkori
{blk,rt}@cs.appstate.edu

Funded in part by:

Developing Software and Methodologies for
eBook/Browsers to Enhance Learning (NSF
DUE 044572)

Transforming Computer Science Education
with Microlabs (NSF DUE 1122752)

Table of Contents

Preface	3
Introduction	4
Getting Started	5
Logging In	5
Viewing the WAGS Editor Screen	6
Navigation Bar	7
Registering Students	7
Creating Exercises	8
Exercise Structure	8
Class Implementation	8
HelperClasses	9
Uploading an Exercise	10
Editing Exercises	11
Administrative Class Files	11
Exercise Actions	12
Using the CodeEditor	13
Reviewing Submissions	14
Logical Microlabs	15
Accessing	15
Reviewing	16
Code Magnet Microlabs	16
TroubleShooting	16
Glossary	18

Preface

This guide serves as introductory material on the Web-Based Automatic Grading System (WAGS). WAGS is a multifaceted rich web application designed to help teach computer science concepts through a series of logical, programming, and “magnet” Microlabs.

This guide is written primarily for Computer Science (CS) instructors who are evaluating or planning to use WAGS. As such, basic understanding of some computer science terms is expected (such as *compilation*, *class*, *binary tree*), while terms specific to the WAGS system will be defined in the text and referenced in a glossary at the end of the manual.

An initial reading of the manual should help orient the reader, while subsequent viewings can use the manual as a reference rather than an entire procedure. A troubleshooting guide is included at the end of the manual addressing some of the more common error messages. As this is the first iteration of the WAGS project, feedback is welcomed at pmezmar@gmail.com.

The WAGS system uses "sections" to group together accounts (e.g., a group of students for a class like DataStructures). These sections allow each administrator to view WAGS as a system solely for their academic class, although it may be being used by multiple campuses or classes simultaneously. Throughout the remainder of the text, "section" always refers to a group of accounts (usually an academic class, like CS3481), while "class" refers to a class in a programming context (such as "Node.java is a *class*").

For this manual, terms listed in the glossary are printed in **bold**, while words in *italics* have an action associated with them (e.g., "click", "fill in", etc). Furthermore, any words directly relating to code will be in the font `courier new`.

While reading this manual, it is strongly suggested you navigate to <http://cs.appstate.edu/wags> and navigate the website while reading, although the manual does include some images.

Introduction

WAGS is an online submission tool meant to streamline **Microlab** execution in a classroom setting. A Microlab is a short assignment aiming to quickly test student understanding, often in a lecture setting. As such, Microlabs should be completable in 5-15 minutes, and provide useful feedback to the student in the case of an incorrect attempt. Multiple attempts are supported and the instructor may extend the due date in order to let students finish outside of class.

WAGS handles three different types of Microlabs: **logical**, **programming**, and **code magnets**. Logical Microlabs are visual in nature, and do not include writing any code. They tend to more focused on concepts than coding, such as tapping on the nodes in a binary tree to perform a certain traversal. These are usually followed up with either programming or code magnet Microlabs. In programming Microlabs the students would be responsible for writing a method to perform the certain traversal, with given input and return type. Code Magnets are similar, except excerpts of the code are provided for students who must then arrange and nest the different segments appropriately. Often the outcome of Code Magnets and Programming Magnets will be an identical program, only created through different means.

Overview of the WAGS System

This manual covers the WAGS system from an administrative (i.e., instructor's) perspective. As the administrator's abilities in WAGS is a superset of that of a normal user, such an introduction should adequately prepare the instructor for any events on the user side. The administrative tasks being covered includes:

1. Logging in to the System
2. Viewing the WAGS Editor Screen
3. The Navigation Bar
4. Registering Students
5. Creating Exercises
6. Uploading Exercises
7. Editing Exercises
8. Using the CodeEditor
9. Reviewing Submissions
10. Logical Microlabs
11. Code Magnet Microlabs

Getting Started

WAGS was developed using Google Web Toolkit (GWT). GWT compiles Java applications to JavaScript customized for all main browsers. Choose your favorite browser and navigate to <http://cs.appstate.edu/wags>. You should have been provided with a personalized username along with this document. Figure 1 shows you the WAGS login screen.



FIGURE 1. The WAGS login screen located at <http://cs.appstate.edu/wags>, displayed in a Chrome browser

Logging In

To login, simply enter your account name and the initial password "password". Then, click either the *Programming*, *Logical* or *Magnet* button. The *Programming* button will take you directly to the **WAGS Editor**, where the programming Microlabs and administrative functions take place. The *Logical* button takes you to the **Logical Microlab Index Screen**, where Logical Microlabs are displayed. The *Magnets* button takes you to the **Code Magnet Microlab Index Screen**, where you can select assigned magnet Microlabs to complete.

On your first login, no Logical or Code Magnet Microlabs will be visible. Thus, click the *Programming* button. A window will pop up asking you to change your password and verify it. WAGS will accept any string as a password, it is up to you to make sure the password is suitably strong. After entering and re-entering your new password, click *Close*, and you will receive a notification along the top of your browser window confirming your password change.

Viewing the WAGS Editor Screen

The WAGS editor screen is divided into portions. At the top is a navigation bar, which is discussed in the next section. Underneath the navigation bar the screen is split into a left and right half. The left half is the **CodeEditor**, where all code writing occurs for programming Microlabs.

The right half (called the **TabPanel**) has five tabs. The currently selected tab determines what the right half of the screen displays. An administrator has access to the *FileBrowser*, *Review*, *Exercises*, *Students*, and *Desc* (Description) tab. A student only has access to the *FileBrowser*, *Review*, and *Desc* tabs. Figure 2 displays the right half of the screen in closer detail.

Tab Overview

FileBrowser. The FileBrowser tab displays your files being used with the WAGS system. This will be examined more closely in subsequent sections.

Review. The Review tab takes the place of stdout (standard out) for the Microlabs. After a submission the Review tab is automatically selected and shows output for the lab.

Exercises. The Exercise tab handles all administrative tasks concerning exercises. This includes adding, deploying, and reviewing exercises.

Students. The Student tab allows for registering of students and changing of account passwords.

Desc. The Desc tab stands for description, and can display a Portable Document Format (PDF) file for each exercise, outlining the goals for the student.

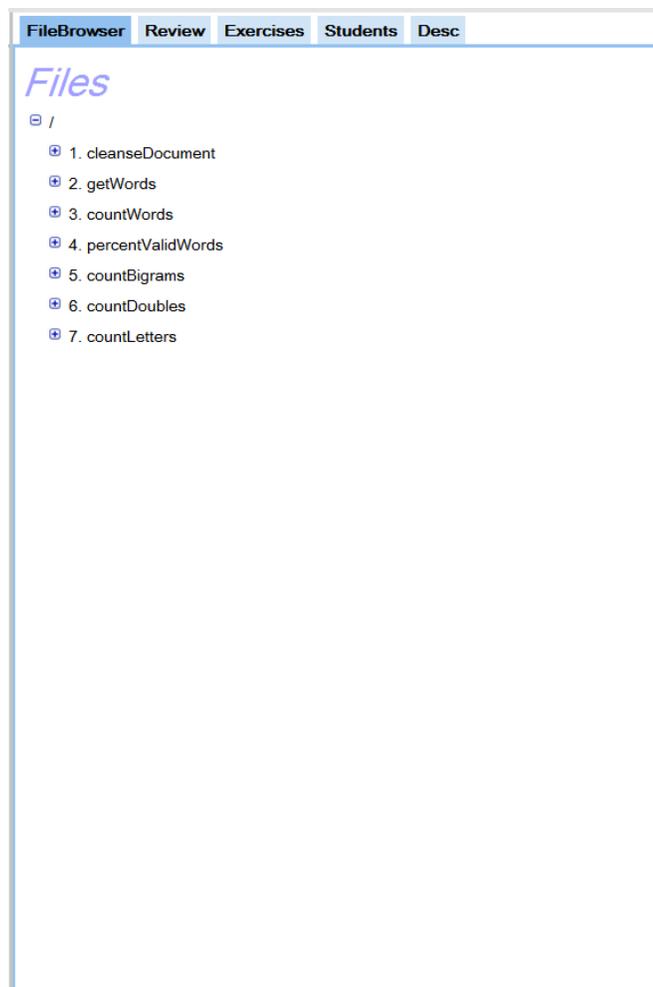


FIGURE 2. The TabPanel, with FileBrowser currently selected and showing a directory of files

The Navigation Bar

The Navigation Bar provides a constant interface among all areas of Wags in efforts to simplify navigation among different aspects of the system.



Hello, TutorialAdmin! [Editor](#) [Magnets](#) [Logical](#) [Logout](#)

FIGURE 3. The Right side of the navigation bar with anchors to the various sections of Wags

On the left side of the Navigation Bar is the Wags name, while on the right side links are provided to the different sections of Wags. The Editor link takes you to the editor screen, the Magnets link takes you to the Magnet Microlab Index Screen, and the Logical link takes you to the Logical Microlab Index Screen.

A link to Logout is also provided. Regardless of what portion of Wags is currently being used, the Navigation Bar remains at the top to provide a consistent navigation interface.

Registering Students

TextBox Registering

Fill in the *First Name* and *Last Name* textboxes for each student. This creates an account for each First Name, Last Name pair of the form LastName.FirstName. This method of registering is only recommended for very small class sizes.

CSV Registering

Alternatively, an administrator can upload a Comma Separated Value document with entries of the form " Student Last Name, Student First Name, Student E-mail address". This creates accounts for each student with their e-mail address as the identifier, and the initial password "password." Each student will be asked to change their password on their first log in.

For example, the entry "Smith, Jane, janesmith@gmail.com" would create the account "janesmith@gmail.com" with the password "password".

Password Changing

The Student Tab also has functionality for changing a student's password. To do so:

- 1) Select the User Account from the *ListBox*
- 2) Enter the new password
- 3) Re-enter the password in the second textbox. (Note: Set the password to "password" to allow a student to change his/her password next login)
- 4) On successfully changing a password, a green notification bar will flash at the top of the screen.

Creating Exercises

The WAGS system comes with a collection of Programming Microlab files, a suite of Logical Microlabs built directly into the system, and a few Code Magnet example Microlabs. Additionally, an administrator can upload their own Programming Microlabs to add to the system. This section deals specifically with creating your own Programming Microlabs.

Exercise Structure

Each Programming Microlab is composed of at least three classes. The mandatory three classes are referred to as the **SolutionClass**, **SkeletonClass**, and **TestClass**.

The **TestClass** contains the `main()` method, and is the driver for checking correct operation. The **SkeletonClass** is the class the students can see and alter - one is created for each account within the section. The **SolutionClass** should be identical to the **SkeletonClass**, but with the methods filled in correctly by the instructor. The **TestClass** can then call and compare the methods in the **SkeletonClass** and the **SolutionClass**.

Class Implementation

TestClass. In order to test for correct operation, the **TestClass** must read in one variable from the command line. This variable will reside in `args[0]` in the main method. The **TestClass** should maintain knowledge of the student's correct completion of the Microlab in a `boolean` variable, `true` meaning the student correctly implemented the necessary methods.

The last printed statement from a **TestClass** (to `System.out`) should reflect the status of the `boolean`. If the `boolean` is `true` (i.e., the student completed the assignment), it should cause `System.out.println(args[0])` to be called. Otherwise, `System.out.println("Incorrect. Try again")` should be called. This is mandatory for automated exercise reviewing.

SkeletonClass. The skeleton class should consist of mostly empty methods that the student is responsible for implementing. The WAGS system includes functionality to further limit the editable portion of the class. WAGS relies on two special tags to do this. The editable text must lie in between the tags `//<end!TopSection>` and `//<end!MidSection.`

It is very important to have the correct spelling of these tags, including no space between the slashes and the opening angle bracket. Anything above the first tag and anything below the second tag will not be editable to the student. At the top of the **CodeEditor** there is the word "*Code*" which is clickable. Clicking on *Code* will cause the **Review** tab to be selected, and the entirety of the class will be shown (but not editable) as seen in Figure 4 overleaf.

```

5.countBigrams/skeleton Save Code Submit
Dictionary countBigrams(String[] words)
{
    // Returns a Dictionary of letter pairs that appear in all the
    // words and their associated counts.
    // For example, "hello" contains the bigrams "he", "el", "ll",
    // "lo".
    return null;
}

FileBrowser Review Exercises Students Desc
// Michael Kepple
// February 5th, 2012
package countBigrams;
import java.util.Iterator;
import java.util.Scanner;

public class DictionarySkeleton
{
    public StringBuffer cleanseDocument(StringBuffer buffer)
    {
        // This method will convert all upper case letters to lower
        // case and then delete all characters except
        // lower case letters and spaces.
        StringBuffer result = new StringBuffer("");
        return result;
    }

    public String[] getWords(StringBuffer buffer)
    {
        // This method returns an array of words from the buffer.
        String[] result = new String[0];
        return result;
    }

    public Dictionary countWords(String[] words)
    {
        // This method returns a Dictionary where the pair (word, count)
        // appears for all words in the
        // document.
        return null;
    }

    double percentValidWords(Dictionary wordCount)
}

```

FIGURE 4. The editable portion to the left, the entire class to the right in the Review Tab

SolutionClass. The Solution Class should be identical to the Skeleton Class, but with each method filled in correctly. It can also remove the tags from the Skeleton Class, but this isn't necessary.

HelperClasses

For some Microlabs, the three class structure may prove insufficient. For example, for Microlabs concerning binary trees, it may be helpful to have access to a `Node` class. There are two ways to implement such a class, referred to as a **HelperClass**. A **HelperClass** can be implemented as an inner class residing within the **SkeletonClass** (so students can view it via the *Code* button), or as a separate class entirely (the suggested method).

Inner Class. To implement the **HelperClass** as an **Inner Class** requires two additional stipulations

- All classes must have a `package` statement, and belong to the same `package`
- All non-skeleton classes must import the inner class within the skeleton

For example, if you have the `package` `traversals` and the class `Node` within the `TraversalSkeleton` class, you must include the line:

```
import traversals.TraversalSkeleton.Node;
```

Outer Class. Implementing a **HelperClass** as an **Outer Class** is simple, and will be covered in the **Uploading an Exercise** section (below). This implementation of a **HelperClass** is visible to the student in the **FileBrowser**, but not editable.

Uploading an Exercise

After a Programming Microlab is created, it is easy to upload and is automatically available to your students. On the **Editor Screen**, click the *Exercises* tab. There are many sections under this tab, but we are only immediately concerned with the "Uploading" section.

To upload an Exercise:

1. Type the Exercise Name in the *Exercise Name* textbox
2. Select the appropriate SolutionClass, SkeletonClass, and TestClass files using the corresponding FileUpload Widgets (see Figure 5)
3. Optionally, you can include an Opening and Closing Date - students will only be able to submit the exercise in the period of time in between the Open and Close Date. [Note: The exercise will be visible outside of these dates, but will not accept submissions.]
4. Click the *Add Exercise* button
5. A notification will appear at the top of the screen confirming your success or any errors that may have occurred

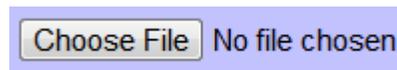


FIGURE 5. A FileUpload Widget displayed in Chrome

To upload an Outer Class Helper:

1. Select the appropriate exercise from the Exercise Actions ListBox
 - The exercise must already be created with the mandatory three classes following the steps in the above "To upload an Exercise" section
2. Use the Helpers FileUpload Widget to select the appropriate Helper Class
3. Click the *Upload Class* button
4. A notification will appear at the top of the screen stating your success or any errors that may have occurred
5. Repeat for any subsequent Helper Classes

To upload a Description:

1. Select the appropriate exercise from the Exercise Actions ListBox
2. Using the Description FileUpload Widget, select the appropriate PDF file
3. Click "Upload Class"
4. A notification will appear at the top of the screen stating your success or any errors that may have occurred
5. Any subsequent Descriptions will overwrite the previous description for the exercise

A Note on Naming Exercises. Wags provides a lot of flexibility in naming your exercise. Nonetheless, there are some good practices to consider. The FileBrowser will display all Exercises in a directory structure, with each “/” in a file name denoting a new directory. Thus, if a student is enrolled in multiple courses each using Wags, naming each exercise “<Classname>/<ExerciseName>” can provide helpful clarity in navigation for the student.

This option is not enforced by the system as currently most students are only enrolled in a single class using Wags, and then this naming system provides unnecessary layers to navigate in the FileBrowser.

Editing an Exercise

Once an exercise is uploaded, you may find that you want to make some changes to it. Rather than have to edit the original files and re-upload the entire exercise, WAGS allows you to edit your exercise within the WAGS system. This is mainly done through the **CodeEditor** and the **FileBrowser** tab.

Administrative Class Files

As an administrator, you are able to alter any file in your section. Where a student can only alter the editable portion of a SkeletonClass, an administrator can alter the SkeletonClass, SolutionClass, TestClass, and any uploaded HelperClasses (see Figure 6).

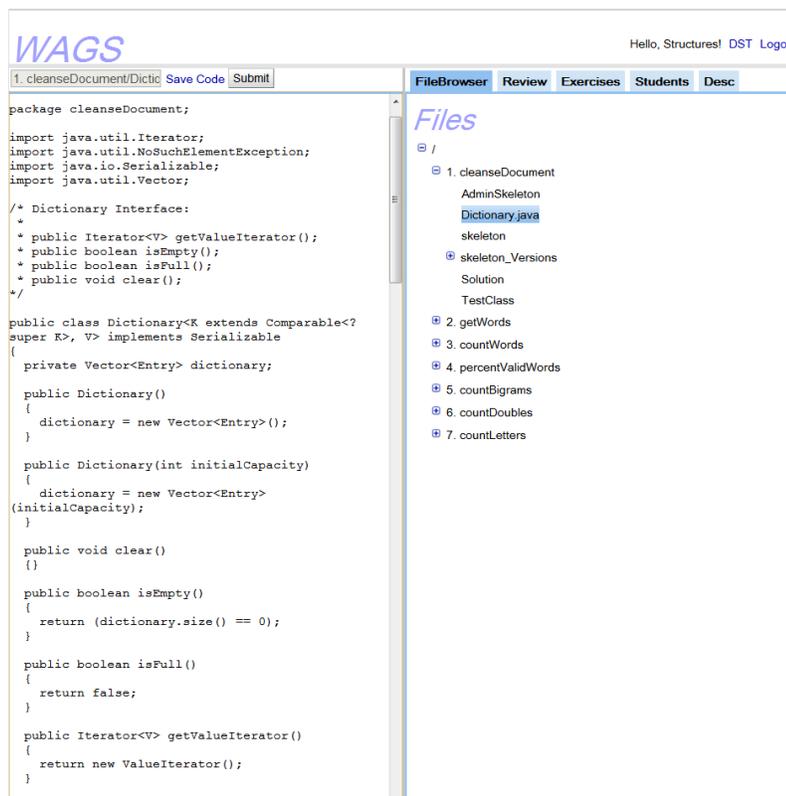


FIGURE 6. An administrator is capable of editing the Dictionary.java helper class

Skeleton Classes. An administrator has two different SkeletonClasses: a regular skeleton class ("skeleton" in Figure 5) and an **AdminSkeleton** class. The regular SkeletonClass works exactly like it does for a student, and can be used to make sure the exercise works on the WAGS system. The AdminSkeleton class cannot be submitted, but can be used to revise the skeleton class all students receive.

Committing Changes. Every change to an exercise should be accompanied with a test of the edited exercise. To ensure this, the changes to an exercise are only committed when the Administrator submits the exercise. Submitting an exercise is covered in the section **Using the CodeEditor**.

CAUTION: Every time you submit an exercise as an Administrator, all files for that exercise are updated. Thus, *make sure your most recent submission compiles.*

Programming Microlab Actions

In addition to editing the files of an exercise, there are a few other actions you can take. These are found on the *Exercises* tab underneath the heading **Programming Microlab Actions**. In addition to uploading HelperClasses and Descriptions (covered in the section **Uploading an Exercise**) there are four buttons you can use to change your exercise. The first step for any of these buttons is

- Select the appropriate exercise from the *ListBox* at the top of the **Exercise Actions** subsection

Once the appropriate exercise is selected, you can use any of the four buttons:

- *Review*: Displays information about each student's attempts to complete the chosen exercise.
- *Update Student Skeletons*: Replaces student SkeletonClass text with the administrators AdminSkeleton contents.
- *Toggle Visibility*: Toggles whether or not the student can see the exercise in their **FileBrowser**.
- *Delete*: Deletes an exercise and all associated information, including all files and submissions for that exercise. Note: This cannot be undone.

Using the CodeEditor

The **CodeEditor** is the heart of the WAGS System Programming Microlab functionality. It allows for writing of code that is then compiled on a server and run. The **CodeEditor** allows you to view, alter, and submit files for compilation.

The CodeEditor Header

At the top of the **CodeEditor** is a panel. This panel displays the current filename, and has links to *Save* your file, view the complete *Code* for the class, and a button to *Submit* the exercise.

Saving. The WAGS system has multiple safeguards to keep students from losing their code. The WAGS system

- saves altered code every 30 seconds
- saves each submission for an exercise
- allows the student to save by clicking the *Save* link

In order to save after each new submission, the **FileBrowser** creates a `skeleton_Versions` subdirectory for each exercise. Here, each submission is saved with incrementing version numbers. In order to work on an old skeleton version, select the appropriate file under the `skeleton_Versions` directory. This copies that text into the **CodeEditor**, meaning you can alter the text without losing the skeleton you are working on.

The Code Button. Clicking the *Code* button allows a student to see the entire SkeletonClass rather than just the portion the student can edit. *Note:* This is a very useful functionality, but often overlooked by students. It should be emphasized at the introduction of the WAGS System.

Submitting an Exercise. A student or administrator can test their current SkeletonClass simply by clicking the *Submit* button. After submitting an exercise, the student will receive one of three notifications:

- *Failed to Compile:* A red bar will flash across the top of the screen with the words "Failed to Compile". This counts as a failed submission for the student.
 - The review panel will display the error message
- *Incorrect, Try Again:* A yellow bar will flash across the top of the screen with the words "Incorrect, Try Again". This counts as a failed submission for the student.
 - The review panel will display the Standard Out from the exercise.
- *Correct!:* A green bar will flash across the top of the screen with the word, "Correct!". This counts as a successful submission for the student.
 - The review panel will display the Standard Out from the exercise.

Reviewing Submissions

Individual Exercise Review

As a submission tool, WAGS keeps track of submissions and completed assignments for easy access by the instructor. Reviewing submissions can be done by:

1. Clicking the *Exercise* tab
2. Selecting the appropriate exercise from the drop-down Listbox
3. Clicking the *Review* button

A grid will appear listing all students in alphabetic order by username, along with the file they most recently submitted for the exercise, the number of submissions used, whether or not the student completed the exercise successfully, and their partner (if they had one – otherwise, this field is left blank).

Comprehensive Section Review

Wags also allows for reviewing all grades for an entire section with a click of a button. At the bottom of the **Exercise Tab** there is a heading entitled “Comprehensive Review”, with a lone *Review* button underneath it. Clicking this button will download a CSV report of all grades for the semester. Importing this document into Excel will show a grid with student names running down the left side, and exercises along the top.

Each cell indicates the number of attempts a student made, and whether or not they successfully completed the exercise. Note: Once a student successfully completes an exercise, the number of attempts for that exercise no longer continues to increment. This way, students can re-attempt any completed exercise for help on the next one without any academic penalty.

Logical Microlabs

The WAGS System comes with a pre-installed suite of Logical Microlabs. These Microlabs are clumped into six different groups, although more will be added later:

- Traversals
- Insert Nodes
- Build Binary Search Tree
- Build Binary Tree
- Radix Sort
- Heaps

There are usually 3-6 Microlabs for each section. Making certain Microlabs visible (and the rest invisible) is done by clicking the appropriate *checkboxes* followed by clicking the *Set Exercises* button on the *Exercises* tab **Logical Microlab Actions** form.

Accessing the Logical Microlabs

Access to the Logical Microlabs is granted in two ways. One is from the Login screen, where the user could click *Logical* instead of *Programming* to be taken to the **Logical Microlab Index Page**. Alternatively, the user could click the *Logical* link text to the *Logout* link in the header of the **WAGS Editor Screen**, which also takes the user to the Logical Microlab Index Page.

Logical Microlab Index Page. This page lists the individual Microlabs available to the user (e.g., there are six Traversals Microlabs belonging to the Traversals group checkbox on the *Exercises* tab). Clicking the *Attempt* button next to the name of each Microlab takes you to the **Logical Microlab Problem Page**, where the Microlab can be attempted.

Logical Microlab Problem Page. This page lists the instructions for the problem, has a *Back*, *Reset*, and *Evaluate* button, as well as the **Problem Area** where the user uses a mix of drawing and drag-and-drop utilities to complete the Microlab (see figure 6).

- *Back* button: Returns you to the **Logical Microlab Index Page**
- *Reset* button: Returns the **Problem Area** to its initial orientation
- *Evaluate* button: Checks the proposed solution, resulting in feedback informing of a correct solution, or hints in how to fix the incorrect solution

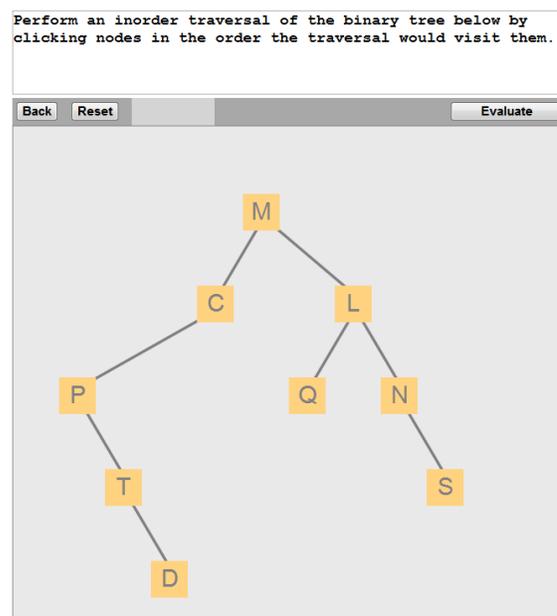


FIGURE 6. The Logical Microlab Problem Page

Reviewing Logical Microlabs

The process for reviewing Logical Microlabs is very similar to that of reviewing Programming Microlabs.

- 1) Navigate to the **Editor** screen, and click on the *Exercises* tab.
- 2) Select the appropriate Logical Microlab from the drop-down ListBox underneath the heading **Logical Microlab Actions**.
- 3) Click the *Review* button.

A grid similar to that for programming Microlabs will appear, but only with fields for the Username, whether or not the submission was correct, and the number of attempts. Logical Microlab grades are also reported in the Comprehensive Review.

Code Magnet Microlabs

Code Magnet Microlabs are assigned in the same manner as Logical Microlabs. Currently, very few Code Magnet Microlabs are supplied with Wags. Code Magnets are assigned “Groups” to help ease navigation, and each Administrator automatically has access to the “Default” group. For now, “Find Sum of an Int Array” is the only completely supported problem, to be used as an example.

In future updates, Administrators will be able to create and assign their own code magnet problems, using a similar three class structure as seen in Programming Microlabs. However, the “Student” class will be built using Code Magnets.

Code Magnets currently can be reviewed individually, but are not part of the Comprehensive Review.

TroubleShooting

WAGS uses a Notification system to inform the user of different events. Notifications take the form of a bar flashing across the top of the screen with some informative text. Usually, this bar will be green signifying the successful completion of the attempted task. In the event that a red bar flashes instead, this section gives more information into the potential problem at hand. The error messages are sorted alphabetically.

Error messages may also appear in the *Review* tab when occurring during the submission process.

"Administrate file error while writing: [msg]" : A file was incorrectly written during compilation . [msg] gives more specific information.

"bad password" : Password does not match user account.

"Compilation Error" : The code had a syntax error - most likely in the submitted code. The review panel should print a stacktrace for debugging.

"Dates are not in the correct format" : Unrecognized date format - use format "DD monthName YYYY HH:MM:SS".

"Error in matching language for compilation" : The Microlab cannot be identified as either Java or Prolog - check file extensions on Exercises class files.

"Error in matching language to execution" : The Microlab cannot be identified as either Java or Prolog - check file extensions on Exercises class files.

"Error in matching solution file extension" : The solution file is neither a Java nor a Prolog file.

"Exercise not currently visible" : Attempted to submit a file for an invisible exercises. Usually means the exercise has expired. If prematurely invisible, can be made visible on the **Editor** screen underneath the *Exercises* tab.

"File name needed." : Tried to save a file without a file name. Legacy error - should not appear.

"Login failed. Check username and password" : Self explanatory

"Must be logged in as administrator" : Most likely, session expired when trying to upload an exercise. Log in again.

"Must be logged in to save a file." : Tried to save a file after session has expired. File shouldn't have been saved through time savings, so no harm done.

"Must be logged in to upload a file" : Tried to upload a file once session expired. Log in again.

"Must have same number first and last names" : Tried submitting via textbox with an unmatched first or last name.

"Passwords don't match" : The password textbox text and the re-enter password textbox text do not match.

"Please check class name - it needs to match the skeleton class" : The class name of the submitted file does not match the class name the exercise expects.

"Please check class name for the Administrative Skeleton" : The class name of the submitted file does not match the class name the exercise expects.

"Please enter a password" : Tried to register password as "".

"Please only upload plain text or source files" : Tried to upload an invalid file type. Check file extension.

"Please only upload plain text or source files (skeleton)" : The skeleton file is of an invalid file type.

"Please only upload plain text or source files (sol)" : The solution file is of an invalid file type.

"Problem writing student file" : The student's code failed to be written on the server - contact the Systems Administrator.

"There was an internal error" : The system was unable to create the necessary directory on the server to compile the Microlab. Contact the Systems Administrator.

"This exercise has expired" : Attempted to submit a file that has expired. The administrator can change the **CloseDate** on the *Exercises* tab.

"User doesn't exist. Weird." : Somehow, the password was being updated for a user that doesn't exist. Shouldn't appear.

"User is not admin" : A regular user attempted to invoke a privileged action. Should not appear

Glossary

AdminSkeleton: An administrative class file that can be edited, but not submitted. Changes to the AdminSkeleton can be distributed to students through use of the *Update Skeletons* button on the *Exercises* tab.

CodeEditor: The left hand side of the **Editor Screen** where files can be edited and submitted for review.

Logical Actions: A section of the *Exercise* tab allowing for actions to be performed on Logical Microlabs.

Editor Screen (WAGS Editor): The main screen for Programming Microlabs, composed of a header panel, the **CodeEditor**, and the **TabPanel**.

Exercise Actions: A section of the *Exercise* tab allowing for actions to be performed on already uploaded Programming Exercises.

FileBrowser: The contents of the *FileBrowser* tab, which contains a directory structure of files available to the student or administrator.

HelperClass: An optional class from Programming Microlabs, that contains a useful API for the given exercise. It is visible to the student, but not editable.

Logical Microlab: A conceptual Microlab that does not include any code writing. Usually includes some interactive tapping or drag and drop functionality.

Logical Microlab Index Page: The webpage containing links to all available **Logical Microlab Problem Pages**.

Logical Microlab Problem Page: The webpage for an individual logical Microlab.

Problem Area: The editable portion of the **Logical Microlab Problem Page** that can be set to its initial state by pressing the *Reset* button.

Microlab: A short exercise that can be completed during lecture time to assess student understanding in an interactive environment. In the WAGS system, Microlabs are subdivided into **Programming** and **Logical** Microlabs.

Programming Microlab: A Microlab that requires the use of the **CodeEditor** to fill in some blank methods for a class in some exercise. Part of the implementation process of concepts covered in Logical Microlabs.

SkeletonClass: The class made available to students with one or more blank or unfinished methods the students must fill in for the exercise. Part of Programming Microlabs.

SolutionClass: The class included in an exercise that has correctly implemented methods that the **SkeletonClass** is tested against, via the **TestClass**. Part of Programming Microlabs.

TabPanel: The right side of the **Editor Screen**, allowing for navigation between different tabs provided by the WAGS System.

TestClass: The driver class that runs the methods of a provided **SolutionClass** and student code in a **SkeletonClass**, and signifies to the system whether or not the student was successful.

Vita

Philip Meznar was born in Dallas, Texas in 1988. After moving to South Carolina and Canada, Philip grew up in Peoria, Arizona, until leaving for college in the fall of 2006. His parents are Martin and Linda Meznar. He and his wife Natalie Meznar currently reside in Boone, N.C. Philip received his Bachelor's in Mathematics from Bryan University before continuing his education at Appalachian State University. He received his Master's in Computer Science in 2013. He hopes to continue developing his skills as a software developer and engineer throughout his professional career.