Executable Examples

For Programming Problem Comprehension

John Wrenn Shriram Krishnamurthi

Brown University

2019-12-04

Executable Examples

Executable Examples

For Programming Problem Comprehension

John Wrenn Shriram Krishnamurthi

Brown University

ITiCSE ’14

A Qualitative Think-Aloud Study of Novice Programmers’ Code Writing Strategies

Jacqueline Whalley
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219608
jwhalley@aut.ac.nz

Nadia Kasto
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219999
nkasto@aut.ac.nz

ABSTRACT
This paper presents part of a larger long term study into the cognitive aspects of the early stages of learning to write computer programs Tasks designed to trigger learning events were used to provide the opportunity to observe student learning, in terms of the development and modification of cognitive structures or schemata, during think aloud sessions. A narrative analysis of six students’ attempts to solve these tasks is presented. The students’ progression in learning and attitudinal approaches to learning is examined and provides some insight into the cognitive processes involved in learning computer programming.

Categories and Subject Descriptors
K.3 [Computers & Education]: Computer & Information Science Education – Computer Science Education

Keywords
Vygotsky, think aloud, schemas, novice programmers.

1. INTRODUCTION
Educators are well aware, that for novices learning to program is particularly difficult. Many studies have pointed to the fact that students cannot write code, cannot read and reason about code

learning. In order to do this we have taken inspiration from psychological theories of learning to design a research instrument to trigger learning events which result in changes of knowledge structure or adjusts the incoming information to a current knowledge structure. These cognitive structure changes are often explained using the notions of “assimilation to” and “accommodation of” cognitive schema. Assimilation and accommodation are common themes within the psychological study of learning. Assimilation relates to a process of modifying (usually by expanding) an existing cognitive structure (or schema) so that a new piece of information fits within that structure [2]. Accommodation occurs when the new information is too complex to be integrated into the existing structure - this means that, cognitive structures change in response to the new information or even that a new structure is formed [7].

Another theory of learning, developed by Vygotsky [14], included the notion of a *zone of proximal development* (ZPD). ZPD has been defined as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers" ([14], p86). Vygotsky believed that when a student is at the ZPD for a particular task, providing the appropriate assistance

Before I explain executable examples, I’m going to take us back to a summers’ day in sweden, five years ago, when Jacqueline Whalley and Nadia Kasto presented a talk aloud study. The interviewers gave novice students a programming problem, and instructed novices to solve it, all while talking-aloud their progress. But.

A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies

Jacqueline Whalley
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219608
jwhalley@aut.ac.nz

Nadia Kasto
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219999
nkasto@aut.ac.nz

> **Interestingly, [half of the participants] retrieved the 'counting integers' schema.** The students did not recognize that their program would not work and did not attempt to verify the correctness of their solutions. All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.

Keywords
Vygotsky, think aloud, schemas, novice programmers.

1. INTRODUCTION
Educators are well aware, that for novices learning to program is particularly difficult. Many studies have pointed to the fact that students cannot write code, cannot read and reason about code

Another theory of learning, developed by Vygotsky [14], included the notion of a *zone of proximal development* (ZPD). ZPD has been defined as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers" ([14], p86). Vygotsky believed that when a student is at the ZPD for a particular task, providing the appropriate assistance

> **Interestingly, [half of the participants] retrieved the 'counting integers' schema.** The students did not recognize that their program would not work and did not attempt to verify the correctness of their solutions. All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.

After reading the problem, half their participants retireved a familiar (but inappropriate) schema!

A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies

Jacqueline Whalley
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219608
jwhalley@aut.ac.nz

Nadia Kasto
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219999
nkasto@aut.ac.nz

> Interestingly, [half of the participants] retrieved the 'counting integers' schema. **The students did not recognize that their program would not work** and did not attempt to verify the correctness of their solutions. All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.

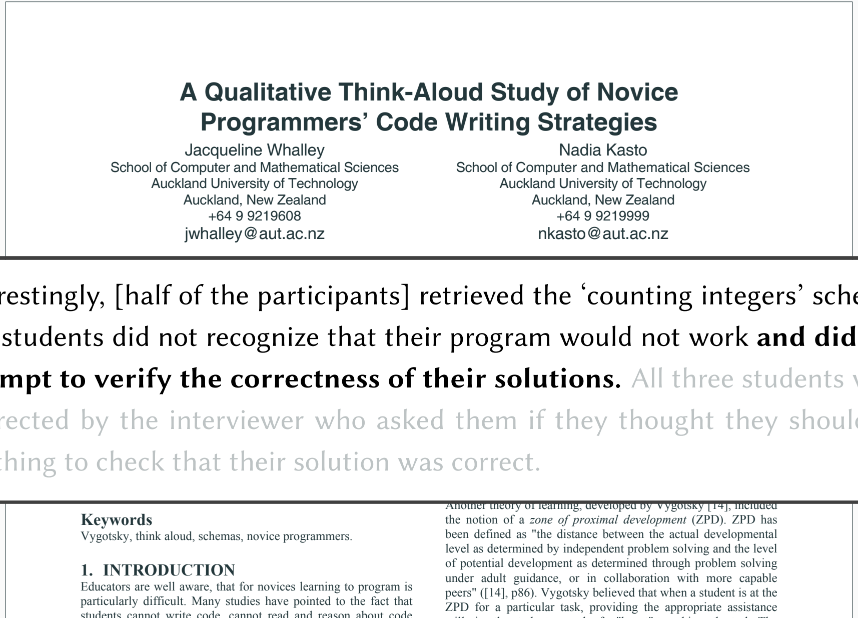
Keywords
Vygotsky, think aloud, schemas, novice programmers.

1. INTRODUCTION
Educators are well aware, that for novices learning to program is particularly difficult. Many studies have pointed to the fact that students cannot write code, cannot read and reason about code

Another theory of learning, developed by Vygotsky [14], included the notion of a *zone of proximal development* (ZPD). ZPD has been defined as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers" ([14], p86). Vygotsky believed that when a student is at the ZPD for a particular task, providing the appropriate assistance

Interestingly, [half of the participants] retrieved the 'counting integers' schema. **The students did not recognize that their program would not work** and did not attempt to verify the correctness of their solutions. All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.

They did not recognize that this schema was inappropriate.


A Qualitative Think-Aloud Study of Novice Programmers’ Code Writing Strategies

Jacqueline Whalley
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219608
jwhalley@aut.ac.nz

Nadia Kasto
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219999
nkasto@aut.ac.nz

Keywords
Vygotsky, think aloud, schemas, novice programmers.

1. INTRODUCTION
Educators are well aware, that for novices learning to program is particularly difficult. Many studies have pointed to the fact that students cannot write code, cannot read and reason about code

Another theory of learning, developed by Vygotsky [14], included the notion of a *zone of proximal development* (ZPD). ZPD has been defined as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers" ([14], p86). Vygotsky believed that when a student is at the ZPD for a particular task, providing the appropriate assistance

> Interestingly, [half of the participants] retrieved the ‘counting integers’ schema. The students did not recognize that their program would not work **and did not attempt to verify the correctness of their solutions.** All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.

They did nothing to verify the correctness of their soution.

ITiCSE '14

A Qualitative Think-Aloud Study of Novice Programmers' Code Writing Strategies

Jacqueline Whalley
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219608
jwhalley@aut.ac.nz

Nadia Kasto
School of Computer and Mathematical Sciences
Auckland University of Technology
Auckland, New Zealand
+64 9 9219999
nkasto@aut.ac.nz

> Interestingly, [half of the participants] retrieved the 'counting integers' schema. The students did not recognize that their program would not work and did not attempt to verify the correctness of their solutions. **All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.**

Keywords
Vygotsky, think aloud, schemas, novice programmers.

1. INTRODUCTION

Educators are well aware, that for novices learning to program is particularly difficult. Many studies have pointed to the fact that students cannot write code, cannot read and reason about code

Another theory of learning, developed by Vygotsky [14], included the notion of a *zone of proximal development* (ZPD). ZPD has been defined as "the distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance, or in collaboration with more capable peers" ([14], p86). Vygotsky believed that when a student is at the ZPD for a particular task, providing the appropriate assistance

Interestingly, [half of the participants] retrieved the 'counting integers' schema. The students did not recognize that their program would not work and did not attempt to verify the correctness of their solutions. **All three students were redirected by the interviewer who asked them if they thought they should do anything to check that their solution was correct.**

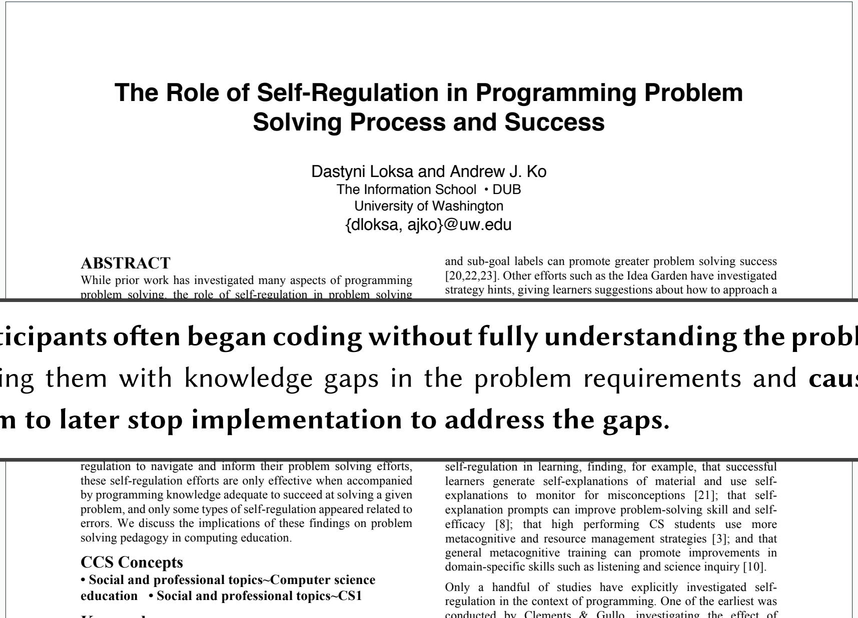
And it was not until the interviewer tipped them off that they realized their mistake.

Problem misunderstandings are pervasive.

Problem misunderstandings are pervasive.

It turns out these incidents are pervasive in talkaloud.

ICER ’16



The Role of Self-Regulation in Programming Problem Solving Process and Success

Dastyni Loksa and Andrew J. Ko
The Information School • DUB
University of Washington
{dloksa, ajko}@uw.edu

ABSTRACT

While prior work has investigated many aspects of programming problem solving, the role of self-regulation in problem solving

regulation to navigate and inform their problem solving efforts, these self-regulation efforts are only effective when accompanied by programming knowledge adequate to succeed at solving a given problem, and only some types of self-regulation appeared related to errors. We discuss the implications of these findings on problem solving pedagogy in computing education.

CCS Concepts

• Social and professional topics~Computer science education • Social and professional topics~CS1

and sub-goal labels can promote greater problem solving success [20,22,23]. Other efforts such as the Idea Garden have investigated strategy hints, giving learners suggestions about how to approach a

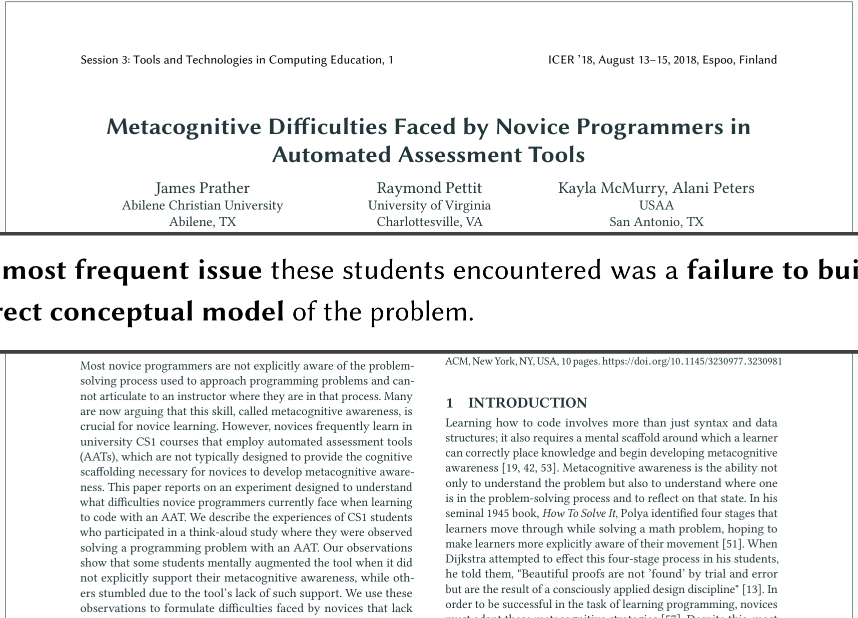
self-regulation in learning, finding, for example, that successful learners generate self-explanations of material and use self-explanations to monitor for misconceptions [21]; that self-explanation prompts can improve problem-solving skill and self-efficacy [8]; that high performing CS students use more metacognitive and resource management strategies [3]; and that general metacognitive training can promote improvements in domain-specific skills such as listening and science inquiry [10].

Only a handful of studies have explicitly investigated self-regulation in the context of programming. One of the earliest was conducted by Clements & Gullo, investigating the effect of

> **Participants often began coding without fully understanding the problem**, leaving them with knowledge gaps in the problem requirements and **causing them to later stop implementation to address the gaps.**

Participants often began coding without fully understanding the problem, leaving them with knowledge gaps in the problem requirements and **causing them to later stop implementation to address the gaps.**

We saw it again at ICER ’16, when Loksa and Ko observed that many students in their talkaloud study began coding without a full understanding of the problem.



Session 3: Tools and Technologies in Computing Education, 1 ICER ’18, August 13–15, 2018, Espoo, Finland

Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools

James Prather, Abilene Christian University, Abilene, TX
Raymond Pettit, University of Virginia, Charlottesville, VA
Kayla McMurry, Alani Peters, USAA, San Antonio, TX

Most novice programmers are not explicitly aware of the problem-solving process used to approach programming problems and cannot articulate to an instructor where they are in that process. Many are now arguing that this skill, called metacognitive awareness, is crucial for novice learning. However, novices frequently learn in university CS1 courses that employ automated assessment tools (AATs), which are not typically designed to provide the cognitive scaffolding necessary for novices to develop metacognitive awareness. This paper reports on an experiment designed to understand what difficulties novice programmers currently face when learning to code with an AAT. We describe the experiences of CS1 students who participated in a think-aloud study where they were observed solving a programming problem with an AAT. Our observations show that some students mentally augmented the tool when it did not explicitly support their metacognitive awareness, while others stumbled due to the tool’s lack of such support. We use these observations to formulate difficulties faced by novices that lack

ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3230977.3230981

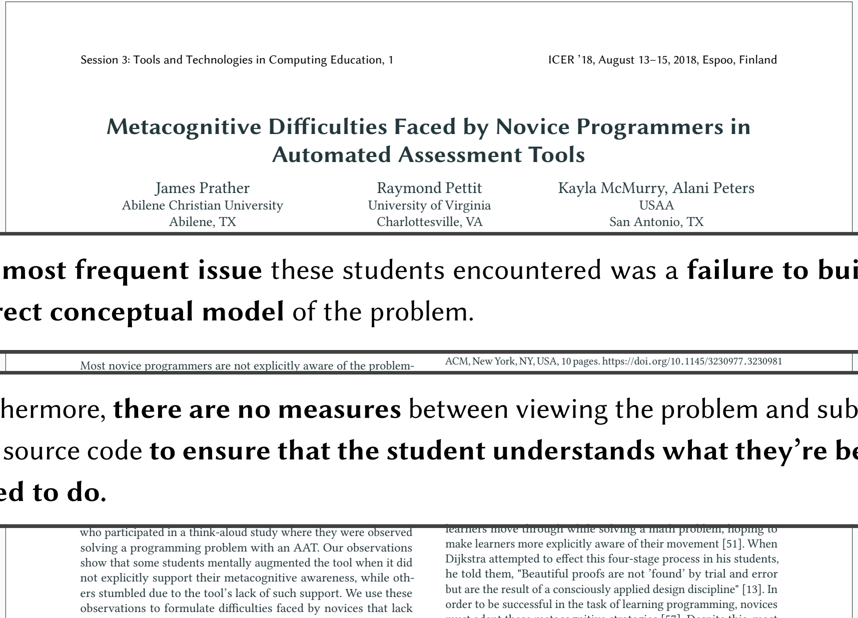
1 INTRODUCTION

Learning how to code involves more than just syntax and data structures; it also requires a mental scaffold around which a learner can correctly place knowledge and begin developing metacognitive awareness [19, 42, 53]. Metacognitive awareness is the ability not only to understand the problem but also to understand where one is in the problem-solving process and to reflect on that state. In his seminal 1945 book, *How To Solve It*, Polya identified four stages that learners move through while solving a math problem, hoping to make learners more explicitly aware of their movement [51]. When Dijkstra attempted to effect this four-stage process in his students, he told them, "Beautiful proofs are not 'found' by trial and error but are the result of a consciously applied design discipline" [13]. In order to be successful in the task of learning programming, novices

> The **most frequent issue** these students encountered was a **failure to build a correct conceptual model** of the problem.

The **most frequent issue** these students encountered was a **failure to build a correct conceptual model** of the problem.

...and once more in 2018, when Ray presented on a talkaloud plagued by misunderstandings of the problem itself.



> The **most frequent issue** these students encountered was a **failure to build a correct conceptual model** of the problem.

> Furthermore, **there are no measures** between viewing the problem and submitting source code **to ensure that the student understands what they're being asked to do.**

Inspiring my work, the authors noted that their task absolutely no mechanisms to prevent these sorts of errors!

reinterpretation
is the process of building understanding

...but students don't do it.

reinterpretation
is the process of building understanding

...but students don't do it.

This process of building a conceptual model from an abstract description of a task is *reinterpretation*, and students don't do it.

This is truly alarming. It's not just talk-alouds lack mechanisms to prevent misunderstandings before final submission; our coursework lacks it too.

Why don't students reinterpret the problem statement?

Why don't students reinterpret the problem statement?

We need to ask *why*?

Why don’t students reinterpret the problem statement?

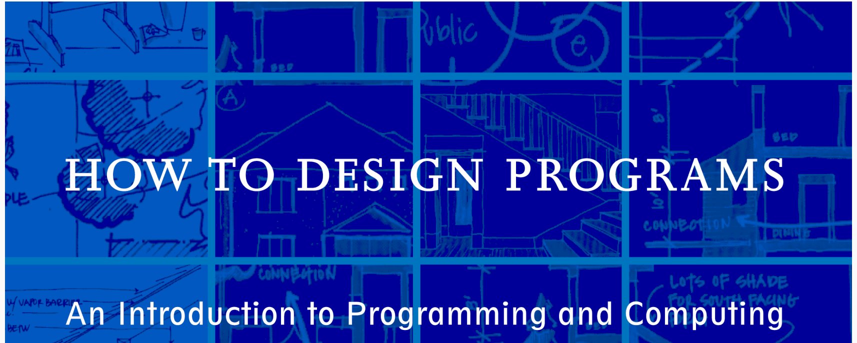
1. reinterpretation is **difficult**

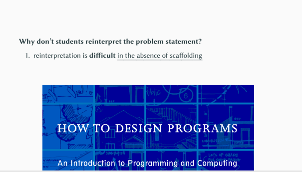
Why don’t students reinterpret the problem statement?
1. reinterpretation is difficult

Foremost, reinterpreation is difficult because it’s often unstructured and students may lack the vocabulary to do it effectively.

Why don’t students reinterpret the problem statement?

1. reinterpretation is **difficult** in the absence of scaffolding





A problem-solving methodology, such as that used by the Bootstrap curricula suggests a possible solution.

1. **From Problem Analysis to Data Definitions**
   Identify what must be represented and how it is represented.
2. **Signature, Purpose Statement, Header**
   State what kind of data the function consumes and produces.
3. **Input–Output Examples**
   Work through examples that illustrate the function's purpose.
4. **Function Template**
   Translate the data definitions into an outline of the function.
5. **Function Definition**
   Fill in the gaps in the function template.
6. **Testing**
   Ensure your implementation conforms to your understanding.

The design recipe is a six-step which scaffolds solving programming problems.

1. **From Problem Analysis to Data Definitions**
   Identify what must be represented and how it is represented.
2. **Signature, Purpose Statement, Header**
   State what kind of data the function consumes and produces.
3. **Input–Output Examples**
   Work through examples that illustrate the function's purpose.
4. **Function Template**
   Translate the data definitions into an outline of the function.
5. **Function Definition**
   Fill in the gaps in the function template.
6. **Testing**
   Ensure your implementation conforms to your understanding.

1. From Problem Analysis to Data Definitions
Identify what must be represented and how it is represented.
2. Signature, Purpose Statement, Header
State what kind of data the function consumes and produces.
3. Input–Output Examples
Work through examples that illustrate the function's purpose.
4. Function Template
Translate the data definitions into an outline of the function.
5. Function Definition
Fill in the gaps in the function template.
6. Testing
Ensure your implementation conforms to your understanding.

Only the last three steps involve writing code, the rest scaffold reinterpretation.

1. **From Problem Analysis to Data Definitions**
   Identify what must be represented and how it is represented.
2. **Signature, Purpose Statement, Header**
   State what kind of data the function consumes and produces.
3. **Input–Output Examples**
   Work through examples that illustrate the function's purpose.
4. **Function Template**
   Translate the data definitions into an outline of the function.
5. **Function Definition**
   Fill in the gaps in the function template.
6. **Testing**
   Ensure your implementation conforms to your understanding.

1. From Problem Analysis to Data Definitions
Identify what must be represented and how it is represented.
2. Signature, Purpose Statement, Header
State what kind of data the function consumes and produces.
3. Input–Output Examples
Work through examples that illustrate the function's purpose.
4. Function Template
Translate the data definitions into an outline of the function.
5. Function Definition
Fill in the gaps in the function template.
6. Testing
Ensure your implementation conforms to your understanding.

A critical step of reinterpretation is example-writing.

3 **Input–Output Examples**

Work through examples that illustrate the function's purpose.

3 Input–Output Examples
Work through examples that illustrate the function's purpose.

An input–output example is just an assertion of how a function should behave on a given input.

3 **Input–Output Examples**

Work through examples that illustrate the function's purpose.

```
sqrt(9)  is  3
```


3 Input–Output Examples
Work through examples that illustrate the function's purpose.
sqrt(9) is 3

sqrt(9) is 3

3 **Input–Output Examples**
Work through examples that illustrate the function's purpose.

```
sqrt(9)  is  3
sqrt(4)  is  2
```


3 Input–Output Examples
Work through examples that illustrate the function's purpose.
sqrt(9) is 3
sqrt(4) is 2

sqrt(4) is 2

3 **Input–Output Examples**

Work through examples that illustrate the function's purpose.

```
sqrt(9)  is   3
sqrt(4)  is   2
sqrt(4)  is  -2?
```


3 Input–Output Examples
Work through examples that illustrate the function's purpose.
sqrt(9) is 3
sqrt(4) is 2
sqrt(4) is -2?

But isn't sqrt(4) also -2?

3 **Input–Output Examples**

Work through examples that illustrate the function's purpose.

```
sqrt(9)  is   3
sqrt(4)  is   2
sqrt(4)  is  -2?
sqrt(2)  is  ???
```


3 Input–Output Examples
Work through examples that illustrate the function's purpose.
sqrt(9) is 3
sqrt(4) is 2
sqrt(4) is -2?
sqrt(2) is ???

And what if the result is irrational? How do you write the output?

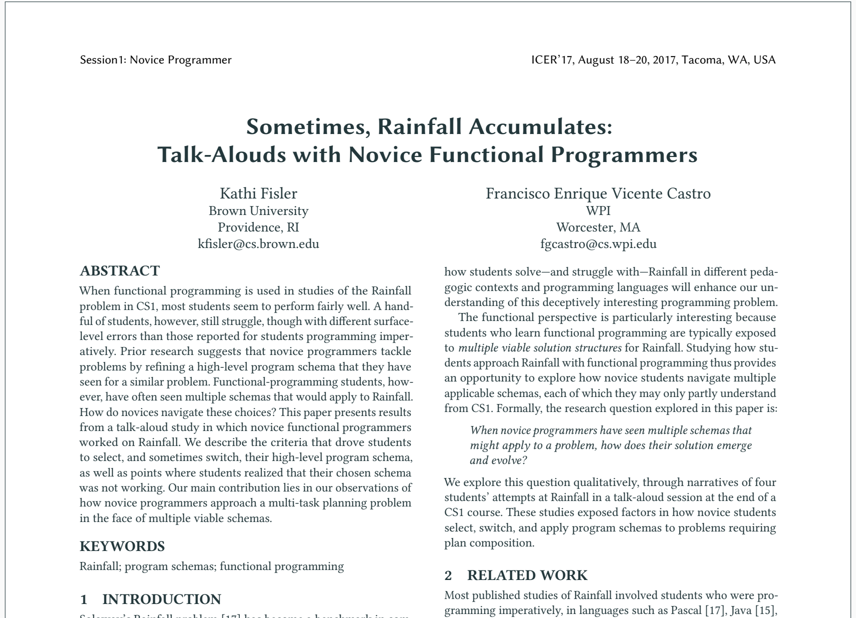
3 **Input–Output Examples**
Work through examples that illustrate the function's purpose.

```
sqrt(9)  is   3
sqrt(4)  is   2
sqrt(4)  is  -2?
sqrt(2)  is  ???
sqrt(-1) is  ???
```


3 Input–Output Examples
Work through examples that illustrate the function's purpose.
sqrt(9) is 3
sqrt(4) is 2
sqrt(4) is -2?
sqrt(2) is ???
sqrt(-1) is ???

Or if it's imaginary? These are the sorts of questions we need to answer *before* we begin our implementation.

ICER ’17



Session1: Novice Programmer ICER’17, August 18–20, 2017, Tacoma, WA, USA

Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers

Kathi Fisler
Brown University
Providence, RI
kfisler@cs.brown.edu

Francisco Enrique Vicente Castro
WPI
Worcester, MA
fgcastro@cs.wpi.edu

ABSTRACT

When functional programming is used in studies of the Rainfall problem in CS1, most students seem to perform fairly well. A handful of students, however, still struggle, though with different surface-level errors than those reported for students programming imperatively. Prior research suggests that novice programmers tackle problems by refining a high-level program schema that they have seen for a similar problem. Functional-programming students, however, have often seen multiple schemas that would apply to Rainfall. How do novices navigate these choices? This paper presents results from a talk-aloud study in which novice functional programmers worked on Rainfall. We describe the criteria that drove students to select, and sometimes switch, their high-level program schema, as well as points where students realized that their chosen schema was not working. Our main contribution lies in our observations of how novice programmers approach a multi-task planning problem in the face of multiple viable schemas.

KEYWORDS

Rainfall; program schemas; functional programming

1 INTRODUCTION

how students solve—and struggle with—Rainfall in different pedagogic contexts and programming languages will enhance our understanding of this deceptively interesting programming problem.

The functional perspective is particularly interesting because students who learn functional programming are typically exposed to *multiple viable solution structures* for Rainfall. Studying how students approach Rainfall with functional programming thus provides an opportunity to explore how novice students navigate multiple applicable schemas, each of which they may only partly understand from CS1. Formally, the research question explored in this paper is:

> *When novice programmers have seen multiple schemas that might apply to a problem, how does their solution emerge and evolve?*

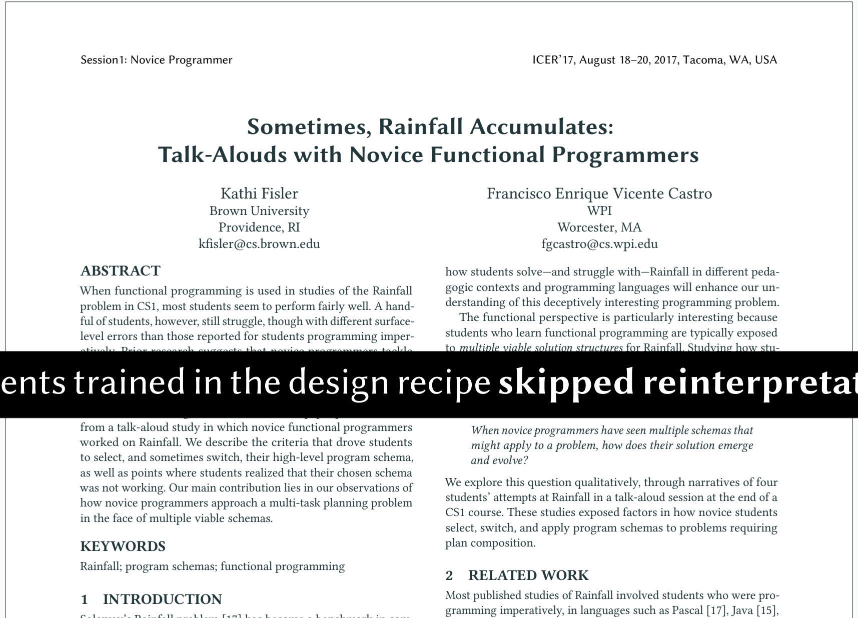
We explore this question qualitatively, through narratives of four students’ attempts at Rainfall in a talk-aloud session at the end of a CS1 course. These studies exposed factors in how novice students select, switch, and apply program schemas to problems requiring plan composition.

2 RELATED WORK

Most published studies of Rainfall involved students who were programming imperatively, in languages such as Pascal [17], Java [15],

It’s not.

ICER ’17



Sometimes, Rainfall Accumulates: Talk-Alouds with Novice Functional Programmers

Kathi Fisler
Brown University
Providence, RI
kfisler@cs.brown.edu

Francisco Enrique Vicente Castro
WPI
Worcester, MA
fgcastro@cs.wpi.edu

ABSTRACT
When functional programming is used in studies of the Rainfall problem in CS1, most students seem to perform fairly well. A handful of students, however, still struggle, though with different surface-level errors than those reported for students programming imper-

from a talk-aloud study in which novice functional programmers worked on Rainfall. We describe the criteria that drove students to select, and sometimes switch, their high-level program schema, as well as points where students realized that their chosen schema was not working. Our main contribution lies in our observations of how novice programmers approach a multi-task planning problem in the face of multiple viable schemas.

KEYWORDS
Rainfall; program schemas; functional programming

1 INTRODUCTION

how students solve—and struggle with—Rainfall in different pedagogic contexts and programming languages will enhance our understanding of this deceptively interesting programming problem.

The functional perspective is particularly interesting because students who learn functional programming are typically exposed to *multiple viable solution structures* for Rainfall. Studying how stu-

When novice programmers have seen multiple schemas that might apply to a problem, how does their solution emerge and evolve?

We explore this question qualitatively, through narratives of four students’ attempts at Rainfall in a talk-aloud session at the end of a CS1 course. These studies exposed factors in how novice students select, switch, and apply program schemas to problems requiring plan composition.

2 RELATED WORK

Most published studies of Rainfall involved students who were programming imperatively, in languages such as Pascal [17], Java [15],

Students trained in the design recipe **skipped reinterpretation**!

2019-12-04

Executable Examples
└Background
└Talk-Alouds

Students trained in the design recipe **skipped reinterpretation**!

It’s not.

Why don’t students reinterpret the problem?

1. reinterpretation is difficult (in the absence of scaffolding)

Why don’t students reinterpret the problem?
1. reinterpretation is difficult (in the absence of scaffolding)

So it can’t be *just* the case that reinterpretation is too difficult without scaffolding.

Why don’t students reinterpret the problem?

coding is interactive

reinterpretation is inert

Why don’t students reinterpret the problem?
coding is interactive
reinterpretation is inert

First, it’s inert. It’s *only when programming* that a student engages with an interactive system that executes their input engages them with feedback.

Why don't students reinterpret the problem?

coding provides feedback

reinterpretation is unconstructive

(sometimes)

Why don't students reinterpret the problem?
coding provides feedback
reinterpretation is unconstructive
(sometimes)

Secondly, scaffolding reinterpetation may encourage students to think about the problem, but it doesn't ensure that doing so is constructive.

1. **From Problem Analysis to Data Definitions**
   Identify what must be represented and how it is represented.
2. **Signature, Purpose Statement, Header**
   State what kind of data the function consumes and produces.
3. **Input–Output Examples**
   Work through examples that illustrate the function's purpose.
4. **Function Template**
   Translate the data definitions into an outline of the function.
5. **Function Definition**
   Fill in the gaps in the function template.
6. **Testing**
   Ensure your implementation conforms to your examples.

1. From Problem Analysis to Data Definitions
Identify what must be represented and how it is represented.
2. Signature, Purpose Statement, Header
State what kind of data the function consumes and produces.
3. Input–Output Examples
Work through examples that illustrate the function's purpose.
4. Function Template
Translate the data definitions into an outline of the function.
5. Function Definition
Fill in the gaps in the function template.
6. Testing
Ensure your implementation conforms to your examples.

If a student's misconception is consistently reflected both in their examples and in their implementation.

1. **From Problem Analysis to Data Definitions**
   Identify what must be represented and how it is represented.
2. **Signature, Purpose Statement, Header**
   State what kind of data the function consumes and produces.
3. **Input–Output Examples**
   Work through examples that illustrate the function's purpose.
4. **Function Template**
   Translate the data definitions into an outline of the function.
5. **Function Definition**
   Fill in the gaps in the function template.
6. **Testing**
   Ensure your implementation conforms to your examples.

1. From Problem Analysis to Data Definitions
Identify what must be represented and how it is represented.
2. Signature, Purpose Statement, Header
State what kind of data the function consumes and produces.
3. Input–Output Examples
Work through examples that illustrate the function's purpose.
4. Function Template
Translate the data definitions into an outline of the function.
5. Function Definition
Fill in the gaps in the function template.
6. Testing
Ensure your implementation conforms to your examples.

Then it will not be detected by adapting those examples as test cases.

Why don't students reinterpret the problem?

reinterpretation must be

compelling & helpful

Why don't students reinterpret the problem?

reinterpretation must be
compelling & helpful

We need to make examples something students can *run*, and running them should provide useful, actionable feedback.

Executing Examples

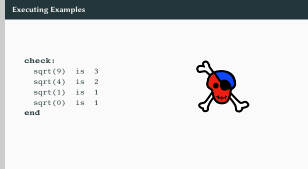
```
sqrt(9)  is  3
sqrt(4)  is  2
sqrt(1)  is  1
sqrt(0)  is  1
```


Executing Examples
sqrt(9) is 3
sqrt(4) is 2
sqrt(1) is 1
sqrt(0) is 1

To make these examples evaluable, we need to make them executable. In Pyret, a minor change

Executing Examples

```
check:
  sqrt(9)  is  3
  sqrt(4)  is  2
  sqrt(1)  is  1
  sqrt(0)  is  1
end
```



turns them into syntactically valid test cases. These can now be theoretically run, but the student doesn't have anything to run them against because they haven't begun their implementation. However, if we could run them, we'd like to know two things:

Evaluating Executable Examples

1. **valid**
   consistent with the problem statement

Evaluating Executable Examples
1. valid
consistent with the problem statement

First, that they're valid; *consistent* with the problem specification.

Evaluating Executable Examples

1. **valid**
   consistent with the problem statement
2. **interesting**
   good at detecting gaps in understanding

Evaluating Executable Examples
1. valid
consistent with the problem statement
2. interesting
good at detecting gaps in understanding

Second, they should be *interesting* and exercise important conceptual corners of the problem.

Evaluating Executable Examples

1. **valid**
   consistent with the problem statement
2. **interesting**
   good at detecting gaps in understanding

Evaluating Executable Examples
1. valid
consistent with the problem statement
2. interesting
good at detecting gaps in understanding

We can rephrase both of these types of feedback in terms of execution. A suite of examples or tests is *valid*

Evaluating Executable Examples

1. **valid**
   accepts correct implementations
2. **interesting**
   good at detecting gaps in understanding

Evaluating Executable Examples
1. valid
accepts correct implementations
2. interesting
good at detecting gaps in understanding

if it *accepts* correct implementations.

Evaluating Executable Examples

1. **valid**
   accepts correct implementations ("wheat")
2. **interesting**
   good at detecting gaps in understanding

Evaluating Executable Examples
1. valid
accepts correct implementations ("wheat")
2. interesting
good at detecting gaps in understanding

We call this set of implementations "wheat". And it's interesting

Evaluating Executable Examples

1. **valid**
   accepts correct implementations (“wheat”)
2. **interesting**
   rejects buggy implementations

Evaluating Executable Examples
1. valid
accepts correct implementations (“wheat”)
2. interesting
rejects buggy implementations

if it rejects many logically buggy implementations.

Evaluating Executable Examples

1. **valid**
   accepts correct implementations ("wheat")
2. **interesting**
   rejects buggy implementations ("chaff")

Evaluating Executable Examples
1. valid
accepts correct implementations ("wheat")
2. interesting
rejects buggy implementations ("chaff")

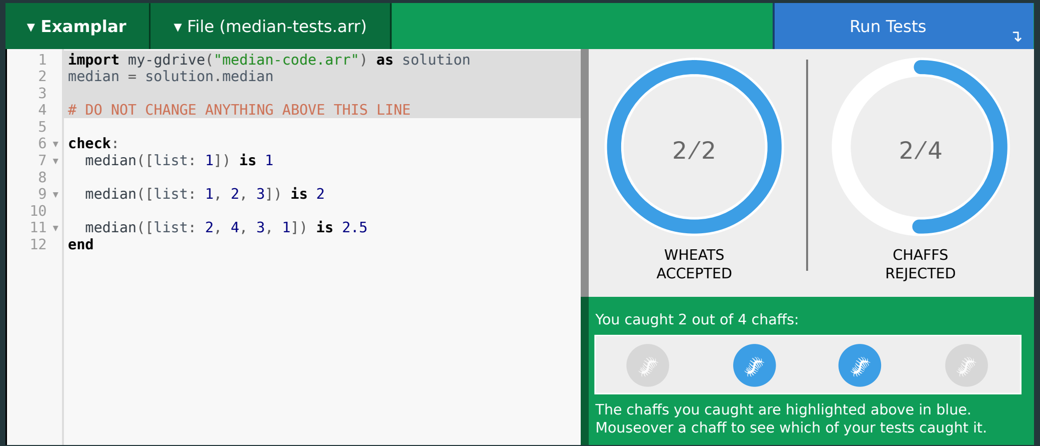
We call this set of implementations the "chaff".

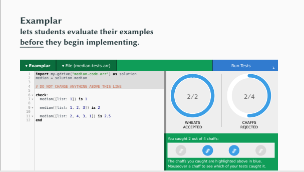
If students can trial run their examples against a set of wheat and chaff implementations, they can evaluate whether they're valid and interesting.

These implementations should be provided by the instructor, but not in a way where the student can inspect their source code.

Examplar

lets students evaluate their examples before they begin implementing.





We put these ideas into action in Examplar, an editing environment for examples, that enables students to evaluate their examples *before* they've begun their implementation.

Note: it's a problem if students give valid tests for the mean, since it may indicate a misonception about what "median" is.

Hypotheses

executable examples are

compelling & helpful

Hypotheses

executable examples are
compelling & helpful

We believed that with this tool, executable examples would be compelling and helpful to write.

Study Context

1. accelerated introduction to CS
2. 67 students
3. mostly first-year
4. typically some background in programming

Study Context
1. accelerated introduction to CS
2. 67 students
3. mostly first-year
4. typically some background in programming

We deployed Examplar in a fall-semester accelerated introduction to CS. 67 students, mostly first-years, completed the course. They typically had some prior background in programming, but not in testing.

To ensure that students gained familiarity in the tool, we required the instructor gave an in-class demonstration of Examplar during the first lecture, and required them to use Examplar on the first assignment (assigned that day).

Was Examplar compelling?

Was Examplar compelling?

To test whether students found Examplar compelling to use, we removed the requirement to use the tool after the first assignment.

Did Examplar convince students to use it?

With usage requirement:

- **100%** submitted at least once.
- median **22** submissions/student.

Did Examplar convince students to use it?
With usage requirement:
- 100% submitted at least once.
- median 22 submissions/student.

With the usage requirement, which all students adhered to, the median student evaluated their examples about twenty times.

Did Examplar convince students to use it?

With usage requirement:

- **100%** submitted at least once.
- median **22** submissions/student.

Without usage requirement:

- median **1** non-user per assignment.
- median **36** submissions/student, per assignment.

Did Examplar convince students to use it?
With usage requirement:
• 100% submitted at least once.
• median 22 submissions/student.
Without usage requirement:
• median 1 non-user per assignment.
• median 36 submissions/student, per assignment.

Without the usage requirement, nearly all students continued using Examplar for every assignment. The median number of non-users per assignment was 1 and non-use was intermittent.

Now, this class also required students to submit final test suites, along with their final implementations. Because of the close correspondence between examples and tests, it's conceivable that students only bothered with Examplar *because* they were required to test.

When final test suites were not required?

On a **multi-part** assignment that **did not require test suite submission**

When final test suites were not required?

On a **multi-part** assignment that **did not require test suite submission**

Fortunately, there was one assignment for which *no* final test suite submission was required. On this multi-part assignment

When final test suites were not required?

On a **multi-part** assignment that **did not require test suite submission**, Examplar was used by...

96% for at least one subproblem

When final test suites were not required?

On a **multi-part** assignment that **did not require test suite submission**, Examplar was used by...

96% for at least one subproblem

96% of students used examplar for at least one subproblem.

When final test suites were not required?

On a **multi-part** assignment that **did not require test suite submission**, Examplar was used by...

96% for at least one subproblem

71% for **all** subproblems

When final test suites were not required?
On a **multi-part** assignment that **did not require test suite submission**, Examplar was used by...
96% for at least one subproblem
71% for **all** subproblems

71% of students used examplar for at least *all* subproblems.

Was Examplar **helpful**?

Was Examplar helpful?

So students used Examplar. But was it actually helpful? To answer this, we compared the quality of final submissions from 2018, to those in 2017, when students did *not* have Examplar available.
Both offerings the same instructor, similar populations, and nearly identical assignments.
In both years, students submitted both final test suites, and final implementations.
On the assignments that were unchanged between years, we drew direct comparisons between the quality of the submissions in each year.

First, we considered the quality of their test suites. Like examples, tests should be *valid*.

Validity Improved Drastically

test suites in 2017 were

4.8$\times$ more likely to be invalid

than suites in 2018.

$\chi^2(1, N = 589) = 52.373$, $p < 0.01$, $\phi = 0.303$

Validity Improved Drastically
test suites in 2017 were
4.8× more likely to be invalid
than suites in 2018.
$\chi^2(1, N = 589) = 52.373$, $p < 0.01$, $\phi = 0.303$

Test suites in 2017 were nearly five times more likely to be invalid, than test suites in 2018.

Validity Improved Drastically

Final submissions in 2017 were...

12.9× more likely to test underspecified behavior

6.1× more likely to test incorrect behavior

1.3× more likely to be malformed

...than final submissions in 2018.

Validity Improved Drastically

Final submissions in 2017 were...

12.9× more likely to test underspecified behavior

6.1× more likely to test incorrect behavior

1.3× more likely to be malformed

...than final submissions in 2018.

We can attribute this decline to three factors. Suites in 2017 were 13 times more likely to test underspecified behavior, 6 times more likely to test incorrect behavior, and slighty more likely to be broken in ways the gummed up the autograder.

Implementation Correctness Improved Slightly

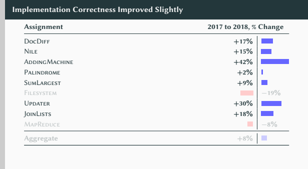
Assignment	2017 to 2018, % Change
DocDiff	+17%
Nile	+15%
AddingMachine	+42%
Palindrome	+2%
SumLargest	+9%
Filesystem	−19%
Updater	+30%
JoinLists	+18%
MapReduce	−8%
Aggregate	**+8%**

$\chi^2(1, N = 1144) = 2.94$, $p = 0.086$, $\phi = 0.053$

We also observed improvements in their implementations. In aggregate, the fraction of student implementations that were *completely* correct (i.e., they passed *every* test in the instructor's test suite) increased by 8%.

Implementation Correctness Improved Slightly

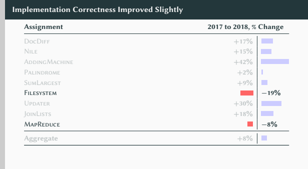
Assignment	2017 to 2018, % Change
DocDiff	+17%
Nile	+15%
AddingMachine	+42%
Palindrome	+2%
SumLargest	+9%
Filesystem	−19%
Updater	+30%
JoinLists	+18%
MapReduce	−8%
Aggregate	+8%



We observed these improvements in nearly every assignment

Implementation Correctness Improved Slightly

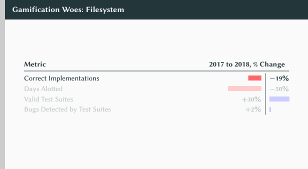
Assignment	2017 to 2018, % Change
DocDiff	+17%
Nile	+15%
AddingMachine	+42%
Palindrome	+2%
SumLargest	+9%
Filesystem	−19%
Updater	+30%
JoinLists	+18%
MapReduce	−8%
Aggregate	+8%



with the exception of two. In one case, Filesystem, correctness declined percipitously.

Gamification Woes: Filesystem

Metric	2017 to 2018, % Change
Correct Implementations	−19%
Days Alotted	−50%
Valid Test Suites	+30%
Bugs Detected by Test Suites	+2%



On Filesystem, the fraction of students that submitted completely correct implementations declined by nearly twenty percent.

Gamification Woes: Filesystem

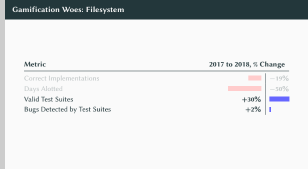
Metric	2017 to 2018, % Change
Correct Implementations	−19%
Days Alotted	−50%
Valid Test Suites	+30%
Bugs Detected by Test Suites	+2%

Gamification Woes: Filesystem
Metric 2017 to 2018, % Change
Correct Implementations −19%
Days Alotted −50%
Valid Test Suites +30%
Bugs Detected by Test Suites +2%

Taken alone, this isn't suprising, because the 2018 offering cut the number of days alotted to this assignment in half.

Gamification Woes: Filesystem

Metric	**2017 to 2018, % Change**
Correct Implementations	−19%
Days Alotted	−50%
Valid Test Suites	+30%
Bugs Detected by Test Suites	+2%



But what is surprising is that the fraction of test suites that were valid improved by thirty percent, and their ability to detect buggy implementations didn't decline at all.

Results

executable examples are

compelling & helpful...

Results

executable examples are
compelling & helpful...

So, while executable examples are, broadly speaking, compelling and helpful,

Results

executable examples are compelling & helpful...

...but gamification may be a double-edged sword.

Results
executable examples are compelling & helpful...
...but gamification may be a double-edged sword.

ICER ’18

Metacognitive Difficulties Faced by Novice Programmers in Automated Assessment Tools

James Prather
Abilene Christian University
Abilene, TX
jrp09a@acu.edu

Raymond Pettit
University of Virginia
Charlottesville, VA
raymond.pettit@gmail.com

Kayla McMurry, Alani Peters
USAA
San Antonio, TX
kayla.mcmurry,alani.peters@usaa.com

John Homer
Abilene Christian University
Abilene, TX

Maxine Cohen
Nova Southeastern University
Ft. Lauderdale, FL

> The feedback from Athene seems to have given [several participants] a false sense of progression through the problem.

university CS1 courses that employ automated assessment tools (AATs), which are not typically designed to provide the cognitive scaffolding necessary for novices to develop metacognitive awareness. This paper reports on an experiment designed to understand what difficulties novice programmers currently face when learning to code with an AAT. We describe the experiences of CS1 students who participated in a think-aloud study where they were observed solving a programming problem with an AAT. Our observations show that some students mentally augmented the tool when it did not explicitly support their metacognitive awareness, while others stumbled due to the tool’s lack of such support. We use these observations to formulate difficulties faced by novices that lack

can correctly place knowledge and begin developing metacognitive awareness [19, 42, 53]. Metacognitive awareness is the ability not only to understand the problem but also to understand where one is in the problem-solving process and to reflect on that state. In his seminal 1945 book, *How To Solve It*, Polya identified four stages that learners move through while solving a math problem, hoping to make learners more explicitly aware of their movement [51]. When Dijkstra attempted to effect this four-stage process in his students, he told them, "Beautiful proofs are not 'found' by trial and error but are the result of a consciously applied design discipline" [13]. In order to be successful in the task of learning programming, novices

The feedback from Athene seems to have given [several participants] a false sense of progression through the problem.

In that ICER paper last year which inspired this work, the authors noted that the feedback from Athene—a system which provided on-demand feedback on student’s implementations—seemed to induce a false sense of progress. Something similar may have happened on Filesystem.

Results

executable examples are

compelling & helpful...

Results
executable examples are
compelling & helpful...

Results

executable examples are

compelling & helpful...

...but still we know very little about student testing.

Results
executable examples are
compelling & helpful...
...but still we know very little about student testing.

We also discovered that we know very little about student testing behavior as a whole.

ITiCSE ’14

Do Student Programmers All Tend to Write the Same Software Tests?

Stephen H. Edwards and Zalia Shams
Department of Computer Science
Virginia Tech
2202 Kraft Drive, Blacksubrg, VA 24060 USA
+1-540-231-5723
{edwards, zalia18}@cs.vt.edu

ABSTRACT

While many educators have added software testing practices to their programming assignments, assessing the effectiveness of student-written tests using statement coverage or branch coverage has limitations. While researchers have begun investigating alternative approaches to assessing student-written tests, this paper reports on an investigation of the quality of student written tests in terms of the number of authentic, human-written defects those tests can detect. An experiment was conducted using 101 programs written for a CS2 data structures assignment where students implemented a queue two ways, using both an array-based and a link-based representation. Students were required to write their own software tests and graded in part on the branch coverage they achieved. Using techniques from prior work, we were able to approximate the number of bugs present in the collection of student solutions, and identify which of these were detected by each student-written test suite. The results indicate that, while students achieved an average branch coverage of 95.4% on their own solutions, their test suites were only able to detect an average of 13.6% of the faults present in the entire program population. Further, there was a high degree of similarity among 90% of the student test suites. Analysis of the suites suggest that students were following naive, “happy path” testing, writing basic test cases covering mainstream expected

Keywords

Software testing, automated assessment, automated grading, mutation testing, programming assignments, test coverage, test quality, happy path.

1. INTRODUCTION

Many educators have been adding software testing to their programming courses since the idea was first proposed over a dozen years ago [8][9]. Test-driven development [3], or at least using xUnit-style unit testing frameworks such as JUnit, are one approach. Automated assessment tools have grown to support assessing how well students test their own code [4].

More recently, however, some education researchers have begun investigating the quality of student-written tests, as well as techniques to evaluate this quality. By test quality—for a single test case, or an entire test suite—we mean its ability to detect bugs or faults in the software under construction. While most grading tools use some form of code coverage metric (e.g., statement coverage or branch coverage) to assess test thoroughness based on how much of the student’s program is executed during testing, this metric may overestimate test quality. Aaltonen et al. [1] proposed mutation testing as an alternative metric, while Shams and Edwards investigated its feasibility for classroom use and

One of the more widely read papers on the topic debuted at ITiCSE ’14 and asked: Do student programmers all tend to write the same tests?

Do Student Programmers All Tend to Write the Same Software Tests?

Edwards & Shams characterized student's test suites as:

1. **short** (only one student wrote more than 21 test cases)
2. **similar** (89% of students wrote exactly 21 test cases)
3. **ineffective** (missed a "significant proportion" of bugs).

alternative approaches to assessing student-written tests, this paper reports on an investigation of the quality of student written tests in terms of the number of authentic, human-written defects those tests can detect. An experiment was conducted using 101 programs written for a CS2 data structures assignment where students implemented a queue two ways, using both an array-based and a link-based representation. Students were required to write their own software tests and graded in part on the branch coverage they achieved. Using techniques from prior work, we were able to approximate the number of bugs present in the collection of student solutions, and identify which of these were detected by each student-written test suite. The results indicate that, while students achieved an average branch coverage of 95.4% on their own solutions, their test suites were only able to detect an average of 13.6% of the faults present in the entire program population. Further, there was a high degree of similarity among 90% of the student test suites. Analysis of the suites suggest that students were following naive, "happy path" testing, writing basic test cases covering mainstream expected

1. INTRODUCTION

Many educators have been adding software testing to their programming courses since the idea was first proposed over a dozen years ago [8][9]. Test-driven development [3], or at least using xUnit-style unit testing frameworks such as JUnit, are one approach. Automated assessment tools have grown to support assessing how well students test their own code [4].

More recently, however, some education researchers have begun investigating the quality of student-written tests, as well as techniques to evaluate this quality. By test quality—for a single test case, or an entire test suite—we mean its ability to detect bugs or faults in the software under construction. While most grading tools use some form of code coverage metric (e.g., statement coverage or branch coverage) to assess test thoroughness based on how much of the student's program is executed during testing, this metric may overestimate test quality. Aaltonen et al. [1] proposed mutation testing as an alternative metric, while Shams and Edwards investigated its feasibility for classroom use and

Edwards & Shams characterized student's test suites as:
1. **short** (only one student wrote more than 21 test cases)
2. **similar** (89% of students wrote exactly 21 test cases)
3. **ineffective** (missed a "significant proportion" of bugs).

The authors characterized student test suites as being short (only one student wrote more than 21 tests), similar (89% wrote exactly 21 test cases), and inneffective at catching bugs.

Reading this, you might be wondering: *why would I care about examples and test writing?*.

ITiCSE '14

Do Student Programmers All Tend to Write the Same Software Tests?

Edwards & Shams characterized student's test suites as:

1. **short** (only one student wrote more than 21 test cases)
2. **similar** (89% of students wrote exactly 21 test cases)
3. **ineffective** (missed a "significant proportion" of bugs).

1. INTRODUCTION

Our population's suites (produced both with and without Examplar):

1. **long** (students wrote an average of 39 test cases),
2. **varied** (some students wrote more than 200 tests)
3. **effective** at catching bugs.

similarity among 90% of the student test suites. Analysis of the suites suggest that students were following naive, "happy path" testing, writing basic test cases covering mainstream expected

proposed mutation testing as an alternative metric, while Shams and Edwards investigated its feasibility for classroom use and

Edwards & Shams characterized student's test suites as:
1. short (only one student wrote more than 21 test cases)
2. similar (89% of students wrote exactly 21 test cases)
3. ineffective (missed a "significant proportion" of bugs).
Our population's suites (produced both with and without Examplar):
1. long (students wrote an average of 39 test cases),
2. varied (some students wrote more than 200 tests)
3. effective at catching bugs.

Well, in contrast to these observations, the test suites we studied (even without examplar) were long, varried, and highly effective at catching bugs.

The differences between our contexts are innumerable. Different places, different languages, different pedagogies—to name a few.

Takeaway

executable examples are compelling & helpful

…but gamification may be a double-edged sword.
…and we still know very little about student testing.

Executable Examples for Programming Problem Comprehension

John Wrenn
Computer Science Department
Brown University
Providence, Rhode Island, USA
jswrenn@cs.brown.edu

Shriram Krishnamurthi
Computer Science Department
Brown University
Providence, Rhode Island, USA
sk@cs.brown.edu

ABSTRACT
Flawed problem comprehension leads students to produce flawed

form of input–output assertions, independent of testing their imple-
mentations. However, without an implementation to run assertions

Takeaway
executable examples are compelling & helpful
…but gamification may be a double-edged sword.
…and we still know very little about student testing.
Executable Examples for Programming Problem Comprehension

What these differences tell us is that our understanding of student testing is in its infancy and our field's accepted precepts about testing might actually not generalize as well as we thought. The factors that influence student test writing may be within our control! While executable examples are compelling and helpful, I'll conclude with a plea: there is so much more to discover, and it's up to us to discover it.

Thank you, I look forward to your questions.