Nipping Bugs in the Bud – Student Mistakes in Introductory Computer Science

Nivedita Chopra, advised by Roy Maxion and Robert Simmons

Background

Bugs in computer programs may be caused by mistakes in the programmer's thought process. Mitigating the causes of common mistakes in introductory computer science classes can help improve the quality of code that students write in future classes, as well as in industry or academia.

Aim

We want to mitigate common mistakes made by students in the Principles of Imperative Computation course (15-122, taught in C0 and C) by analyzing the bugs committed by students in the class, determining the mistakes in thought processes that caused these bugs, and devising ways to correct these mistakes by changing our teaching methods.

Data

We collected information about bugs committed by students, including a description of the bug, an optional code snippet, and the student's reasoning behind the bug. This information was collected throughout the Fall 2014 semester (335 students) and during one lab in the Spring 2015 semester (288 students).

Method

We classified each bug according to the IEEE Standard Classification of Software Anomalies (2010), determined the mistakes behind common bugs, and proposed ways to mitigate such mistakes.

Analysis

We found 63 logic bugs, 21 data bugs, and 7 interface bugs per the IEEE standard. An additional category of bugs emerged that we called "comprehension bugs" (32 instances), which are caused by a misunderstanding of concepts, specifications, or error messages. Among logic bugs, the largest subcategory was due to edge cases (31 instances). Edge cases are error conditions occurring at the extremes of operating parameters, perhaps not thoroughly considered by the programmer.

CATEGORIZATION OF BUGS OBSERVED IN 15-122

Logic Bug (as per the IEEE classification)

Traverse through a linked list performing a certain operation on each element

```
while (L->next != NULL) {
  //do something;
  L = L->next;
}
```

Bug : *Misses the last element in the list*

Data Bug (as per the IEEE classification)

Write a function to check if a given integer is in a linked list

```
bool is_in(list L, int n) {
  while(L->start != NULL) {
    if(L->start->data == n) return true;
    L->start = L->start->next;
  }
  return false;
}
```

Bug : *Destroys the linked list during an effect-free operation*

Interface Bug (as per the IEEE classification)

Write a function that removes the green component of a given pixel

Interface :

```
pixel make_pixel(int a, int r, int g, int b);
int get_green(pixel p);
```

Implementation :

```
typedef int pixel;
```

Client code (written by student) :

```
pixel remove_green(pixel p) {
  return (p & 0xFF00);
}
```

Bug : *Relies on the implementation while writing client code, which should only deal with the interface*

Comprehension Bug (not seen in the IEEE classification)

1) Create the list containing 1, 2, and 4, in order, using the cons and nil functions

```
[list cons(int, list); list nil();]
list L = cons(1, cons(2, cons(4)));
```

Bug : *Fails to realize that the cons function takes two arguments*

2) The offset is a signed 16 bit integer that is given as a two-byte operand to the instruction. (Instructions are stored in an array of (unsigned) bytes (ubyte *P))

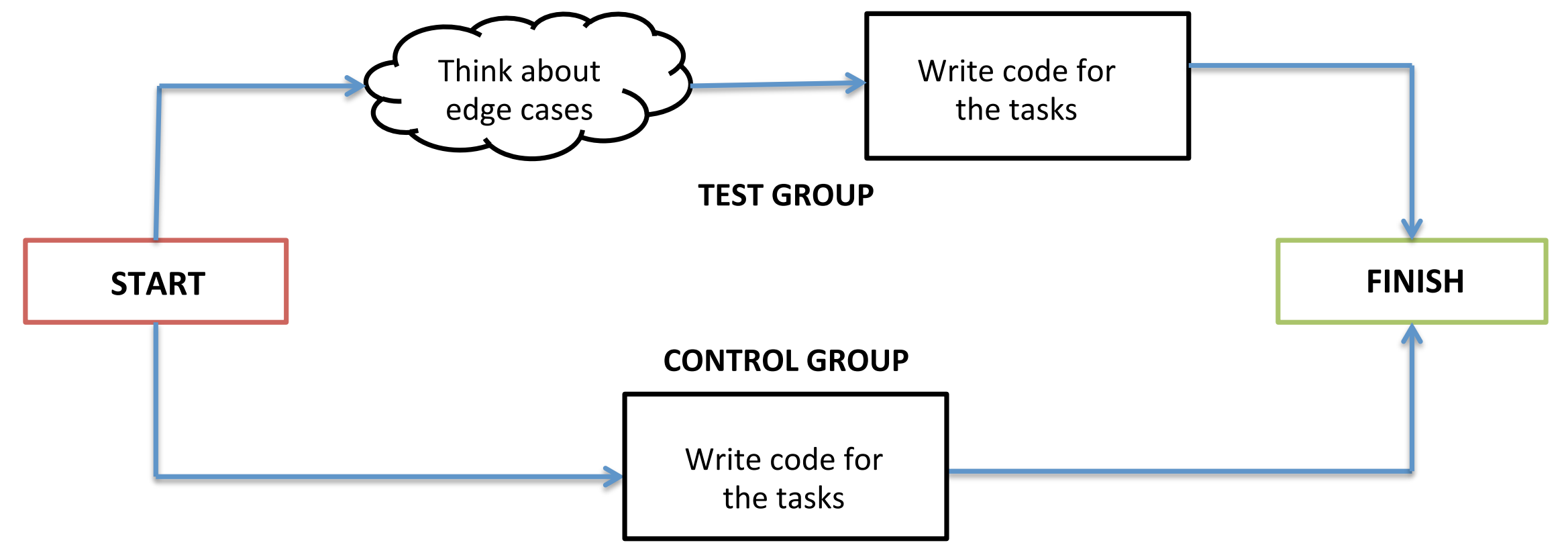
```
int16_t o1 = (int16_t)(int8_t)P[pc+1];
int16_t o2 = (int16_t)(int8_t)P[pc+2];
int16_t offset = ((o1 << 8) | o2);
```

Bug : *Casts both o1 and o2 into signed integers to make resultant offset signed. Sign extension on o2 may alter the final quantity.*

Pilot Experiment

With the aim of mitigating mistakes caused by lack of attention to edge cases, we conducted a pilot experiment during one of the weekly labs in the Spring 2015 semester. We split the students in the class into two groups — a test group and a control group. We encouraged the test group to spend five minutes thinking about edge cases in the assigned problem before beginning to code; the control group received no special instructions.

PILOT DESIGN



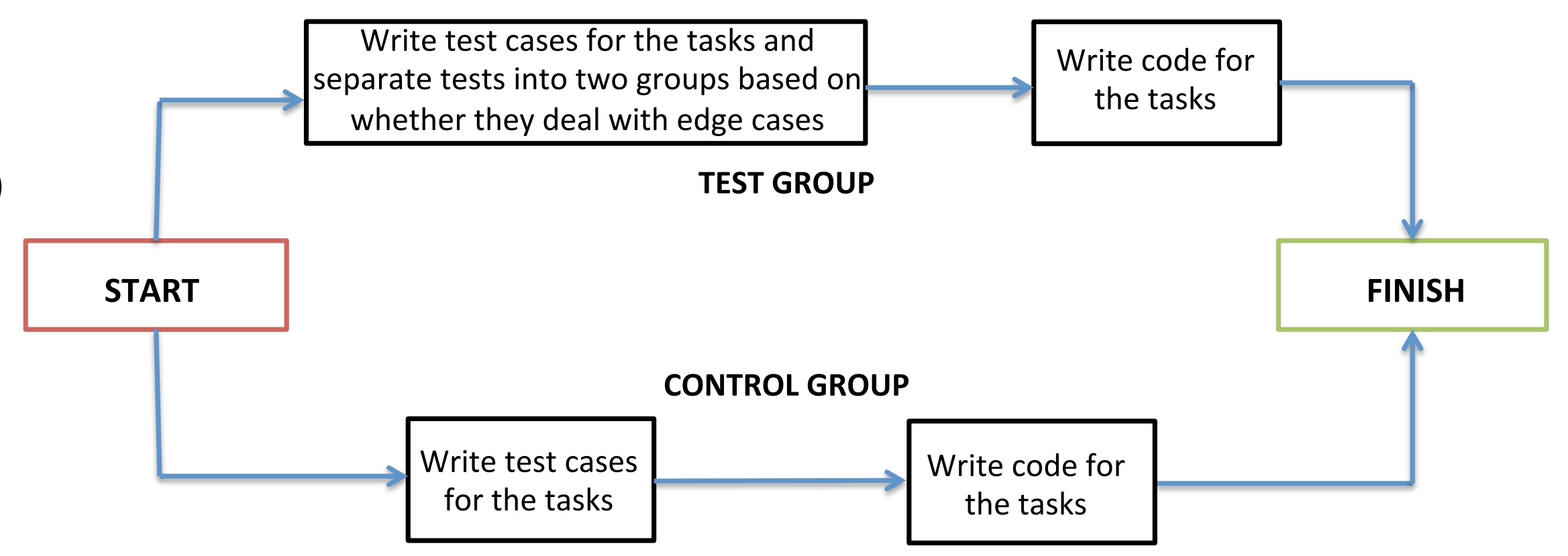
Results

We expected that students who think about edge cases prior to coding would perform better in terms of faster completion times and more incremental progress to solution. When we analyzed the pilot as a full-scale experiment, the analysis yielded a negative result — the activity did not seem to have any effect on time to completion or on progress toward an error-free solution.

Conclusion

The pilot helped us assess the feasibility of conducting this experiment in a classroom setting and highlighted details in the experimental design that need to be improved upon for a full-scale experiment to be performed in Spring 2016. The negative result is difficult to explain, since we lack evidence as to whether the students actually thought about edge cases. We proposed an improved experimental design for Spring 2016 that provides such evidence. Additionally, consistent instructions across lab sessions, and random assignment of students to the two groups, may garner a valid and more conclusive result in the future.

IMPROVED DESIGN


Write test cases for the tasks and separate tests into two groups based on whether they deal with edge cases
Write code for the tasks
TEST GROUP
START
FINISH
CONTROL GROUP
Write test cases for the tasks
Write code for the tasks

Reference. "IEEE Standard Classification for Software Anomalies," IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993), Jan. 7 2010.