Ravi Woods

Final Year Project

{Interim Report

Table of Contents

CHAPTER 1

INTRODUCTION & SPECIFICATION

In the 21st century, it is becoming increasingly likely that children and young adults will need to program in their lifetime. However, while resources for teaching mathematics, it can sometimes become frustrating how little help you get as a novice programmer. While schools and university courses guide you to understand, it is ultimately up to you to try to fully grasp how a program works, or why it currently isn't working. To novice programmers this is a daunting task, often because code is not inherently visual. While we are used to interacting with computers through well-designed user interfaces, it is down to the novice programmer to instead interact with a computer just through text. So, in this project, I propose an implementation of two visual tools which will, hopefully, help novice programmers get over that initial shock, and help them to understand programming more fully.

1.1 Pointer Declarations

Many novice programmers find reading and writing pointer declarations particularly difficult. For example, in the declaration `char *str[10]`, working out that `str` is an array of pointers to characters, with size 10, is not easy to work out.

This is further complicated with keywords. For example, many beginners have issues with the difference between:

1. `int const *` - A pointer to a constant integer
2. `int * const` - A constant pointer to an integer

Another complication is with function pointers. Even advanced level programmers would struggle with `char *(*fp)( int, float *)`. So, David Anderson devised the Clockwise Spiral Rule (Anderson, 1994) which makes this problem less tricky, by showing how to read these declarations using a clockwise spiral.

However, from personal experience, I have still had difficulty reading these type of declarations. I believe having a graph displayed would help beginners more easily parse these types of

declaration, so I plan to do just that. So, for our case above, `char *str[10]` , the graph would be similar to Figure 1.1

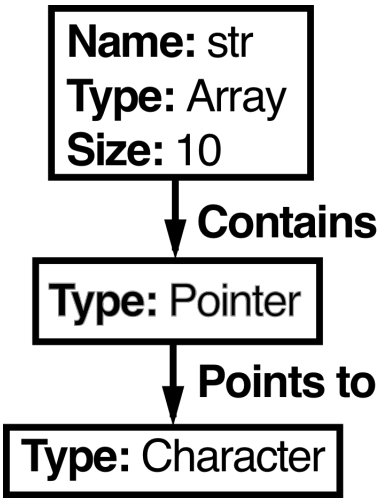

FIGURE 1.1. Example graph output for `char *str[10]`

1.2 Last-Write Analysis

Often, since many lines of a program have the option of changing a variable, it is often difficult to know where an erroneous write may occur. This is part of the reason classes are so valuable to programmers - variables can be encapsulated within a smaller section of code, so writes can be tracked more easily. However, novice programmers do not necessarily learn classes quickly, so need to understand what is being written to when. Thus, for the second half of the project, I propose a last-write analysis tool. Using static analysis and control flow graphs, I can ascertain the possible last writes of a variable at a current line. T

his analysis has three levels of complexity. The first is seen in Listing 1.1. If we enquire about the value of x at the end of line 6, the system would tell you that the last write may have been at line 5, or at line 2. The second level of complexity would be that the system tells you that the last-write of x depends on the value of y.

Listing 1.1:

```
1 int main() {
2    int x = 10;
3    int y = x*5/2;
4    if (y > 20) {
5       x = 7;
6    }
7 }
```

The last level of complexity is seen in Listing 1.2. If we enquire about the value of x[2] at the end of line 8, the system would tell you that the last write may have been at line 7, or at line 3, depending on the state of variable y. A similar effect could be produced with vectors.

Listing 1.2:

```
int main() {
   int x[5];
   x[2] = 3;
   int y = 10*5/2;
   if (y > 20) {
      x[1] = 7;
      x[2] = 12;
   }
}
```

CHAPTER 2

BACKGROUND RESEARCH

2.1 User Interfacing

In his seminal book on the subject of User Interfacing, *The Design of Everyday Things*, the designer Don Norman says that there is an inherent paradox in modern technology, where new technologies should benefit our lives, but instead they often add stress, due to added complexity (Norman, 2001, p 32). This, he says, is the challenge for the designer. In my case, this holds true. There are many tools and resources that novice programmers can use to aid their learning, and my product has the opportunity to work alongside school and university courses. However, a badly designed product could hinder, rather than help, the path to learning. Programming concepts can be complex, but a well designed User Interface could help to simplify those concepts.

Norman goes on to codify seven fundamental principles of designing interactions with users (Norman, 2001, p 72). We will discuss the most important three:

- Feedback
  In the writer Steve Krug's book on web usability, *Don't Make me Think!*, he wrote that good assistance is timely. For my system, this means that live analysis is a must, with tooltips shown quickly after relevant code is typed. Secondly, good assistance is brief. My system has the option of giving lots of information about the user's code. However, keeping it brief, with an option of giving more information, limits distractions. Finally, good assistance is unavoidable. Using size, colour and placement, a design can be achieved where the user is always alerted to the new information available to them. (Krug, 2014, p 47)

- Conceptual Model
  The conceptual model of a system is the user's model of a system. While it may be abstracted away from the reality of the technology itself (for exactly a file directory on a computer). According to Norman, it does not matter if the model is abstracted, as long as it is useful to help the user understand and remember the system's functionality. With my system, the abstraction is between the model my system provides regarding data in C-like languages, and the underlying reality of how those systems are actually implemented.

- Signifiers
  Signifiers communicate where, and what, action can take place on the system. Krug says that, for example, clickable elements must be obviously so (Figure 2.1). Everyone interface element that a user has to consciously think about takes away from the task at hand, so it must be obvious how signifiers can be interacted with.

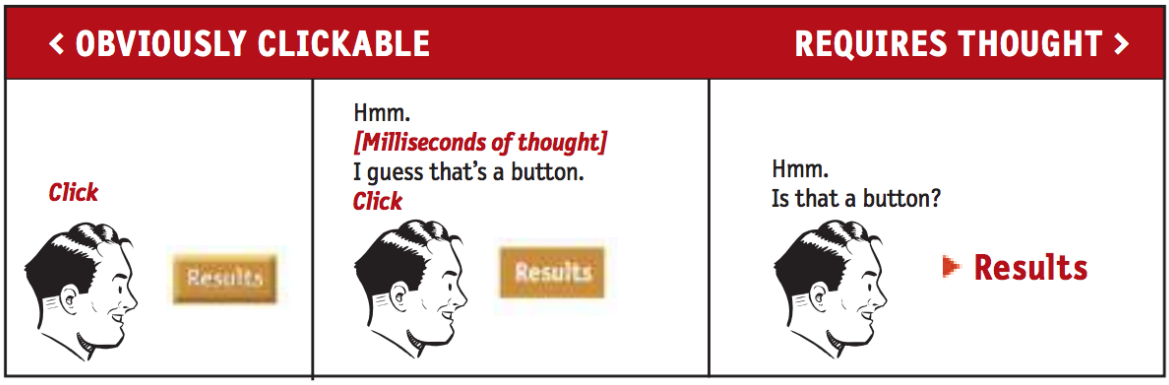

FIGURE 2.1. Obvious action in clickable elements. Figure reproduced from Krug, 2014.

2.2 Competing Products

Before undertaking this project, I looked at a set of competitors, who may have implemented something similar to what I look to do. While I initially looked at Linters, many are distinct from the kind of work I seek to do. Originally, Lint was produced as "a command which examines C source programs, detecting a number of bugs and obscurities" (**Johnson78lint**), and most Linters for C++ stick to this original idea. However, my work is not focused on providing information at times when code may be buggy, instead allowing users to have a better visualization of what the code is doing. We will look at three similar tools.

2.2.1 Visual Assist

Visual Assist is a licensed IDE, which improves on the features provided by Visual Studio code. However, one of its features is relevant here. Visual Assist allows you to search throughout either the current source file, or the whole project, for references to the current variable (similar to my last-write analysis). However, each reference is differentiated by whether it is a read (blue) or a write (pink). The main difference between this and my implementation is the fact that mine will not find all references, instead finding relevant previous writes to the current variable through control flow analysis.

2.2.2 IntelliJ IDEA

IntelliJ IDEA is a Java IDE produced by Jetbrains. While it is not useful for programming in C++, it allows data flow analysis of Java code. So, with the 'Dataflow to here' tool, you can go backwards showing the previous assignments and method calls for the current variable. Since

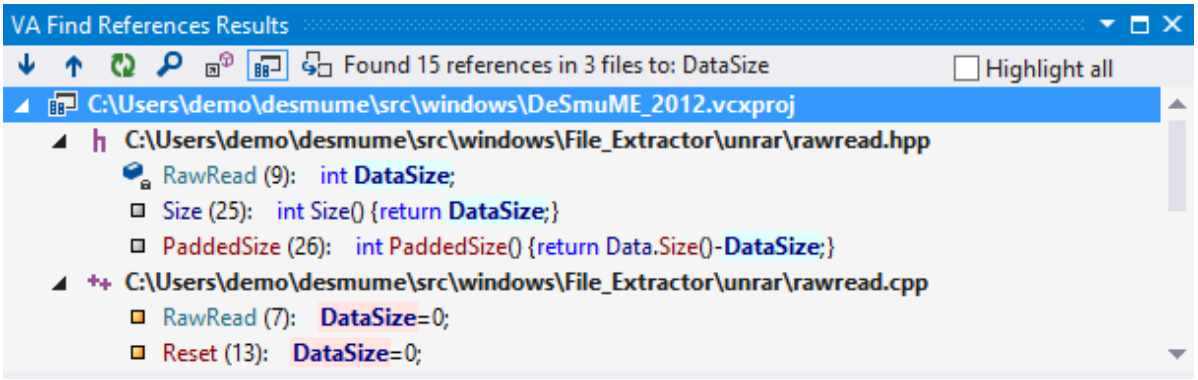

FIGURE 2.2. Finding references with Visual Assist. Figure reproduced from Krug, 2014.

the software is proprietary, however, there is no possibility to examine or use the technology used in the tool. However, by examining the UI of the tool, and of the IDE in general, I can help create a better User Experience in my final design. For example, the hierarchical view used by the 'Dataflow to here' tool could be modified for my design.

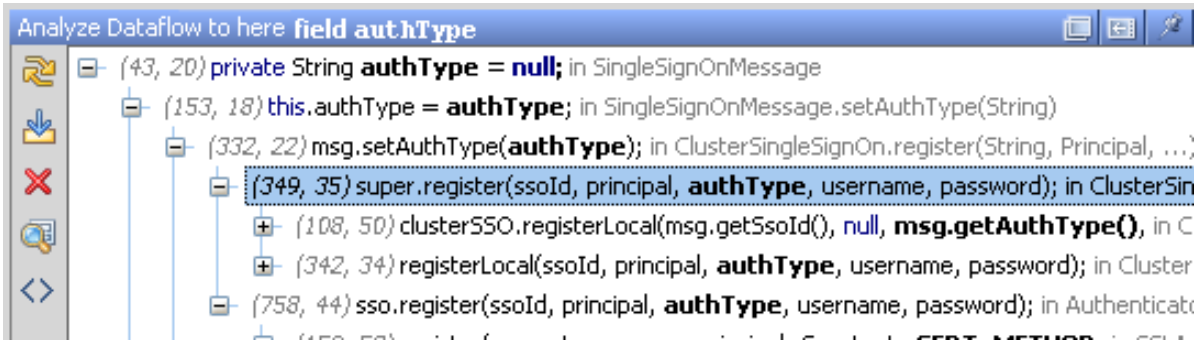

FIGURE 2.3. 'Dataflow to here' tool with IntelliJ IDEA. Figure reproduced from Jetbrains, n.d.

2.2.3 Jeliot-C

Starting from Jeliot, a program visualization tool for Java from the University of Joensuu, Finland, a visualization for C programs was created (Kirby, Toland, and Deegan, n.d., p1). Jeliot itself visualizes classes, and other OOP constructs (FIG), but the aim for Jeliot-C was to visualize C-type constructs such as pointers, making the aim similar to my graphing of pointer declarations. The visualization tool used for Jeliot-C was Visual InterPreter - a visualization of a subset of C++, called C–. While much of this work seems to supercede my work into the graphing of pointer declarations, there has been no known work on Jeliot-C after an initial prototype in 2010. Since this prototype is unavailable, it is safe to therefore assume that my work will break at least some new ground into code visualization.

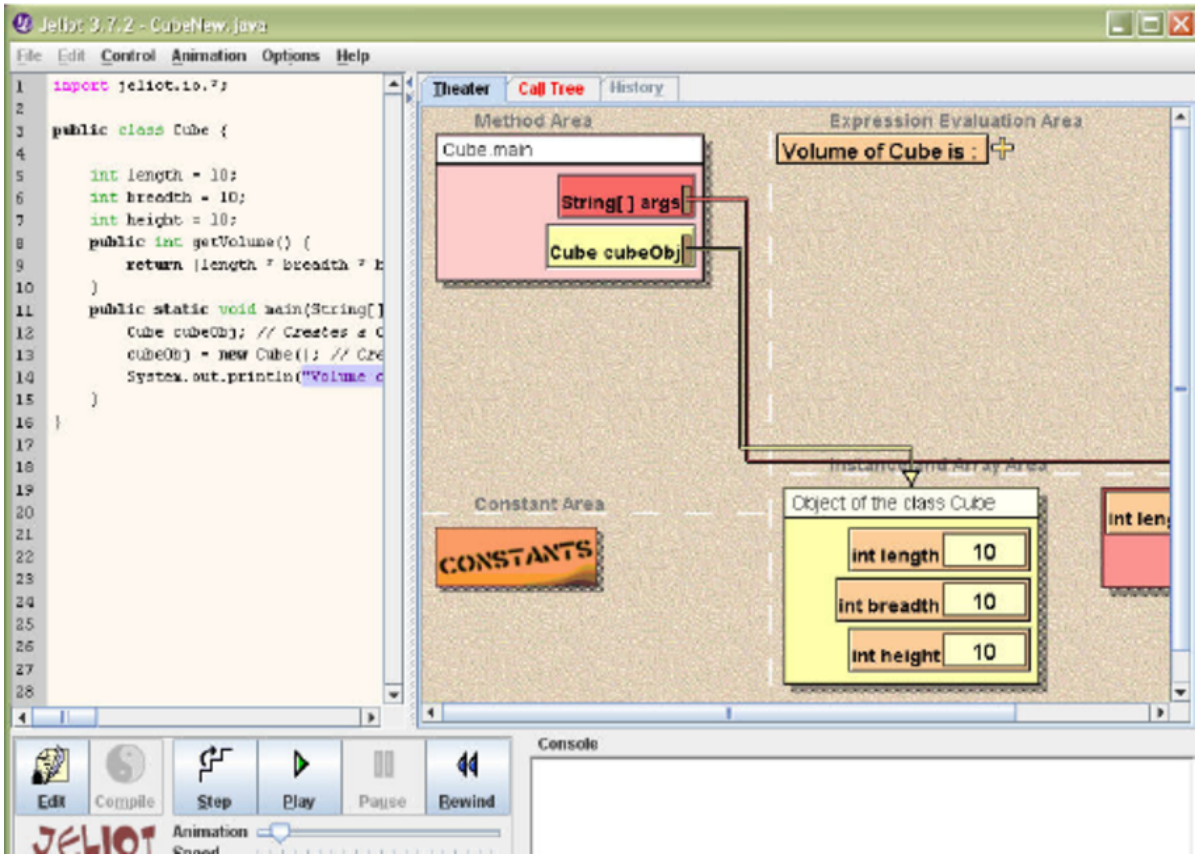

FIGURE 2.4. Example of Java visualisation with Jeliot. Figure reproduced from **\IeC {\textflorin }\r aisar_pinter_radosav_2011**

CHAPTER 3

BACKEND INFRASTRUCTURE

3.1 C++ Parser

For the purposes of this project, the output of the parser needs to be an abstract syntax tree, which the application has the capability of referring to. It can do this in two ways. The first, more sophisticated way, is through specific bindings to the AST. While this could be built around a parser in .NET, since F# is a language built for the .NET framework, FABLE itself is built primarily around Node.js. So, it would make sense to find bindings for the parser in C++. The second way would be to turn the complex C++ source into a more manageable data format (such as XML or JSON). This could be undertaken either using a Node.js module, or instead could be distinct from the rest of the infrastructure. We will explore examples of both.

In addition, there is the option to not parse C++ (owing to the complexity of the task) and instead just to use a parser of C. While this may become useful if project timings are an issue, if a reliable C++ parser is available, it is preferred. This is because C++ is a superset of C and, if time allows, there may be a possibility of focusing on some C++ only constructs within implementation.

Finally, the AST itself would preferably be quite close to the original C++ code. For example, if the AST preserves the original variable names, it becomes easier to refer back to these names when giving feedback to the user. Using these rough guidelines, possible solutions can now be explored. To compare solutions, I will parse four pieces of source code, seen in Appendix A.

3.1.1 CastXML

Initially, CastXML seems like an obvious choice. Described as a "C-family abstract syntax tree XML output tool" (King, 2017), the open source project is built on top of the Clang Compiler. On class declarations (Listing 3.1), the AST shows the structure of the classes well, and doesn't mangle variable names.

Listing 3.1: Part of the CastXML representation of code found in Appendix A.4

```
<Field id="_13" name="field" type="_27" context="_7" access="public"
    location="f1:3" file="f1" line="3" offset="64"/>
```

```
<Method id="_14" name="method" returns="_28" context="_7" access="public"
    location="f1:4" file="f1" line="4" const="1" virtual="1" pure_virtual="1"
    mangled="_ZNK7MyClass6methodEv"/>
<Variable id="_15" name="static_field" type="_27c" context="_7" access="public"
    location="f1:6" file="f1" line="6" static="1"
    mangled="_ZN7MyClass12static_fieldE"/>
<Method id="_16" name="static_method" returns="_27" context="_7" access="public"
    location="f1:7" file="f1" line="7" static="1"
    mangled="_ZN7MyClass13static_methodEv"/>
<OperatorMethod id="_17" name="=" returns="_29" context="_7" access="public"
    location="f1:1" file="f1" line="1" inline="1" artificial="1" throw=""
    mangled="_ZN7MyClassaSERKS_">
  <Argument type="_30" location="f1:1" file="f1" line="1"/>
</OperatorMethod>
<Destructor id="_18" name="MyClass" context="_7" access="public" location="f1:1"
    file="f1" line="1" inline="1" artificial="1" throw=""/>
<Constructor id="_19" name="MyClass" context="_7" access="public" location="f1:1"
    file="f1" line="1" inline="1" artificial="1" throw=""/>
<Constructor id="_20" name="MyClass" context="_7" access="public" location="f1:1"
    file="f1" line="1" inline="1" artificial="1" throw="">
  <Argument type="_30" location="f1:1" file="f1" line="1"/>
</Constructor>
```

However, the main issue with CastXML is that the representation is high level, with function bodies not being shown in the XML at all. For some static analysis, for example one which only concerns control flow between functions, this would cause no issue, however much of the analysis I seek to undertake needs an AST representation of function bodies. So CastXML is not a suitable parser.

CHAPTER 4

IMPLEMENTATION PLAN

4.1 Completed Work

Having settled on a C++ parser implementation with libclang, I tried to interface with it. However, initially, the example code did not work. In addition, it did not seem that there was an interface to get any variable types from libclang. Since this was the only working node-libclang interface I could find, I needed to patch up both of these issues. The first took some time, since the example code given was completely wrong. The second was easier to solve, but still needed an in-depth understanding of the interface to follow a path through it to libclang itself. In addition, it is clear that others have had this issue, with one other person opening a git issue along the same line as me. So, from here I submitted a pull request, and I hope to become a contributor to the interface itself. From here, we have a relatively stable backend infrastructure on which to write code on top of.

4.2 Future Milestones

There are three main sections of the rest of the project, now that the C++ parser has been interfaced to. The first is parsing pointer declarations on the backend. The steps for this are as follows:

(I) Write a set of inputs the problem as a whole, along with the correct completed graph as an F# data structure

(II) Write NUnit tests from the created inputs/outputs

(III) Write code to pass basic pointer statements, with no array declarations

(IV) Write code to pass pointer statements with keywords (const, volatile, etc)

(V) Write code to pass pointer statements with function pointers

(VI) Add recursion to the code, so nested functions can be parsed

The second is undertaking control flow analysis to ascertain the possible previous reads of each variable at any one time. The steps for this are as follows:

(I) Create a reliable infrastructure to make control flow graphs for a given C++ source file.

(II) Create source files to cover the case of any single variable at a given line, along with the correct possibilities of where the last-write occurred.

(III) Write NUnit tests from the created inputs/outputs.

(IV) Use the control flow graphs to pass the tests.

(V) Add the variables that change where the last-write occurred to the NUnit tests.

(VI) Write code to pass these tests.

(VII) Add the possibility of ascertaining the last-writes of arrays and vectors to the tests.

(VIII) Write code to pass these tests.

The final section is producing a User Interface to interface to the back-end elements. This must contain a live code-editor, along with a system to produce pointer declaration graphs, and a system to show the last-writes of variables.

(I) Draw and iterate on designs for the user interface as a whole, either in software or on paper.

(II) Organise initial small-scale user testing for these drawings.

(III) Work out the best system on the front-end to produce pointer declaration graphs, and try converting an F# data structure to a graph on the front-end

(IV) Once testing has taken place, refine the drawings

(V) Implement the drawings in skeleton HTML/Javascript/CSS

(VI) Interface to the backend in it's most basic form.

(VII) Organise further testing for the final product.

Now, for each week in turn, I will estimate how far I can complete on each of these sections:

- 9th-14th May
  Pointer Declarations II
  Last-Write Analysis - II
  User Interface - I

- 15th-21st May
  Pointer Declarations III
  Last-Write Analysis - IV
  User Interface - III

- 22nd-28th May
  Pointer Declarations IV
  Last-Write Analysis - V
  User Interface - V
- 29th May - 4th June
  Pointer Declarations V
  Last-Write Analysis - VII
  User Interface - VI
- 5th-11th June
  Pointer Declarations VI
  Last-Write Analysis - VIII
  User Interface - VII
- 12th-21st June
  User Testing, Report Writing and Buffer Time

4.3 Risks and Fallbacks

While I have just over a week for buffer time and report writing, there are still risks associated with the project. The main one at the moment is the possibility of an unreliable or unhelpful C++ parser at the backend. However, since I plan to start interfacing with it fully by the second week, if issues occur, I can switch to a C parser quickly, and scale back the project.

There are still two other foreign interfaces I am yet to deal with. The first is the production of control flow graphs for last-write analysis. While there are possibilities, such as LLVM or GraphVIZ, this could still be tricky to deal with. However, as my plan has me interfacing with these modules in the first week, I can have a change of tack soon if this causes an issue. If there is an issue, I would aim to resolve it in the second week (pushing back the rest of the deadlines for the last-write section). If there was still no progress, I would likely remove this section from the project, focusing fully on pointer declarations.

The second foreign interface is creating directed graphs in Javascript on the front end. This is relatively low risk, since there are many options, including d3.js and Cytoscape.js. Getting to this on the second week, if there is an issue with this the fallback would be to just output text with arrows pointing between them.

The only other risk is running out of time. Parts of the plan, including function pointer for pointer declaration, and skeleton UI creation in HTML/CSS/JS, could over-run if care is not taken. In addition, large sections of the report need to be written as I go along. So, if time does become tight, last-write analysis would become a smaller element, either ending at part VII, or even part V. Also, the most complex forms of function pointer do not necessarily need to be parsed. Thus, if time becomes an issue, here is where cut backs occur.

CHAPTER 5

EVALUATION PLAN

5.1 Coverage Testing

Of course, it is important for the system to be correct. With regards to pointer declarations, the problem is closed, but a correct system will produce the correct graph for any pointer declaration. With regards to last-write analysis, the problem is somewhat more open. However, a correct system must still show all possible last writes of a given variable, along with all the variables which affect the path of the last-write. Our testing framework does not necessarily need to be extensive, however. The nature of the problem allows for small mistakes to be made. However, it would be helpful to have a large coverage of tests over most aspects of the implementation, both to aid debugging during implementation, and to confirm correctness aftweards.

With F#, there are two possibilities for testing. The first is randomized property-based testing with FsCheck. While it may be possible to work out some properties of our problems, there won't necessarily be a large coverage using property based testing. In addition, FsCheck does not easily move to Javascript, so testing the front-end would likely be put to one side if this was done. The second. is Unit Testing, with NUnit. This allows you to write single test cases, as well as parameterizing test cases, thus allowing you to iterate the test parameter over a list (Wlaschin, 2014). In addition, NUnit tests can be compiled into Javascript using FABLE (Wlaschin, 2014), thus allowing testing of both the front and back ends. The only problem with Unit Testing is that, with pointer declarations, there is no benchmark to compare tests against, so I will likely need to write the correct implementations. However, with last-write analysis, there is a possibility of marking when variables are written to, allowing tests to be automated to some degree. So then, while NUnit seems like an obvious choice, it makes sense to use a version of the Test-Driven Development model to ensure testing is undertaken early on.

5.2 User Testing

Related to the research into User Interface design in 2.1 is User Testing. In *Don't Make Me Think!*, Krug points out some important facts about user testing (Krug, 2014, p133-135):

- "Testing one user is 100 percent better than testing none." A simple point, but an important one. Often designers can be afraid of testing and criticism, or think it's too difficult, so Krug champions a simpler approach to encourage more user testing.
- "Testing one user early in the project is better than testing 50 near the end." Krug makes the point that, while designers do not want to test early prototypes, users can feel freer to comment on unfinished products since it's obviously still subject to change. (Krug, 2014, p145) In addition, for testing to shape the design of a product, it must be done early.
- "Testing is an iterative process." Krug reasons that, with the help of Figure 5.1, multiple smaller tests are better for the final product.

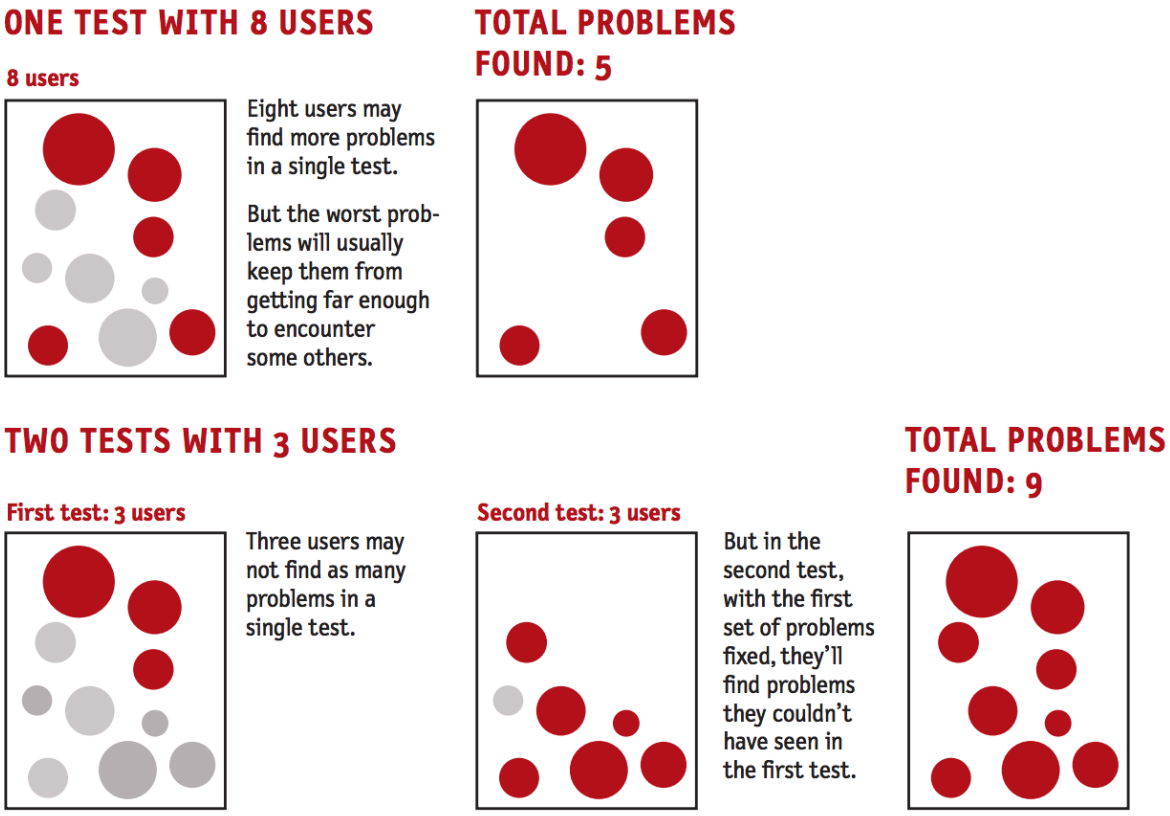

FIGURE 5.1. The case for multiple, smaller test iterations. Figure reproduced from Krug, 2014.

So then, Krug argues for simple, small iterations of testing, throughout the design process. But what does this testing look like? Firstly, "Get it" testing, where the user is shown an interface, and is asked whether they understand the unique selling point of the product, and whether they understand generally how it works without much guidance. Secondly, "Key task" testing, where users are asked to perform specific and key tasks on the system, to see how well the system performs (Krug, 2014, p144). In my evaluation, I expect to perform at least two rounds of testing.

The first, earlier round will be to see the usefulness of the tools, and how they can be changed to be more useful ("Get it" testing), as well as some "Key task" testing to refine the design of the user interface. The second round of testing will be more "Key task" testing, to evaluate a more finalized version of the user interface.

APPENDIX A: CODE SAMPLES FOR C++ PARSER TESTING

A.1 Variable Printing

Simple Source file which declares and prints a variable

```
#include <iostream>
int main() {
    int x = 10;
    std::cout << x << std::endl;
    return 0;
}
```

A.2 Pointer Declaration

File to test a relatively complex pointer declaration

```
int main() {
    char *(*fp)( int, float *);
    return 0;
}
```

A.3 Control Flow

File to test basic control flow with an if statement

```
#include <iostream>
int main() {
    int x = 10;
    int y = x*5/2;
    if (y > 20) {
        x = 7;
    }
    return 0;
}
```

A.4 Classes

Testing capabilities with C++, through a class declaration

```
class MyClass {
  public:
    int field;
    virtual void method() const = 0;

    static const int static_field;
    static int static_method();
};
```

Bibliography

Anderson, David (1994). *The Clockwise/Spiral Rule*. URL: http://c-faq.com/decl/spiral.anderson.html.

Jetbrains (n.d.). *Analyzing Data Flow*. URL: https://www.jetbrains.com/help/idea/2017.1/analyzing-data-flow.html#d831036e121.

King, Brad (2017). *CastXML*. URL: https://github.com/CastXML/CastXML.

Kirby, Stephen, Benjamin Toland, and Catherine Deegan (n.d.). "Program Visualisation tool for teaching programming in C". In:

Krug, Steve (2014). *Don't Make Me Think!* 2nd ed. New Riders.

Norman, Don (2001). *The Design of Everyday Things*. MIT.

Wlaschin, Scott (2014). *Using F# for testing*. URL: https://fsharpforfunandprofit.com/posts/low-risk-ways-to-use-fsharp-at-work-3.