View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Union College: Union | Digital Works

Union College
Union | Digital Works

Honors Theses Student Work

6-2011

Pedagogical Tool for Usability Science Final Project Report

Daniel D. Mendelsohn
Union College - Schenectady, NY

Follow this and additional works at: https://digitalworks.union.edu/theses

Part of the Graphics and Human Computer Interfaces Commons

Recommended Citation
Mendelsohn, Daniel D., "Pedagogical Tool for Usability Science Final Project Report" (2011). *Honors Theses*. 1033.
https://digitalworks.union.edu/theses/1033

This Open Access is brought to you for free and open access by the Student Work at Union | Digital Works. It has been accepted for inclusion in Honors Theses by an authorized administrator of Union | Digital Works. For more information, please contact digitalworks@union.edu.

Pedagogical Tool for Usability Science

Final Project Report

By

Dan Mendelsohn

****

Submitted in partial fulfillment of requirements for honors in the department of Computer Engineering

Union College

June, 2011

Table of Contents:

Abstract ………4

I. Introduction……5

1. Usability……5

1.1 Why is Usability Important?..6

1.2 Motivation………………………………………………………………………………………………………7

II. Background……8

1. Usability Testing in the Professional World………………………………………………………………8

2. Pedagogical Tool for Usability Science: What Has Been Done……………………………………9

III. Goals ………...11

1. Hardware Goals………………………………………………………………………………………………….11

2. Software goals ………………………………………………………………………………………………...12

3. Cost Goals……………………………………………………………………………………………………..13

IV. Design Specification…………………………………………………………………………………………………14

V. Final Implementation……………………………………………………………………………………………….14

VI. Results………18

VII. Future Work…….19

VIII. User Guide……21

1. Programming in DaNick ………………………………………………………………………………………21

1.1 Error Descriptions…………………………………………………………………………………24

2. Preparing the Widget Description File………………………………………………………………...25

3. Loading Programs Onto the Tini………………………………………………………………………..27

4. Running a program……………………………………………………………………………………………27

IX. Development Guide…………………………………………………………………………………………………28

1. Creating a new Widget……………………………………………………………………………………..28

1.1. Widget:Hardware……………………………………………………………………………………….29

1.2. Widget:software……………………………………………………………………………………….31

2.Interpreter Development………………………………………………………………………………………..32

3. Maintaining the Development System…………………………………………………………………..34

4. A Note on Java Versions……………………………………………………………………………………….35

5. Components………………………………………………………………………………………………………..35

X. References……..36

Table of Figures:

Figure 1: A stove with bad usability……………………………………………………………………………………5

Figure 2: A stove with good usability…………………………………………………………………………………6

Figure 3: Cardboard stove concept………………………………………………………………………………….…7

Figure 4: Interaction and goals for mRUE……………………………………………………………………………9

Figure 5: Usability boar with two widgets attached…………………………………………………………….10

Figure 6: Cost Breakdown………………………………………………………………………………………………….13

Figure 7: A line of actual DaNick code split into different tokens……………………………………………15

Figure 8: An example of using BNF rules to identify a line of DaNick………………………………………15

Figure 9: Block diagram representing the software before runtime……………………………………….16

Figure 10: Block diagram representing the software during runtime……………………………………..17

Figure 11: Sample DaNick code……………………………………………………………………………………………22

Figure 12: A sample widget description file……………………………………………………………………………26

Figure 13: A PCB design in Eagle…………………………………………………………………………………………29

Figure 14: A finished PCB…………………………………………………………………………………………………….30

Figure 15: A completed widget……………………………………………………………………………………………31

Figure 16: Full BNF defining DaNick……………………………………………………………………………………….33

Abstract

MENDELSOHN DAN Pedagigical Tool for Usability Science: Designing and building a better way to test usability. Department of Computer engineering June, 2011

Advisors: [James Hedrick, Chris Fernandes, Aaron Cass]

A Sophomore Research Seminar (SRS) at Union College teaches about usability science, the study of designing interfaces that allow the user to accomplish a given task with less time and frustration. In this context, an interface can be anything that allows interaction with a physical or virtual device such as a web browser or the knobs on a stove.

In this SRS, students design interface mockups, called prototypes, out of inexpensive material such as cardboard. Students use these prototypes to test their interfaces on real people, who are asked to perform a task that would be performed on a real appliance. The researcher physically interacts with the prototype to simulate the function of the appliance. The purpose is to gather data, like the amount of time or attempts it takes the user to accomplish their task. The problem with this method of usability testing is that the researcher's interaction can affect the validity of the data.

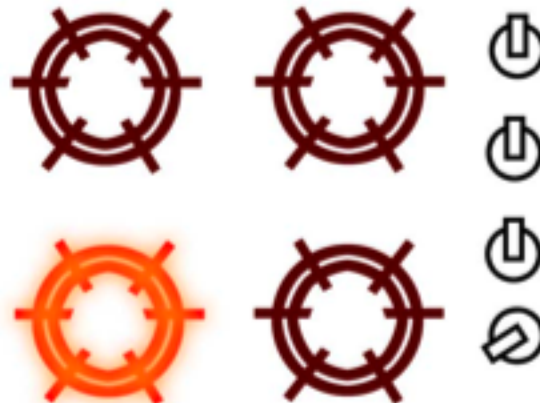
The goal of my project is to develop a system that allows the students to create prototypes that do not require interaction during testing. This involves building devices called widgets: physical devices such as LEDs or switches that represent components of household appliances. I'm also developing a programming language that defines the interaction between

widgets. Using both parts students will be able to design working mockups of household appliances.

I. Introduction

1. Usability

Productivity has always been one of the main goals in the engineering world. It is not smart to work on a project that has little to no progress being made for any amount of work put in. One aspect of today's lifestyle that people believe would add to productivity is technological advances. By replacing people with computers or by adding technological devices to the work force we may be able to accomplish our goals faster. This however is not true. There has been evidence that simply adding technology to the workforce does not actually result in better productivity (1). The reason for this is that technology is not always designed effectively. This is where usability comes in. Usability is the science of designing interfaces to increase performance. What this means is that any device that a user interacts with is designed in such a way that the user can perform whatever task they need in the most efficient way possible. This concept has been incorporated into a lot of different areas today. One example would be the layout of a stove. If someone looked at the stove in figure 1, it would be difficult for him or her to figure out which knob turns on which burner. In figure 2 however, it is very clear which knobs affect which burners.



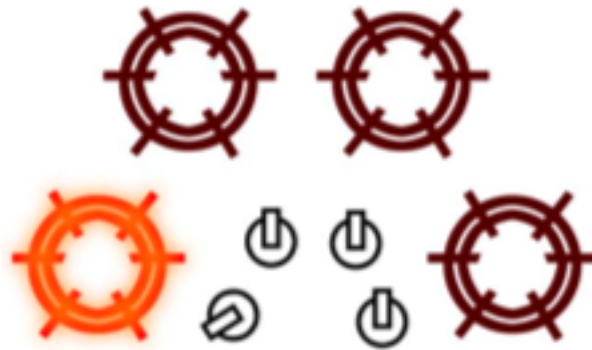


Figure 2: A stove with good usability (3)

Another example would be the standardization of car buttons. Every car has the blinker control on the left of the steering wheel. Pushing the lever up turns the right signal on and pushing it down turns the left signal on. This design has high usability as it becomes intuitive to drivers. If I have ever used a car, then I can get into another and know exactly how to work the turn signals.

1.1 Why is Usability Important

Usability is important so that users can effectively use a piece of technology. It is important for someone to be able to intuitively know how to use something if its use is not designed to be a thought provoking activity. Productivity is increased when something is easy to use as there is less thought going into how to operate, and more going into what is being accomplished. There are plenty of people in the world today who would not have gotten by on a computer that is run by a terminal mechanism. There are too many commands to remember and a lot of ways to mess them up. Today we have graphical user interfaces or GUIs, which allow people to point and click. They can drag something from one folder to another instead of remembering to type something like “cd home/work_folder”. This is extremely important, as computers have become a part of everyday life in almost every workplace. Usability has allowed people to advance with technology even if they are not technologically adept.

1.2 Motivation

The motivation behind working on a usability project comes from Professors Aaron Cass and Chris Fernandes. These two professors are currently teaching a class on usability for sophomores at Union College. Their course works with students to create objects such as alarm clocks or stoves that are very usable (3). In order to study usability, students make mock-ups, or prototypes, out of inexpensive material such as cardboard. These students then test their design by asking human subjects to perform certain tasks. One example would be a researcher asking a participant to turn on the bottom-left burner of a stove. The participant would then turn a knob that they believed corresponded to the given burner, and the researcher would place a red circle over whichever burner the participant activated.

Figure 3. Cardboard Stove concept (3)

There are two problems with this method. First, the process of changing physical components can distract the user and possibly confuse or stall them. Second, the amount of time it takes to turn the burner on is dependent on how quickly the researcher moves. This is harmful to data collection if you

are testing how long it takes for the user to figure out which knob does what. In order to fix these problems an electronic board was created to help students create prototypes, which require no human interaction.

II. Background

1. Usability Testing Methods in the Professional World

The primary accepted method of testing for good usability is called usability lab evaluation (4). In this method, the subject and researcher stand in the same room separated by a one-way mirror or curtain. The user and tester can converse before, during and after the test is being given. There are also video cameras in the room documenting what the subject does when given specific commands. After the experiment is completed, the video is analyzed and the product is changed accordingly.

Another method of usability testing is known as moderated remote usability evaluation or mRUE(4). In mRUE, the subject and researcher operate over a network and can be separated by room, state or even country. This allows the subject to be tested in a space that is comfortable to them, which is more likely the place that they would be using the product they were testing. Testing through mRUE is synchronous, which means that the researcher and subject are both logged on and participating in the study at the same time.

A third method of usability testing is asynchronous remote usability evaluation, or aRUE (4). In this method the test is done similar to mRUE except the researcher and subject are able to log on at different times. The researcher can set up the test and upload it, and the subject can then be tested whenever they have the time to do so. This method allows for better flexibility, but has been proven less effective than mRUE (4).

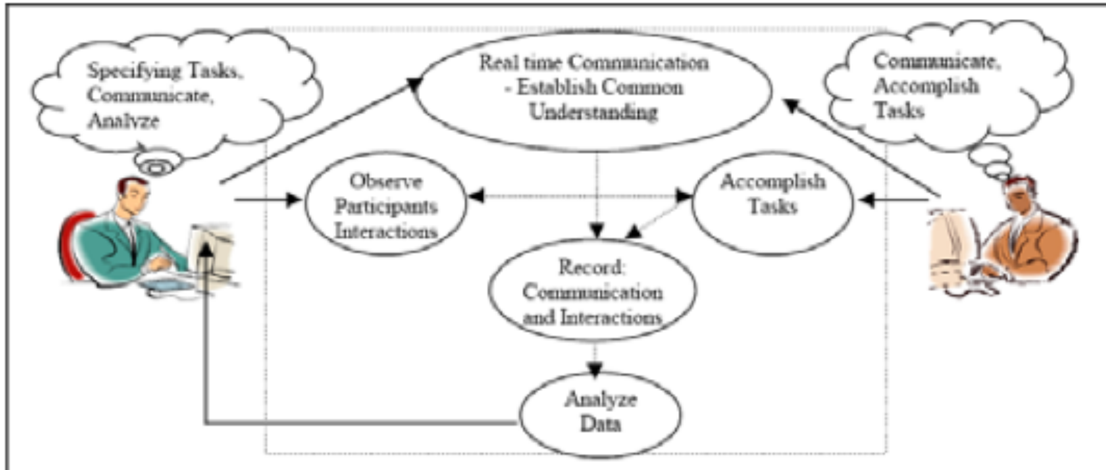


Figure 4: Interaction and goals for mRUE (4)

2. Pedagogical Tool for Usability: What Has Been Done

The work did build on the work done by other students. Two students: Susan Beckhardt and Nick Potvin have worked previously on creating this tool. Before my involvement, the tool consisted of four general-purpose input ports as well as a few “widgets” which can be connected. Widgets can be any device such as LEDs, 7 segment displays, potentiometers, buttons, etc. These widgets are used in order to represent some component of a household appliance.

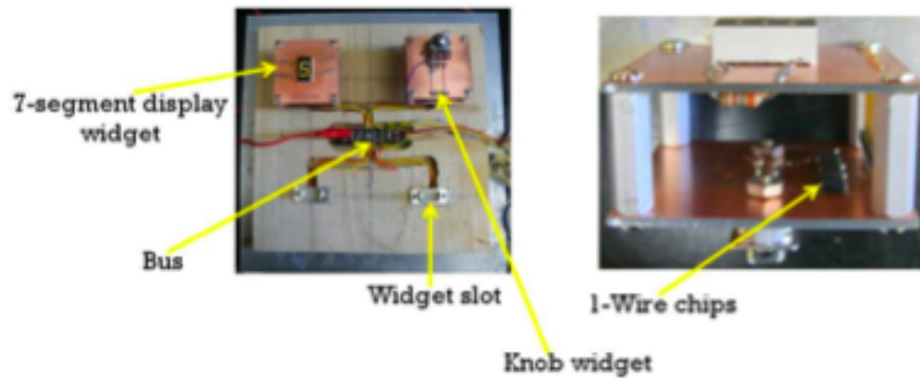


Figure 5: Usability board with two widgets attached (3)

One example would be using two LEDs and two switches to represent a two-burner stove. The board is designed so that these LEDs and Switches can be placed on any of the four slots and programmed to act however the designer wishes. This makes it possible for the designer to choose where to put these components and how they interact in order to create a device. In this case the LEDs could represent burners on a stove and the switches could be knobs. The designers chose to create this board by combining the TINI microcontroller with a one-wire bus system. This design allows a designer to write code in java and run it through a single data wire to the usability board. Nick Potvin has also designed a specialized programming language that allows the different widgets to interact and work together with less complexity than java.

III. Goals:

In this section I will discuss the goals for both the hardware and software aspect of this project. I will be explaining what specifications I planned on for each aspect and describe why these specifications were chosen. I will also be addressing the goal of the overall cost of this project.

1. Hardware Goals:

The purpose of this project is to create a way in which a student can create a prototype of a household appliance that does not require human interaction to simulate functionality. In order to do this it is important to specify how the hardware should look, feel, and function.

In order to make a realistic prototype creation tool for household appliances, we need to make sure that the devices we use (Widgets) are realistic. By this I mean that they effectively represent certain aspects of household appliances. This means that the widgets must be easy to recognize and associate with a concrete aspect of a given appliance. One example of this would be modeling a stove burner with an LED. Though this device does not generate heat like a stove does, it does glow when turned on. This is effective because a stove burner will glow when activated. Another example is representing a stove knob with a potentiometer, which is similar to a knob that one would find on an actual stove.

The other aspect of the look is the layout of the electronic board itself. In order for this board to allow for an effective modeling of an appliance it has to be very versatile. This means that the board must allow for the widgets to be connected in many different ways in order to

allow the device to actually look somewhat like the final product. The board needs to have enough connection points that multiple different layouts of the same household appliance is achievable.

The feel and function of the project is important as well. In order to test how usable a prototype is, it must feel like the actual thing. This means that we cannot use a button to represent a knob. We need to use widgets that feel like the actual part of the appliance being represented. We also need to avoid delay. In order to do this the hardware needs to act as an actual appliance would when prompted to. This means that turning a knob to activate an LED should be instantaneous and without delay.

Overall, it is important to make sure that the usability board looks, feels, and functions just like the final product would. It needs to be realistic and effective if it is going to be a good testing device.

2. Software Goals:

For the widgets to interact without human involvement, there needs to be some programming that defines how they connect. In the past we have used java to make this possible. There are a few problems with this method however. The goal of this project is to create a device that can be used by students to create their own prototype of a household appliance. In order to program interactions between the different components of their prototype in java, the students would have to learn the nuances of java coding, as well as write

a very large amount of code. A simple action such as a button activating an LED can take over 300 lines of Java code.

In order to deal with this problem, my goal is to create a very simple programming language, which can define the interaction between widgets. This program must be robust enough that all different forms of widget interactions are definable. It must also be simple enough for students to be able to write the program without spending too much time learning how to use the language, or writing the program itself. The end result should work exactly the same as if the program was written in java, without needing to use extensive coding for every widget interaction.

3. Cost goals:

The cost goal of this project is outlined in the following list. This was submitted for an IEF grant and received partial funding.

Total: $216

Breakdown:

- LED x 10: $1
- Potentiometer x 4: $2
- 7 segment display x 4: $4
- Buzzer x 4: $6
- Solder: $2
- colored wires: $5
- One-wire chips x 20: $50
- Pic Chips x 10: $30
- Solder paste: $36
- Package of ten Copper boards for printed circuit board fabrication: $70

Figure 6: Cost breakdown for the project

IV. Design Specification:

- All widgets must be built to conform to the one-wire protocol
- Widgets must all be of uniform size
- There must be at least 2.5” between widgets, and the space must be uniform across the board
- Board must have at least 10 slots
- Interpreter must be written in Java (since this is the only supported language on the TINI microcontroller)
- Delay between input widget change and output widget change must be less than .5s
- Input slots must be location independent

V. Final Implementation:

The main goal of this project was to be able to take our new code (DaNick) and turn it into java code that could run on the TINI microcontroller. In order to do this I first needed to research how to make an interpreter, which can understand DaNick and run java. The first method I researched was LEX and YACC. LEX allows you to write a lexical analyzer, which defines tokens to be recognized by the compiler. These tokens are things such as numbers, operators, identifiers, keywords, etc. Once these Tokens were defined they were given to YACC, which is a parser. The parser takes these tokens and compares them to a set of grammatical rules that the creator defines. Using these grammatical rules the compiler can then decide what to do with a given line of code. This system seemed to work very well, however there is a more effective version of these two for java programming. This tool is called Javacc, or Java compiler compiler. This is a program with both a lexical analyzer and a parser built into one. It is

designed for use with Java code and makes it easy to convert a new programming language into Java.

The next step was to spend a lot of time working with Javacc in order to learn exactly how to use it. I learned a lot about how the lexical analyzer works within a short period of time and within a week was able to create my own. This tool was able to identify all Tokens that I had specified for it and could print out what they were when they were discovered.

DaNick code: **Knob controls Burner on statechange by linear**

Keywords: controls on statechange by linear

Identifier: Knob, Burner

Figure 7: A line of actual DaNick code being split into different tokens

In order to make this useful, I worked with a Backus-Naur form or BNF. This allowed me to specify parsing rules for the grammar related to DaNick. This comprised of breaking lines of DaNick code into subsections and specifying names for each. Using a simple BNF format I was able to come up with the following rules:

DaNick code: **Knob controls Burner on *statechange* by <u>linear</u>**

<control> ::= <objectID> <CONTROLS> <objectID> <ON> <command> <BY> <change>

<command> ::= <HOLD> | <CLICK> | *<STCH>*

<change> ::= <u><LINEAR></u> | <INC> | <DEC> | <TOGGLE>

Figure 8: An example of using BNF rules to identify a line of DaNick

After finalizing the BNF, I started coding the rules in the interpreter. In order to make sure that this all worked properly, I programmed the javacc to print out what kind of instruction

it encountered. I also added a skip statement, which caused the program to ignore comments (which start with ~). I successfully had the new lexical analyzer/parser identifying

instructions and being able to split it into the individual parts of data that are needed to now how to perform given tasks.

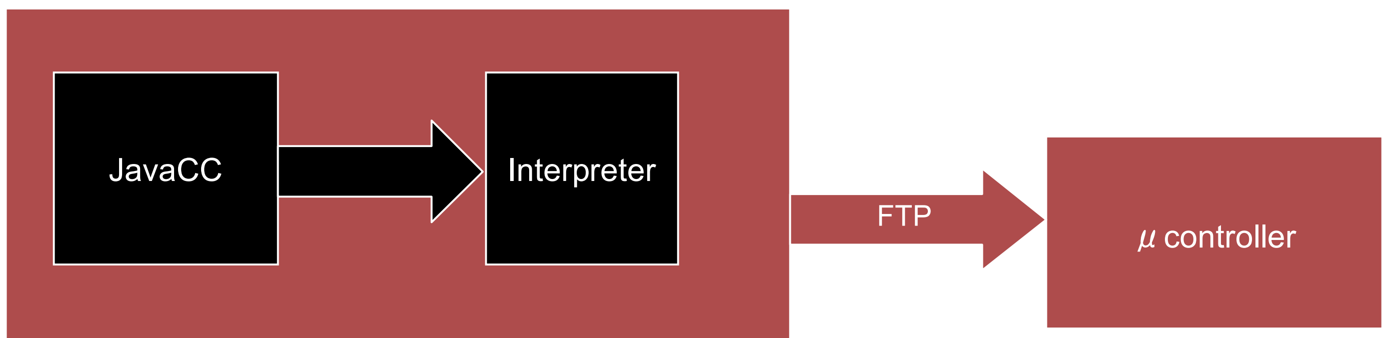


Figure 9: Block diagram representing the software interaction before runtime

Eventually, I was able to add code to the interpreter to allow the interpreter to function properly. This process involved three text files. The first was a file that contained all the necessary information about each physical widget. This file is called a widget description file, and contains information such as address, alias (specific name for the widget), and calibration values for potentiometers. The second is a file called types.txt, which has a list of al of the possible types of widgets that DaNick will represent. Using these files, we were able to read in a DaNick text file and begin to interpret it. The interpreter start by creating objects based on an interface called Widget. This allows the interpreter to initialize various variables within each

Widget. For each widget there is a java class that implements Widget and defines how it should act under given conditions. The final result allows us to take a DaNick program, run it through the interpreter, and result in a functioning program on the usability board.

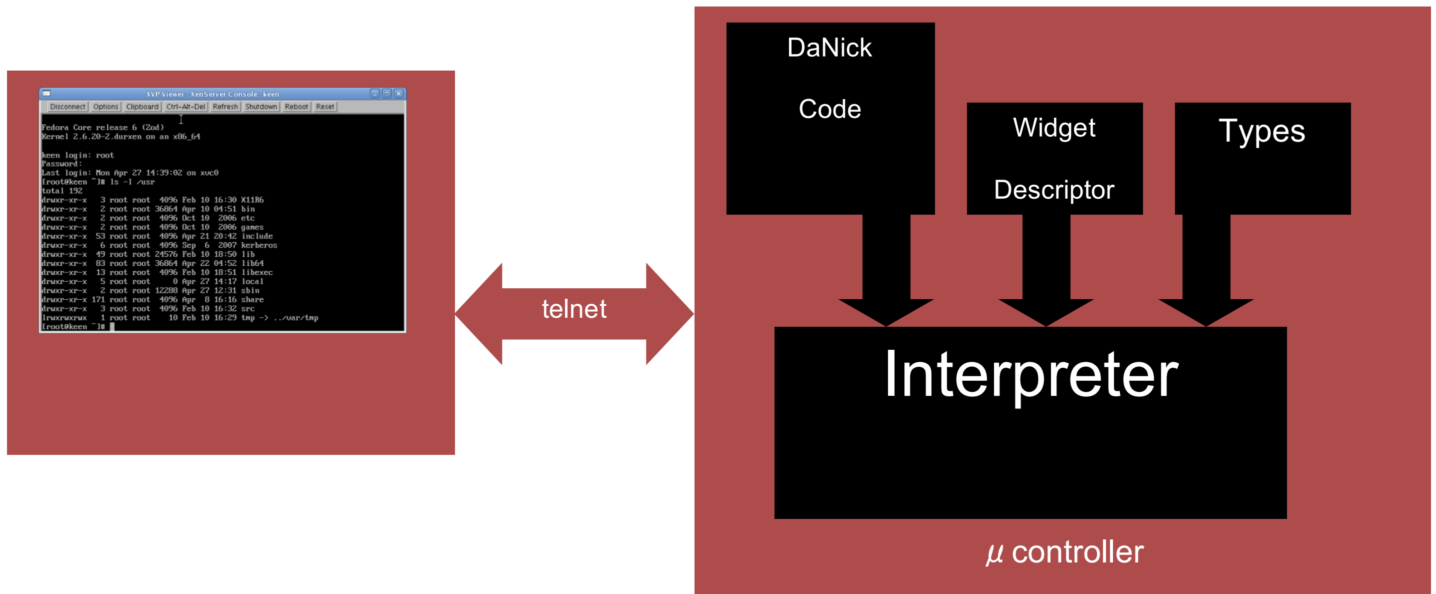


Figure 10: block diagram representing the software at runtime

VI. Results:

At the end of my time working on this project, there was a lot of progress made towards the final goal of a usability prototype tester with no human interaction. As far as the hardware is concerned, we currently have a board with location independence, which spaces the widgets evenly. This allows for versatility on the usability board, which lets students design and build multiple different layouts of the same appliance. The number of slots however does not quite meet the original plan for 10 or more.

As for the widgets themselves, they are all built to conform to the one-wire protocol. They maintain a uniform size, and are varied enough that we can represent many different components of household appliances. There is no noticeable delay between any two connected widgets, however building a packet sniffer to watch the protocol will show what the real delay between the two is.

Finally, while looking at the software aspect of the project, a lot of progress has been made. I went from only being able to program in java to building a java-based interpreter, which allows the user to program in DaNick. We are now able to write simple code that defines the interaction between widgets, which is analyzed and run through java. This allows the user to deal with much less code and run a program, which is exactly the same as if they had written the project in Java.

VII. Future Work:

Although this project has come a long way, there are still many ways that it can move forward and become more advanced. The first way to improve the project is to augment the interpreter. The interpreter can currently work for defining interactions between potentiometers and 7-segment displays. The groundwork for adding new widgets to the functionality has already been laid, and is described in detail in the users guide. In order for this to become a full interpreter of DaNick there is still a good deal of functionality that can be built in. By expanding it's functionality, the interpreter should eventually be able to handle any DaNick code that is written.

Another useful next step would be examining PIC chips for use within widgets. Currently two things limit the amount of information on a widget. First, the use of the one-wire protocol uses the same bus for both sending and receiving information to the TINI. This means that a limited amount of information can be flowing across the bus in either direction at any given time. The second is the use of the one-wire chips on the widgets. Currently, we use 4 chips for something as simple as a 7-segment display. This means that we need to search for multiple addresses and change multiple values in order to update it. By using a PIC chip, we should be able to implement some code on the widget itself and avoid using the bus. For example we could just tell a seven segment display to display a 5, then the PIC chip would know how to do this. Currently we have a book titled: Serial Communications by Roger L. Stevens. This book talks about using one-wire protocol on a PIC chip. It provides code and suggests the use of the PIC16F84A, which is probably the best place to start.

In order for this project to become a useful tool in testing usability without human interaction, we eventually need to build a logger, which will be able to track exactly when changes are made. This should be able to keep track of all operations occurring on the board with information such as timing and number of times something happens. This must be coupled with a controller than can run the tests themselves. It should know what it is looking for and be able to see timing between commands given to the participant, and when operation occurs. It should also be able to tell if the participant failed in a given task, such as turning on a specific burner.

A goal for the long term would be to convert DaNick, or the java version of the code into a GUI or graphical language. The purpose of this would be to create a user-friendly graphical representation of the board that allows people with no programming experience to define interaction between widgets. The main benefit of a GUI would be that the user could physically drag a line from one widget to another and define how they interact.

Quick Fixes:

There are a few thins that can be done to improve this project quickly. These are parts of the project I would have worked on if I had a bit more time.

- Changing the name of Par.tini (for parser) to Int.tini(for interpreter)
- Use the widget description file information to eliminate the need for a types.txt file

- Keep the program running when a widget is removed so that you can rearrange the design while the program is still running. This is done through catching an exception thrown by oneWireContainer

- Add the other widgets that have been built to DaNick functionality

- Expand the board to at least 10 slots

VIII. User Guide:

In this section I will describe the steps needed in order to use the project components that are complete.

- Writing a program in DaNick in order to define the interaction between widgets

- Preparing the widget description file with all of the necessary information for each physical widget

- Loading programs onto the TINI microcontroller

- Running programs on the tini Microcontroller

1. Programming in DaNick:

Currently there is very limited functionality in DaNick programming. There are two widgets that are programmed to work using our interpreter. These are the seven-segment display and potentiometer. Here are the steps to writing a program using these two. One things to note is that every line of code, other than the final line “done” must end with a “;”.

```
~This "~" represents a comment
~Declare the seven-segment display "burner"
Line 1: seg7 burner;
Line 2: ~specify which physical widget "burner" refers to
Line 3: burner alias seg71;
Line 4: ~Specify the states and display type of the seven-segment display
Line 5: burner is seg7(0,10,numeric);
Line 6: ~Declare our potentiometer
Line 7: pot knob;
Line 8: ~Specify which physical widget "knob" refers to
Line 9:knob alias pot1;
Line 10: ~define how the two widgets interact
Line 11: knob controls burner on statechange by linear;
Line 12: done
```

Figure 11: Sample DaNick code defining a linear interaction between a potentiometer and 7-segment display

i. Declaring an object:

In order for the widgets to be recognized, you first need to declare them and give them a type (lines 1 and 7). This is done by writing the line: *type myname*. The *type* is the widget type that is being declared. Currently this can be either pot or seg7 for potentiometer or 7-segment display respectively. In the future there will be other types that can be used, such as button, LED and so on. *myname* is whatever name you choose to give the widget you are declaring. For example I named my potentiometer “knob” and my seven segment display “burner.”

ii. Specifying a Physical Widget:

In order for the TINI to communicate with the physical widget, you next need to specify the widget being used (lines 3 and 9). The name of the widget is found in the widget description file (described later). In order to specify the widget you use the code: *myname* alias *widgetname*. In this line, *myname* is the name of the widget that you already declared in step i. Using *widgetname* looks in the widget description file and checks if the *widgetname* exists. If it does, this allows the TINI to find out important information about the physical widget such as address and calibration values.

iii. Defining Widget Characteristics:

Once the widgets have been declared and specified, we can define the states in our output widgets (line 5). This is done by writing the line: *myname is type (num1, num1)* or in the case of the 7-segment display (which currently is the only functioning output widget in DaNick): *myname is type (#, #, displaytype)*. In this code, we use the type to make sure the type we are using is consistent with the widget's type. The first number is the first state we want to use. This must be within the overall range of the widget's states (for the 7-seg currently has 0-9). The second number defines the number of states. This must also be within the range available (7-seg has 10 states – the starting state). The last value *displaytype* specifies how the seven-segment display with show the information. Currently the only valid value is "numeric," in the future there will be alphabetic, hex, and many others.

iv. Defining Interactions Between Widgets:

Finally, once everything has been initialized properly, you need to define the interaction (line 11). This is done by using the line: *myinput controls myoutput on statechange by changetype*. “myinput” is the input widget described, “myoutput” is the output widget described, “changetype” defines how the interaction works. Currently the only working “changetype” is linear, however future work will allow for toggle, increment and decrement.

v. Finishing the DaNick Program:

In the end we need the word “done” to signify that the program is over (line 12). Save this code in a txt file and load it onto the TINI.

1.1. Error Descriptions

There are many errors that you may run into during the process of running a DaNick program. Here is a list of common errors you may run into and what they mean:

i. mywidget does not exist: This error shows that the widget mywidget is not defined in the widget description file

ii. mywidget is not connected: This error says one of three things. First, the widget may not actually be connected to the board. If it is however, there is a hardware problem. Either the widget itself has some problem with it (most likely the connection with the one-wire chip), or the board itself has some problem with the wiring.

iii. Mywidget is not declared: this means that the widget “mywidget” was not declared earlier in DaNick code, which is necessary before any specification happen.

iv. Exception in reading the type file: this means that there is a problem connecting to the types.txt file

v. Number format exception: This means that the format of your numbers does not conform to the necessary regulations (the number must be an int).

vi. Exception in reading the widget file: there was a problem connecting to Widget.txt

vii. Failure to change latch state: this means that a widget was disconnected while the program was running. You must restart the program when this occurs.

2. Preparing the widget description file:

In order for the program to work properly, the widgets need to be defined properly. This is done through a few simple steps. Currently the widget description file is located on the flash drive at /Dan Mendelsohn/code/make/Widget.txt.

i. Naming the Widget:

Write the widget’s alias on the first line. This is the name the program will look for when specifying the physical widget.

ii. Necessary Information:

On the second line put the address first, then any other information that is necessary. For a seven-segment display, this information is the three other addresses that the widget uses. Note: the order of these addresses matters. For the potentiometer, we need to put calibration values that define the difference between states. These values are 100* the voltage value that splits the states apart. This is necessary, as the potentiometers are not necessarily linear. The values on this line also define the number of states recognized on the potentiometer.

iii. Completing the Widget Description File:

Save this file as Widget.txt (this should have every possible widget in it, even if it is not being used for a given program).

Here is an example of a widget description file:

```
pot1
850000000F671520 3 7 100 215 270 322 350 372 388 400
seg71
18000000021D983A 740000002354A3A 98000000021C253A A50000000233F73A
```

Figure 12: A sample widget description file. This file specifies a potentiometer with 10 states, and a seven-segment display

3. Loading programs onto the TINI

In order to load the programs onto the TINI there are a few simple steps.

i. Establishing Connection:

First you need to connect to the TINI via FTP. You can do this by connecting the TINI to the network via Ethernet. Once this is done, you can use a terminal to connect by typing the command: ftp *IPaddress*. The "IPaddress" is the IP address of the TINI itself. The one that I have been using is 149.106.40.6.

ii. Logging In:

Once connected, the program will ask you for a username. The username for our TINI is "root". Once this is in, the program will ask for a password. The password for our TINI is "tini". If the password is wrong the FTP will go into a dead state and you need to quit and try again.

iii. Commands:

Once you are connected to the TINI there are two commands that we can use. First we can delete a file "filename" by using the command: *del filename*. Next we can put a file onto the TINI. This is done by using the command: *put /path/filename1 filename2*. The path is the full path starting at /home/…… that leads to the file that you are trying to load onto the TINI. The "filename1" line specifies the name of the file you are loading onto the TINI. The second filename "filename2" is the name you wish it to have on the TINI. This is usually the same as the first filename. Note: currently the four files you need to load are: Par.tini, Widget.txt, types.txt, and your DaNick program. These files can be found on the flash drive at /Dan Mendelsohn/code/make/*filename.*

iv. Closing Connection:

When you are done type: *quit* to exit the FTP connection

4. Running a program:

In order to run a program on the TINI, there are a few simple steps that need to be followed.

i. Checking connections:

Before you can do anything, you first need to make sure that the board and TINI are connected to power supplies. The board will connect to a power supply supplying 5.0V. This is done by connecting the red wire to Vcc+ and the black to Vcc-. The TINI connect using a standard AC adapter.

ii. Establishing Connection:

Next you need to connect to the TINI using telnet. This is done by using the terminal and typing the command: *telnet IPaddress*. “IPaddress” refers to the IP address of the TINI. For our TINI it is 149.106.40.6.

iii. Logging In:

Once the connection is established, you will be prompted for a userame. Our username is “root”. Next you will be prompted for a password. Our password is “tini”. Unlike FTP, if you do not enter the correct password, you will start this process over with the username.

iv. Commands:

Now that you are connected, there are a few commands that you can use. The first is the “ls” command. Using “ls” will list al of the files that are present on the file. In order to run the program we need to have Par.tini, Widget.txt, types.txt and whatever your program is called. Next, we can run the actual program by typing “java Par.tini.” This will run the Par.tini file, which is the interpreter. This program will ask you to enter a filename. This should be the name of the file holding your DaNick code. Finally, each program runs continuously until it is killed. In order to end the program you need to open a separate telnet connection using steps i-iii. Then you can type “ps” to list the running process. There will be a number before Par.tini if you are currently running it. In order to end the program type “kill #” where the # is the number before Par.tini.

v. Closing Connection:

Finally, when you are done type “logout” in order to close the connection with telnet.

IX. Development Guide

In this section I will describe how to develop the parts of this project that I created. This includes Creating a new widget, interpreter development, Maintaining the development system, and a note on Java versions.

1. Creating a new widget:

In order to create a new widget there are two processes. First you need to make the widget hardware itself. Then you need to add the widgets functionality to the interpreter.

1.1. Widget: hardware:

i. Designing the Circuit:

When creating a widget we start with a simple circuit diagram. This diagram must include connection to the 9-pin connector, and use of the one-wire chip(s).

ii. Designing the PCB:

Next you need to use a program called Eagle to create a PCB layout of the given circuit. This allows you to design a PCB with two sides of connections.

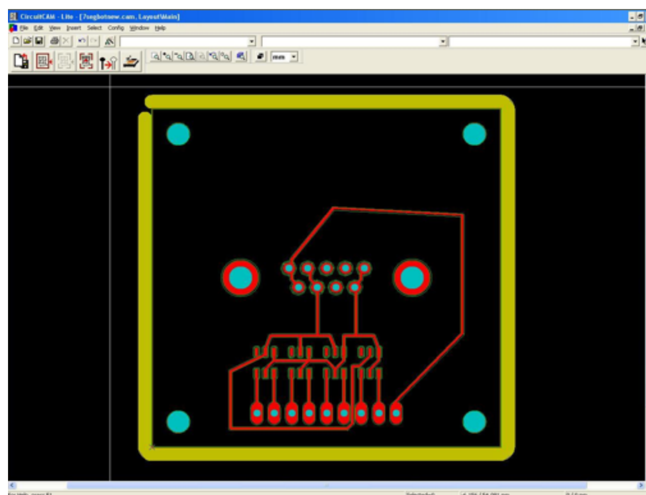


Figure 13: A PCB design in eagle (this is the bottom of the seven-segment display)

iii. Creating the PCB:

Once the PCB has been designed we can use Board Master and Union's PCB fabrication system. This allows you to create PCBs for each widget. There should be 2 PCBs per widget.

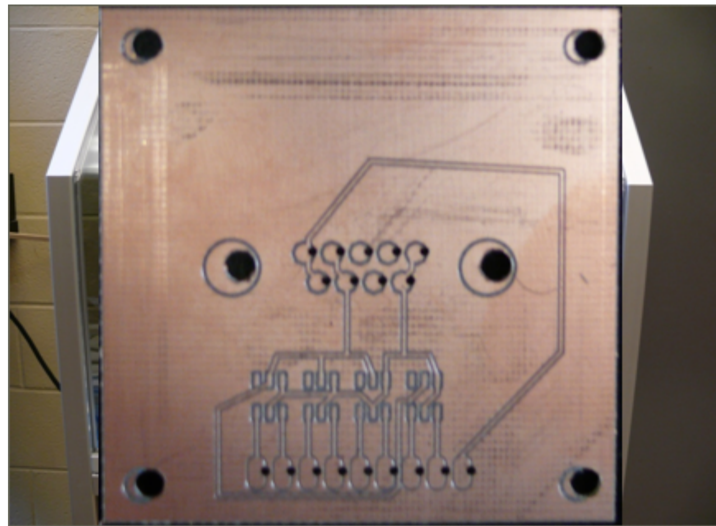


Figure 14: A finished PCB (7-segment display)

iv. Surface Mounting the Chip:

Once you have the PCBs made it is time to surface-mount the one-wire chips. This is done using solder paste. Using a solder paste guide found here: http://www.seattlerobotics.org/encoder/200006/oven_art.htm you should be able to surface mount a chip using solder paste, and a conventional toaster oven.

v. Connecting the Rest of the Components:

Once the chip has been mounted it is time to solder the rest of the components to the board. This is done by hand. Note: when connecting the 9-pin connection we have a specific way the pins need to be connected. If the numbering scheme is 1-5(across the 5 pin part), then 6-9 (across the 4pin part) from left to right, then we need to connect: 1 and 6 to Vcc, 2 and 7 to data, and 5 and 9 to GND.

vi. Finishing the Widget**:**

Finally, using the uniform spacers, we connect the entire thing together in order to complete the widget.

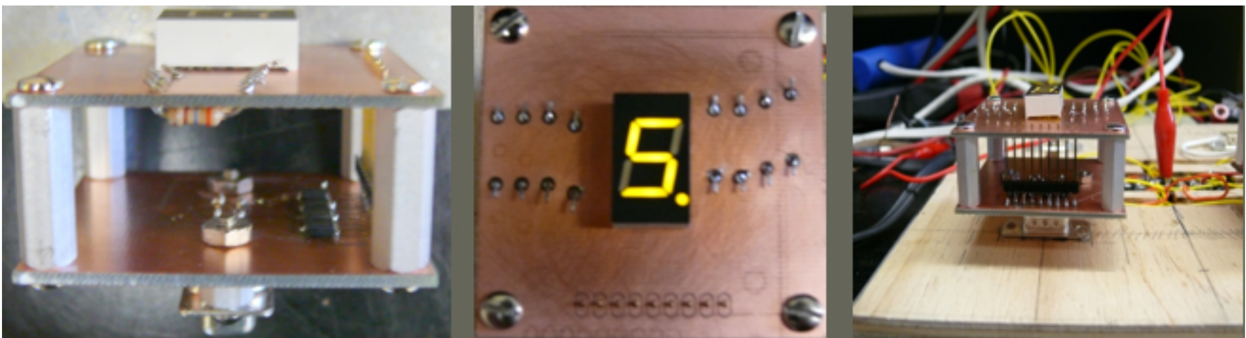


Figure 15: A completed widget from three different angles

1.2. Widget: software:

In order to use the widget in software, we need to program functionality for the widget. This is done with through a few steps:

i. Making the Type Recognizable:

First we need to add the type to the file types.txt. For example, if we were to make an LCD screen we would add the word “lcd” to types.txt.

ii. Recognizing the Physical Widget:

Next we need to make sure the needed information is stored in the Widget.txt file. The address can be found by running the addressGetter.tini program on the tini. Our LCD widget might be called lcd1 and have the address 48000000021F253B.

iii. Writing the Java Class:

Finally, you must write a Java class, which implements the Widget interface. This will define how the widget acts under given DaNick instructions. (See example LCD in the provided flash drive under /Dan Mendelsohn/code/make/LCD.java)

2. Interpreter development:

The interpreter that we use is currently built and called Par.java or Par.tini. If however, you want to modify this to add more functionality it is a matter of dealing with both Javacc and Java. The Lexor and Parser are built in Javacc. This means that the program defines how to Tokenize DaNick code, and defines (using a BNF) how the grammatical rules apply.

< program> ::= (<statement>)+ <DONE>

<statement> ::= (<control> | <modeChange> | <objectdec> | <objectinit> | <specify>) <SEMICOLON> | <modedec> | <modedef>

<control> ::= <objectID> <CONTROLS> <objectID> <ON> <command> <BY> <change>

<modeChange> ::= <objectID> <CHTO> <modeID> <ON> <command> <BY> <change>

<command> ::= <HOLD> | <CLICK> | <STCH>

<state> ::= <NUM>

<change> ::= <LINEAR>| <INC> | <DEC> | <TOGGLE>

<objectdeclaration> ::= <objectID> <objectID>

<objectinit> ::= <object> <IS> <typeID> <OPAREN> [<ID> <COMMA>]<NUM> <COMMA> <NUM> <CPAREN>

<modedec> ::= <MODE> <modeID> <SEMI>

<modedef> ::= <MODEDEF> <modeID> <OBRAC> (<statement>)* <CBRAC>

<specify> ::= <objectID> <ALIAS> <widgetID>

<objectID> ::= <ID>

<typeID> ::= <ID>

<typeID> ::= <ID>

<widgetID>::=<ID>

<modeID>::=<ID>

Reserved Words:

done, on, by, changes_to, controls, hold, click, statechange, state, linear, increment, decrement, toggle, has, states, mode, modedef, alias, is, numeric, alpha

Figure 16: Full BNF defining DaNick

Once the lexor and parser are working, it is simply a matter of adding Java code within the java brace section under each rules' definition. This causes the interpreter to execute the given code when that command is identified. Note: when parsing, javacc uses lookahead. This means that if two or more rules start with the same token or tokens, it will deal with the first

unless otherwise specified. For example “control” and “modechange” both start with an objected. This means that we need to lookahead by 2 to identify which of the two it is.

Once the interpreter has been created we can use a file called makescript. In order to do this, you need to put makescript and the interpreter in the same directory. Using the terminal you must first change directory to whatever holds the two. Then you can type “./makescript interpreterName” note that there is no extension on the interpreterName. The makescript file will compile the interpreter into a .class file, then convert this to a .tini format.

3. Maintaining the Development System:

In order for the makescript to work properly, there must be proper classpath definition. Currently the classpath points exactly where it needs to, but if the directory locations change, we need to make sure of three things. First the classpath must contain the path to the makescript directory, which should contain the interpreter and any files that it relies on (Widget interface and each widget’s class. This is currently stored in /home/mendelsd/Desktop

/SeniorProject/make). Next it must have a path to the java folder, which contains general java libraries (currently this can be found at: /home/mendelsd/Desktop/theDamnJava/

j2sdk1.4.2_19/jre/lib/rt.jar). Finally it needs to link to wherever you have the one-wire libraries stored (these can be found at: /home/mendelsd/Desktop/SeniorProject/TINISDK/tini1.17/bin/ newAllTINI.jar, and /home/mendelsd/Desktop/SeniorProject/TINISDK/tini1.17/bin/tini.jar). Note that if there are duplicates within these areas, the path may get stuck in one of the

sections and not look in the other directories. As long as all three of these are linked properly, makescript should work.

4. A Note on java Versions:

Using the TINI causes a problem when it comes to java. Unfortunately the TINI can only run up to java 1.4.2. There does not appear to be any effort to update this soon, so it can really be a nuisance. In order to prepare for this I suggest writing code in eclipse, and telling it to run using java 1.4.2.

5. Components:

The one-wire chips that we have been using are DS2450S+ and DS2413P+. These are developed by Maxim-ic. You can find information on these at: http://www.maxim-ic.com/datasheet/index.mvp/id/2921/t/al and http://www.maxim-ic.com/datasheet/index.mvp/id/4588/t/al respectively. The solder paste we use can be found by its product numbers: KE1512-ND (paste syringe), 10LL4-ND (plunger), and KDS22TN25-ND (tips). This must be refrigerated. You can find all three parts of the solder paste, as well as the one-wire chips at Digikey. Note: Currently we are not sure if the new solder paste we received works.

X. References:

1. Adler, Winiograd, 1992, *Usability: Turning technologies Into Tools*, Oxford University Press

2. Landauer, 1995, *The trouble with Computers*, MIT press

3. Potvin, Nicholas, “A Pedagogical Tool for Usability Science” March 19, 2009

4. R. Ramili, A. Jaafar, e-RUE : A Cheap Possible Solution for Usability Evaluation, Universiti Kebangsaan Malaysia, IEEE *International symposium on information technology*, 2008

5. Usability, <http://en.wikipedia.org/wiki/Usability#Defining_usability>

6. Usability Professionals association, http://www.usabilityprofessionals.org/