Product Title

Senior Design Final Documentation

The Software Engineering Adventure Line

Jonathan Tomes Erik Hattervig Andrew Koc

February 19, 2014

Contents

List of Figures

List of Tables

List of Algorithms

Mission

Mission statement inserted here.

Document Preparation and Updates

Current Version [1.0.0]

Prepared By:
Erik Hattervig
Andrew Koc
Jonathan Tomes

Revision History

Date	***Author***	***Version***	***Comments***
2/2/12	*Team Member #3*	*1.0.0*	*Initial version*
3/4/12	*Team Member #3*	*1.1.0*	*Edited version*

1

Overview and concept of operations

The overview should take the form of an executive summary. Give the reader a feel for the purpose of the document, what is contained in the document, and an idea of the purpose for the system or product.

1.1 Scope

What scope does this document cover?

1.2 Purpose

What is the purpose of the system or product?

1.2.1 Major System Component #1

Describe briefly the role this major component plays in this system.

1.2.2 Major System Component #2

Describe briefly the role this major component plays in this system.

1.2.3 Major System Component #3

Describe briefly the role this major component plays in this system.

1.3 Systems Goals

Briefly describe the overall goals this system plans to achieve. These goals are typically provided by the stakeholders. This is not intended to be a detailed requirements listing. Keep in mind that this section is still part of the Overview.

1.4 System Overview and Diagram

Provide a more detailed description of the major system components without getting too detailed. This section should contain a high-level block and/or flow diagram of the system highlighting the major components. See Figure 1.1. This is a floating figure environment. LaTeX will try to put it close to where it was typeset but will not allow the figure to be split if moving it can not happen. Figures, tables, algorithms and many other floating environments are automatically numbered and placed in the appropriate type of table of contents. You can move these and the numbers will update correctly.

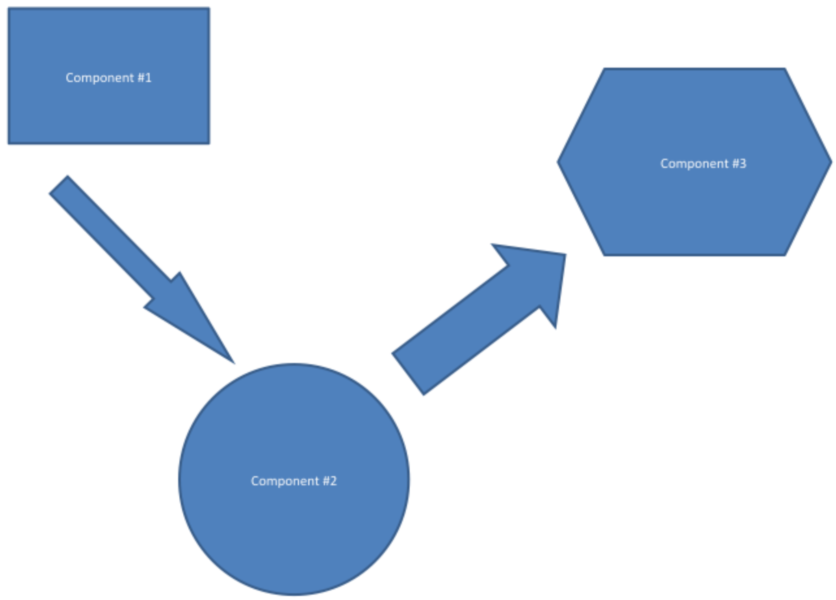

Figure 1.1: A sample figure .... System Diagram

1.5 Technologies Overview

This section should contain a list of specific technologies used to develop the system. The list should contain the name of the technology, brief description, link to reference material for further understanding, and briefly how/where/why it was used in the system. See Table 1.1. This is a floating table environment. LaTeX will try to put it close to where it was typeset but will not allow the table to be split.

7C0	hexadecimal
3700	octal
11111000000	binary
1984	decimal

Table 1.1: A sample Table ... some numbers.

2

Project Overview

This section provides some housekeeping type of information with regard to the team, project, etc.

2.1 Team Members and Roles

The team consists of Erik Hattervig, Andrew Koc, and Jonathon Tomes. Erik Hattervig is the Product Owner, Andrew Koc is the Technical Lead, and Jonathon Tomes is the Scrum Master. Erik is responsible for understanding the overall expectations of the product and communication with the customer about specific details regarding the operation and design of the product. Andrew is responsible for designing the technical aspects of the code. Jonathon is responsible for managing meetings and communication between team members and making sure the project is on schedule.

2.2 Project Management Approach

The sprint length for this project was 2 weeks. We began with a meeting to decide the user needs and split the program accordingly. Each of us would code different parts of the program and then we would all test and re-code as needed.

The code was stored, backed up, and shared through git hub. The back log and ownership was tracked through Trello. The user stories were condensed and placed on Trello to help design break points to split up the program between team members.

2.3 Phase Overview

The first phase of this Testing program was just to begin working on the program. The main purpose was to get to receive a root directory, find a .cpp file in the root. It would then write a log file that starting with a time stamp to be used later to record the results of tests.

After that it would a crawl through the sub directories recursively starting at the root, looking for .tst files that would be test cases for the program. Along with these would be .ans files that would allow us to compare the program output and see wich test cases failed.

It would then out put the results of each test to a log file. With a final log write that writes the percentage of passed and failed tests.

2.4 Terminology and Acronyms

none.

3

User Stories, Backlog and Requirements

3.1 Overview

The overview should take the form of an executive summary. Give the reader a feel for the purpose of the document, what is contained in the document, and an idea of the purpose for the system or product.

The userstories are provided by the stakeholders. You will create he backlogs and the requirements, and document here. This chapter should contain details about each of the requirements and how the requirements are or will be satisfied in the design and implementation of the system.

Below: list, describe, and define the requirements in this chapter. There could be any number of subsections to help provide the necessary level of detail.

3.1.1 Scope

What scope does this document cover? This document would contain stakeholder information, initial user stories, requirements, proof of concept results, and various research task results.

3.1.2 Purpose of the System

What is the purpose of the system or product?

3.2 Stakeholder Information

This section would provide the basic description of all of the stakeholders for the project. Who has an interest in the successful and/or unsuccessful completion of this project?

3.2.1 Customer or End User (Product Owner)

Who? What role will they play in the project? Will this person or group manage and prioritize the product backlog? Who will they interact with on the team to drive product backlog priorities if not done directly?

3.2.2 Management or Instructor (Scrum Master)

Who? What role will they play in the project? Will the Scrum Master drive the Sprint Meetings?

3.2.3 Investors

Are there any? Who? What role will they play?

3.2.4 Developers –Testers

Who? Is there a defined project manager, developer, tester, designer, architect, etc.?

3.3 Business Need

Use this section to define what business need exist and how this software will meet and/or exceed that business need.

3.4 Requirements and Design Constraints

Use this section to discuss what requirements exist that deal with meeting the business need. These requirements might equate to design constraints which can take the form of system, network, and/or user constraints. Examples: Windows Server only, iOS only, slow network constraints, or no offline, local storage capabilities.

3.4.1 System Requirements

What are they? How will they impact the potential design? Are there alternatives?

3.4.2 Network Requirements

What are they?

3.4.3 Development Environment Requirements

What are they? Is the system supposed to be cross-platform?

3.4.4 Project Management Methodology

The stakeholders might restrict how the project implementation will be managed. There may be constraints on when design meetings will take place. There might be restrictions on how often progress reports need to be provided and to whom.

- What system will be used to keep track of the backlogs and sprint status?
- Will all parties have access to the Sprint and Product Backlogs?
- How many Sprints will encompass this particular project?
- How long are the Sprint Cycles?
- Are there restrictions on source control?

3.5 User Stories

This section can really be seen as the guts of the document. This section should be the result of discussions with the stakeholders with regard to the actual functional requirements of the software. It is the user stories that will be used in the work breakdown structure to build tasks to fill the product backlog for implementation through the sprints.

This section should contain sub-sections to define and potentially provide a breakdown of larger user stories into smaller user stories.

3.5.1 User Story #1

User story #1 discussed.

3.5.1.a User Story #1 Breakdown

Does the first user story need some division into smaller, consumable parts by the reader? This does not need to go to the level of actual task definition and may not be required.

3.5.2 User Story #2

3.5.2.a User Story #2 Breakdown

User story #2 ....

3.5.3 User Story #3

3.5.3.a User Story #3 Breakdown

User story #3 ....

3.6 Research or Proof of Concept Results

This section is reserved for the discussion centered on any research that needed to take place before full system design. The research efforts may have led to the need to actually provide a proof of concept for approval by the stakeholders. The proof of concept might even go to the extent of a user interface design or mockups.

3.7 Supporting Material

This document might contain references or supporting material which should be documented and discussed either here if approprite or more often in the appendices at the end. This material may have been provided by the stakeholders or it may be material garnered from research tasks.

4

Design and Implementation

The design of this project is a relativly simple one. In order to test a program there are four steps we need to complete. The first we need to compile the file to be tested. Second we need to locate all the test cases for it. Then after finding the test case we need to run the program with those test cases, and finally we need to evaluate and log the results each test case.

Algorithm 1 Overall Algorithm

Locate .cpp file
Compile .cpp file
while Locate .tst file **do**
  Run .tst file
  if .tst File passes **then**
    $C \Leftarrow C + 1$
  end if
  $T \Leftarrow T + 1$
end while
$PercentagePassed \Leftarrow$C / T

4.1 Locate and Compile .cpp File

4.1.1 Technologies Used

This needed to use the dirent.h library inorder to find the .cpp file, this will be covered further in Finding the test cases, but in general this library allows us to search for files. Then to actually compile the code it uses a system command to invoke g++ and compile it.

4.1.2 Data Flow Diagram

It is important to build and maintain a data flow diagram. However, it may be that a component is best described visually with an architecture diagram.

4.1.3 Design Details

Here is the code for locating the .cpp file, where root is the full path to the directory the .cpp file is located in:

```
DIR* dir;
struct dirent* file;
std::string filename;
bool foundFlag = false;
```

```
    std::string cppFile;

    dir = opendir( root.c_str() );
    while( ( file = readdir(dir) ) != NULL && !foundFlag )
      {
        //Get the file name
        filename = file -> d_name;
        //skip over "." and ".."
        if( filename != "." && filename != ".." )
          {
        if(filename.find( ".cpp" ) != std::string::npos )
          {
            cppFile = filename;
            foundFlag = true;
          }
          }
      }


    if( !foundFlag )
      {
        std::cout << "Could not find a cpp file." << std::endl;
   std::cout << "Ending Program" << std::endl;
        return 0;
      }
```

The following is the code to compile the .cpp file, where root is the directory where the .cpp file is located and progName is the name of the .cpp file:

```
void Compil(  std::string root, std::string progName )
{
  //Create the argument to send to g++
  std::string progPath = "";
  progPath += root;
  progPath += "/";
  progPath += progName;
  //creat the system command and execute it.
  std::string command;
  command = "g++ " + progPath;
  system( command.c_str() );

  return;
}
```

4.2 Locate the Test Cases

4.2.1 Technologies Used

This step of the program relies heavily on the dirent.h library. It uses a recursive function to search the directory and when it finds a subdirectory it calls itself to search that subdirectory.

4.2.2 Component Overview

Two data types from the dirent.h library allow us to search and retrive files from directories. The first DIR* allows us to open a specific directory and red from it. The second struct dirent* is the data type we use to retrive the files we read from a directory. It has two elements that can be used to categorize the file, d_name and d_type. d_name is the name of the file, we can use this to search for certain file extensions, and d_type is a code for the type of file it is. This is important in order to find which files are subdirectories.

4.2.3 Data Flow Diagram

It is important to build and maintain a data flow diagram. However, it may be that a component is best described visually with an architecture diagram.

4.2.4 Design Details

The following is the function to locate .tst files given a root directory:

```
void DirCrawl( std::string rootDir , std::ofstream &logFile , std::string exec , int &passe
{
   DIR* dir = opendir( rootDir.c_str() ); // Open the directory
   struct dirent* file; // File entry structure from dirent.h
   std::string filename;   //used in finding if a file has the extention .tst

   // Read each file one at a time
   // Readdir returns next file in the directory, returns null if no other files exist
   while( ( file = readdir(dir)) != NULL )
   {
      //place file name into string filename for easier checking
      filename = file->d_name;

      // skip over the directories "." and ".."
      if ( filename != "." && filename != ".." )
      {
         // checks if the file is a subdirectory, 4 is the integer idetifyer
         // for the dirent struct on Lixux systems
         if ( (int)file->d_type == 4 )
         {
            //moves into the sub-directory
                DirCrawl( rootDir + filename + "/" , logFile , exec , passed , tested );
         }
         else
         {
            // checks if the file has a .tst in it. string find returns
            // string::npos if the substring cannot be found
            if ( filename.find( ".tst") != std::string::npos )
            {
               // pass the file onto the grader
                    if (run_test_case( rootDir + '/' + filename , exec , logFile ) )
               {
                  passed += 1;
               }
               tested += 1;

            }
         }
      }
   }

   closedir(dir);
```

```
    return;
}
```

The other parameters are passed along to functions it calls, in the case of logFile and exec and passed and tested keep track of the number of tests and the number of those that actually passed.

4.3 Running a Test Case

4.3.1 Technologies Used

This part of the program is done through system calls to the linux terminal.

4.3.2 Component Overview

First the program to be tested is run piping the input from the .tst file and the output to a similarly named .out file. Then using the diff command in linux the .out file, the output of the program, is compared to the .ans file, what should have been the output of the file. The console output is piped into a junk file which is later deleted. If the diff command returns zero then the files contain the same thing.

4.3.3 Data Flow Diagram

It is important to build and maintain a data flow diagram. However, it may be that a component is best described visually with an architecture diagram.

4.3.4 Design Details

Here is the function to test an executable with a given test file, it gets passed the logfile to pass it to the function that writes to the log file. It returns true if the program passed the test case and false if it did not.

```
bool run_test_case( std::string test_file, std::string exec,
                    std::ofstream &log_file )
{
    std::string out_file = test_file;
    std::string ans_file = test_file;
    std::string test_num = "";
    std::string command_string = "";
    int i;
    int result;

    //get test number
    //name for the test file will be "*case###.tst" so the last number is at
    //position length - 5
    for( i = test_file.length() - 5; test_file[i] >= '0' && test_file[i] <= '9'; i-- )
        //since we are reading in backward the new number gets added at the front
        test_num = test_file[i] + test_num;

    //get text for .out file and .ans file
    //remove tst
    out_file.resize(out_file.size() - 3);
    //add out so we have case#.out
    out_file += "out";

    //remove tst
    ans_file.resize(out_file.size() - 3);
    //add ans so we have case#.ans
    ans_file += "ans";
```

```
    //command string = "executable < case.tst > case.out"
    //run the program with input from .tst and pipe output to .out
    command_string = exec + " < " + test_file + " > " + out_file;
    //execute the program
    std::system(command_string.c_str());

    //compare the programs output and the expected output( .out and .ans )
    // diff --ignore-all-space case.out case.ans > nul
    //if it == 0 the files were the same
    // the --ignore ignores whitespace on each line, so trailing spaces
    // or newlines aren't flagged as incorrect
    // > pipes the output into a file called nul
    command_string = "diff --ignore-all-space " + out_file + " " + ans_file + " > nul";
    result = std::system(command_string.c_str());

    //passed test
    if ( result == 0 )
    {
        LogWrite(log_file, test_num,"passed");
        return true;
    }
    //failed test
    else
    {
        LogWrite(log_file, test_num,"failed");
        return false;
    }
}
```

4.4 Logging the Results

4.4.1 Technologies Used

The fstream library is used so that we can write to a file.

4.4.2 Component Overview

There are two functions that fall into this, the first is a function that writes out the result of a single test case and the second writes the final results of the testing. The ofstream is opened in main to append data so that previous runs on the same .cpp file are saved.

4.4.3 Data Flow Diagram

I'm going to build some images for this later.

4.4.4 Design Details

This first function writes the results of a single test case, it is called by the testing function after it knows determines if the case passed.

```
void LogWrite( std::ofstream & fout, std::string testNumber, std::string result )
{
  fout << testNumber << ": " << result.c_str() << std::endl;
  return;
```

```
}
```

The second function is called after the directory crawl returns to the main function and writes out the final result of the tests.

```
void FinalLogWrite( std::ofstream & fout, int numPassed, int numTest )
{
  //Calculate the number of tests failed.
  int numFailed;
  numFailed = numTest - numPassed;
  //Calculate the percent passed.
  float perPassed;
  perPassed = (float) numPassed/numTest;
  perPassed =  (int)(perPassed * 100);
  //Calculate the percent failed.
  float perFailed;
  perFailed = (float) numFailed/numTest;
  perFailed = (int)(perFailed * 100);

  //Write to stream.
  fout << "Percent of tests Passed: " << perPassed <<  "%" << std::endl;
  fout << "Percent of tests failed: " << perFailed << "%" << std::endl;
  return;
}
```

5

System and Unit Testing

The approach taken to testing was a very simple process due to the simplistic nature of the program itself, as such not much detail will be provided.

5.1 Overview

Before each part is pushed up to the repository the programmer would test it and ensure that it was functioning the way they intended it to function. After each individual part was pushed up a main function was written to do the overall algorithm, little tweaks were made to each function so that they would work together properly, but no large changes to the algorithm.

5.2 Dependencies

All of the tests depend on the g++ compiler, and linux platform. The code needs to compile a C++ file inorder to run and test that file.

5.3 Test Setup and Execution

Test cases were given to us by Dr. Logar. Each one was tested and then we went through and manually looked at the .ans and .out files to make sure that the result in the .log file matched with what we would actually determine the answer to be by looking ourselves.

6

Development Environment

Since the program was to test files on a Linux enviornment, it was developed on a Linux enviornment.

6.1 Development IDE and Tools

No special Tools or IDE were used to develop this. Since the code was so simple each programmer just used what ever coding enviornment they wanted. All the code was tested on a Linux machine using the g++ compiler. The debug tool gdb was used to debug the code when necessary.

6.2 Source Control

We used github for source control. The repository can be found at this url:
https://github.com/TheSoftwareEngineeringAdventureLine/ProgramTesterStage1.git

6.3 Dependencies

This program is dependent on the C++ Standard Library as well as the g++ compiler on a Linux system.

6.4 Build Environment

The execuatble is built by the g++ compiler. You can either compile it manually, using the command:

```
g++ -o tester ProgramTester.cpp
```

Or by using the following make file:

```
# compiler
CC = g++

# compiler options
CFLAGS = -c -Wall

all: tester

tester: tester.o
   $(CC) -lm tester.o -o ProgramTester

tester.o: ProgramTester.cpp
   $(CC) $(CFLAGS) ProgramTester.cpp
```

```
clean:
   rm -rf *o tester
```

7

Release – Setup – Deployment

This section should contain any specific subsection regarding specifics in releasing, setup, and/or deployment of the system.

7.1 Deployment Information and Dependencies

Are there dependencies that are not embedded into the system install?

7.2 Setup Information

How is a setup/install built?

7.3 System Versioning Information

How is the system versioned?

8

User Documentation

This section should contain the basis for any end user documentation for the system. End user documentation would cover the basic steps for setup and use of the system. It is likely that the majority of this section would be present in its own document to be delivered to the end user. However, it is recommended the original is contained and maintained in this document.

8.1 User Guide

The source for the user guide can go here. You have some options for how to handle the user docs. If you have some `newpage` commands around the guide then you can just print out those pages. If a different formatting is required, then have the source in a separate file `userguide.tex` and include that file here. That file can also be included into a driver (like the senior design template) which has the client specified formatting. Again, this is a single source approach.

8.2 Installation Guide

8.3 Programmer Manual

9

Class Index

9.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

10

Class Documentation

10.1 Poly Class Reference

Public Member Functions

- Poly ()
- ~Poly ()
- int myfunction (int)

10.1.1 Constructor & Destructor Documentation

10.1.1.a Poly::Poly ( )

My constructor

10.1.1.b Poly::~Poly ( )

My destructor

10.1.2 Member Function Documentation

10.1.2.a int Poly::myfunction (int *a*)

my own example function fancy new function

new variable

The documentation for this class was generated from the following file:

- hello.cpp

Acknowledgement

Thanks

Supporting Materials

This document will contain several appendices used as a way to separate out major component details, logic details, or tables of information. Use of this structure will help keep the document clean, readable, and organized.

Sprint Reports

10.1 Sprint Report #1

10.2 Sprint Report #2

10.3 Sprint Report #3

Industrial Experience

10.4 Resumes

10.5 Industrial Experience Reports

10.5.1 Name1

10.5.2 Name2

10.5.3 Name3

Appendix

Latex sample file:

10.1 Introduction

This is a sample input file. Comparing it with the output it generates can show you how to produce a simple document of your own.

10.2 Ordinary Text

The ends of words and sentences are marked by spaces. It doesn't matter how many spaces you type; one is as good as 100. The end of a line counts as a space.

One or more blank lines denote the end of a paragraph.

Since any number of consecutive spaces are treated like a single one, the formatting of the input file makes no difference to TEX, but it makes a difference to you. When you use LATEX, making your input file as easy to read as possible will be a great help as you write your document and when you change it. This sample file shows how you can add comments to your own input file.

Because printing is different from typewriting, there are a number of things that you have to do differently when preparing an input file than if you were just typing the document directly. Quotation marks like "this" have to be handled specially, as do quotes within quotes: "'this' is what I just wrote, not 'that'".

Dashes come in three sizes: an intra-word dash, a medium dash for number ranges like 1–2, and a punctuation dash—like this.

A sentence-ending space should be larger than the space between words within a sentence. You sometimes have to type special commands in conjunction with punctuation characters to get this right, as in the following sentence. Gnats, gnus, etc. all begin with G. You should check the spaces after periods when reading your output to make sure you haven't forgotten any special cases. Generating an ellipsis . . . with the right spacing around the periods requires a special command.

TEX interprets some common characters as commands, so you must type special commands to generate them. These characters include the following: $ & % # { and }.

In printing, text is emphasized by using an *italic* type style.

A long segment of text can also be emphasized in this way. Text within such a segment given additional emphasis with Roman *type. Italic type loses its ability to emphasize and become simply distracting when used excessively.*

It is sometimes necessary to prevent TEX from breaking a line where it might otherwise do so. This may be at a space, as between the "Mr." and "Jones" in "Mr. Jones", or within a word—especially when the word is a symbol like *itemnum* that makes little sense when hyphenated across lines.

Footnotes[1] pose no problem.

TEX is good at typesetting mathematical formulas like $x - 3y = 7$ or $a_1 > x^{2n}/y^{2n} > x'$. Remember that a letter like x is a formula when it denotes a mathematical symbol, and should be treated as one.

[1] This is an example of a footnote.

10.3 Displayed Text

Text is displayed by indenting it from the left margin. Quotations are commonly displayed. There are short quotations

> This is a short a quotation. It consists of a single paragraph of text. There is no paragraph indentation.

and longer ones.

> This is a longer quotation. It consists of two paragraphs of text. The beginning of each paragraph is indicated by an extra indentation.
>
> This is the second paragarph of the quotation. It is just as dull as the first paragraph.

Another frequently-displayed structure is a list. The following is an example of an *itemized* list.

- This is the first item of an itemized list. Each item in the list is marked with a "tick". The document style determines what kind of tick mark is used.
- This is the second item of the list. It contains another list nested inside it. The inner list is an *enumerated* list.

  1. This is the first item of an enumerated list that is nested within the itemized list.
  2. This is the second item of the inner list. LaTeX allows you to nest lists deeper than you really should.

  This is the rest of the second item of the outer list. It is no more interesting than any other part of the item.
- This is the third item of the list.

You can even display poetry.

> There is an environment for verse
> Whose features some poets will curse.
>
> For instead of making
> Them do *all* line breaking,
> It allows them to put too many words on a line when they'd rather be forced to be terse.

Mathematical formulas may also be displayed. A displayed formula is one-line long; multiline formulas require special formatting instructions.

$$x' + y^2 = z_i^2$$

Don't start a paragraph with a displayed equation, nor make one a paragraph by itself.

10.4 Build process

To build LaTeX documents you need the latex program. It is free and available on all operating systems. Download and install. Many of us use the TexLive distribution and are very happy with it. You can use a editor and command line or use an IDE. To build this document via command line:

```
alta>  pdflatex SystemTemplate
```

If you change the bib entries, then you need to update the bib files:

```
alta>  pdflatex SystemTemplate
alta>  bibtex SystemTemplate
alta>  pdflatex SystemTemplate
alta>  pdflatex SystemTemplate
```

Acknowledgement

Thanks to Leslie Lamport

Bibliography

[1] R. Arkin. *Governing Lethal Behavior in Autonomous Robots*. Taylor & Francis, 2009.

[2] Howie Choset, Kevin M. Lynch, Seth Hutchinson, George A Kantor, Wolfram Burgard, Lydia E. Kavraki, and Sebastian Thrun. *Principles of Robot Motion: Theory, Algorithms, and Implementations*. MIT Press, Cambridge, MA, June 2005.

[3] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[4] V. Lumelsky and A. Stepanov. Path planning strategies for point mobile automation moving amidst unknown obstacles of arbirary shape. *Algorithmica*, pages 403–430, 1987.

[5] S.A. NOLFI and D.A. FLOREANO. *Evolutionary Robotics: The Biology, Intelligence, and Technology of Self-Organizing Machines*. A Bradford book. A BRADFORD BOOK/THE MIT PRESS, 2000.

[6] Wikipedia. Asimo — Wikipedia, the free encyclopedia. `http://upload.wikimedia.org/wikipedia/commons/thumb/0/05/HONDA_ASIMO.jpg/450px-HONDA_ASIMO.jpg`, 2013. [Online; accessed June 23, 2013].