Explorer Post 1010
Botball Team 16-0160
Greater DC Region

Period 2 Software Design

Introduction

- The segment of code reviewed below describes the omni chassis turns and straight line movement. In short the navigation functions and associated global variables.
- This segment was selected because of the challenge the omni chassis and omni chassis wheels in specific provided. Since they were new parts and in a style no one had ever worked with they challenged the programers to learn to accurately steer them.
- This code was reviewed Wednesday, March 23rd, the review being written Saturday, March 26th.

Best Practices Checklist

- Code does indeed use functions to organize code so that different segments can both be customized and reused.
- Code does include comments to facilitate different programmers working on the same piece of code; especially when another programmer is working on a different piece of code. This also helps builders to understand whats going on.
  - By including comments specifying what a specific function does the function can be more easily reused because its purpose is always clear. (yes this is present)
  - By labeling the functions arguments and return values the programmer always knows how to use a specific function should she/he desire to employ it (also present)
- All variable names are descriptive and explain their purpose in the function so that they can be reused throughout the program without too much effort. Thus the code is more efficient and practical to write.
- The code makes good use of named constants so that important values can be repeated in a logical manner (and so the programmers to not have to go back and recheck certain values).
- Code is appropriately formatted not only to show flow of control (which in combination with proper comments make code easy to read) but to help programmers identify errors.
- Any blocks of code that are not in use will have been deleted during code revision (occurring constantly at random intervals). Without large and confusing useless blocks of code the programmers can better understand the bigger picture.
- The issues that the code does have are making sure that the code does not rely on large hardcoded sections. The team is making an effort to use large blocks of code, created for a specific task and creating with it a more useful function that can accommodate multiple circumstances by passing to it only a value that is specific to a certain situation.

Code Analysis Section

Code Analysis Section - Reliability

- Most error detection became apparent through attempting to run the robot; this was doubly useful because it gave the builders insight into how the built appendages perform. This is because the two tasks are very closely aligned; if either the effector or code does not perform it becomes very apparent shortly into testing and not only does it become apparent but it is clear with piece (i.e. effector or code) is not working.
  - When errors are detected through this message the team employs error statements. By printing text to the screen when certain conditions are not called the programmers can immediately notice the location of an error. These error statements are also numerated (with the corresponding location in the code commented accordingly) so that the precise location where something went wrong is ridiculously easy to find.
- This code is extremely reliable because of its use of functions. Functions are obviously incredibly important in any situation but movement, which can vary a lot but often only changes only by changing a value. In this case the value changed would be a parameter and the parameter to be sent into the function is saved as a global variable. Thus what was originally a long and arduous process that required constantly rechecking to find which different variables needed to be hard coded is reduced to a nice and simple function that is extremely multipurpose.
- Since the function is multipurpose (as a result of having a variety of parameters that allow it to suit different situations) the functions themselves are actually very simple. With a simpler function it is harder for the code to mess up; there are no hidden conditions that the code could trip up on (unlike when there are useless blocks of code in a program as then they can conceal small details that run the risk of ruining the robots performance).
- One of the few issues that the robot had was that the robot movement and the added challenge of climbing a crater rim called for more conditions than a simple switch could account for. Thus if the robot did not know how to perform in every situation its reliability is greatly reduced. This issue was resolved by increasing the number of parameters that the function takes in because instead of just having the switch now the more random variables are taken care of.

Code Analysis Section - Maintainability

- The reviewed code (below) is easy to understand, modify, and reuse because of how it makes use of all good coding practices. By having all programmers on the team make good use of a specific set of formatting guidelines when other people work on shared coding tasks they can easily understand the flow of the code. Another technique employed to improve communication between programmers is that people add comments that explain who was working on each specific part of the code and what the work was done. That way of someone does not understand they know exactly who to ask (of course extensive comments heading and throughout help too). In short, communication between team members and extensive

comments (as well as overall correct layouts) allows for all coders to understand the purpose of the code.

- The code is easy to modify because it makes use of a simple set of functions that are extremely multipurpose. By simply changing the values inputted into the function the entire code can easily be modified to suit the builders ever-changing needs. This usage of functions also makes the code very reusable. These functions themselves can be very complex as they can occasionally make use of a lot of mathematics to achieve various goals. To remedy this problem the pseudo code often written for such functions is left in as a comment as a header for the program.
  - This header also has another purpose; it labels the return types and parameters the function uses. This comment is also copied and found at the very top of the program in a list. This easily tells whoever is modifying the doc not only exactly what they have available to use but also how to use it.

-The maintainability of the code could be improved by making sure that all extra comments are periodically removed. While it is extremely useful at fist to know exactly who had done what, after a while the older changes become obsolete and make the code more confusing to work with because they can get confused with newer changes. While the comments serve as communication for the code, in some cases throughout the code they have been overused. This issue will be resolved by using having programmers remember to delete old comments after a few weeks, ensuring that the code maintains an organized state.

Code Analysis Section - Effectiveness

- The reviewed code does accurately and correctly perform its assigned task… And then takes it one step further. By offering a switch and multiple statements the originally difficult movement of the omni chassis is consolidated into one function. Thus this function not only effectively solves the issue of omni chassis movement but it removes the possibility of there being any issues in the future.
- This code correctly performs its given task by performing *any* given task of a similar nature. As the given task was to get the omni chassis to move, a function that supports any and all omni chassis movement is as correct as it gets. Furthermore the code is effective and efficient, so the risk of there being errors based on a small and hidden condition, lost under useless statements is greatly reduced.
- This code is additionally effective by assigning at the beginning global variables that remove the need for constantly hardcoding in various numbers. By having variables that not only reduce error the code can be more effectively used because they are labeled with exactly the movement they are required to perform.
- The code (seen below in the “before” section) had various notable issues, the most important of which being that since it was a new coding concept and not every programmer fully understood exactly how to use it, the direct usage of the omni chassis movement found throughout the program was confusing. There was also various pieces of unused code found throughout with only increased confusion. Topping all that off with too many comments, most of which only pertaining to whatever the programer was planning and not what the code

actually needed to perform, the old code, while mostly capable of performing the given tasks was miserable to work with and needed revision and improvement.

Before Code

```
 drive(0,90,800,900); //90 degree turn right (CW)
        drive(0,90,800,900); //turn CW so arm faces botguy
        //mavomni(2, 900, 2400); //turn CW so arm faces botguy
        armout(); //position arm
        grab(); //grab botguy
/*
Talk about adding segment for claw usage… (to do)
*/
        printf("Botguy grabbed!\n");
        mavomni(1, 500, 700); //back up a little to turn
        break; //continue with code}
    }
/*
  /*Get onto the Crater Rim*/
    printf("Get to the Crater Rim!\n");
    drive(0,-90,800,900); //90 degree turn left (CCW)
/* Remember to label the global variable*/
     //mavomni(3,900,1200); //turn CCW towards ramp
     printf("Go up the ramp!!\n");
    while(digital(touch)==0){ //until touch sensor is hit
      //linefollow();
      mavomni(0, 600, 400); //drive up the ramp
      if(digital(touch)==1){ //when touch sensor is hit
        ao();
        mavomni(1, 500, 700); //back up a little to turn
        drive(0,90,800,900); //90 degree turn CW towards center
field
        //mavomni(2,400,200); //turn CW towards center field
*/
```

- Above is a section of code. As described above it is covered in comments and impossible to trace because it does not always follow proper if statement and loop formatting. Clearly it does not follow the "Best Practices Checklist", mostly because it is only one step above pseudo code. Still in the process, there are even remnants of planning all in the programmer's (an unidentifiable programmer because people have not added names and dates modified at this point) effort to map out their thoughts. The revised code below is light years better.

Improved Code

```
      //Created by Kat A. Last modified: 19 Mar 19:35
      #define left 0 //front left motor port 0
      #define right 1 //front right motor port 1
      #define back 3 //back motor port 3

      void mavomni(char type, int vel, int time)
```

```
{
      switch (type)
      {
            case 0: //driving forward
            printf("driving forward with mav %i for %i ms\n", vel,
time);
            mav(left,vel + 200);
            mav(right,-vel);
            msleep(time);
            ao();
             break;

            case 1:    //driving backward
            printf("driving forward with mav %i for %i ms\n", vel,
time);
            mav(left,-vel);
            mav(right,vel - 300);
            msleep(time);
            ao();
             break;

      }
```

- Obviously the code above is much better than its predecessor (most probably because the programmers had a moment to finalize their thoughts and work to improve its efficiency). Unlike the first excerpt this code is a function, allowing for the work that has already been done to shine through and the same purpose be reused. The included switch allows for a variety of situations to be solved, even if all the cases are similar. With each case being slightly similar, instead of having to continually research not only how to use omni chassis movement you also never have to worry about what slight change in values would be needed to get the code to work as planned.
  - The global variables also aid in this because they incur that you don't have to remember even what change needed to be made to get the code to work, simply typing in key phrases such as “right” and “left” would solve the problem.