CS2303 In Class Exercises

Jan 24, 2020

January 11, 2020

Abstract

Today's goal is that you can grasp in a top-down view, the process of producing a solution to a problem statement, such that the solution is expressed as a C executable. Also, see the second part of code that processes data from the command line.

Name and WPI email: ______________ Type text here

1 Overall Process Steps

The overall process is:

1. Analyze the requirements to find the components
2. Design the interactions between the components to cooperatively solve the problem, at a high level of abstraction
3. Convert the individual interactions into function invocations, identifying parameters and return values.
4. Design data structures to support the parameters and return values.
5. Think about all the functions together, try to place the functions in sequence, from those needed the least other functionality to those depending upon the most other functionality.
6. Consider whether any function would help you check out the program's behavior. You might wish to develop these earliest.
7. Take each function identified above in turn, and for it:
8. Create a stub and a test function.
9. Consider the success criteria for the function.
10. Create test cases.

11. Run the test on the stub and observe failure.

12. Develop the stub.

13. Run the test and look for success.

14. Once all of the components have been tested, create in the production code, the interaction depicted in the sequence diagram. (For more complicated programs you might wish to build intermediate sized aggregations of functions.)

Though in the (older) waterfall process, requirements analysis, design, implementation and test were separate phases that were carried out in sequence, there are more modern techniques. More modern techniques, such as test-driven development, can move back and forth between analysis, design and test-driven development activities.

2 Analyze Requirements

We have seen that we can extract nouns and noun phrases from requirements text. We can use these nouns and noun phrases as a point of departure for identifying components of the solution. The purpose and guidance for components is separation of concerns. We wish, at a later stage, to be able to focus our attention on one component at a time, separating out that one component and being able to be concerned about it while leaving the other components out of our detailed attention. We can begin a sequence diagram by including the component names in the diagram, and giving them (vertical)lifelines. We can think at a higher level of abstraction than code, which has been shown to increase developer productivity, as we consider interactions from one component to another. For example, storage and retrieval of state data can be separated from the user menus and other aspects of the graphical user interface appearance. High level interactions between components appear as horizontal lines in the sequence diagram. These are used later to infer functions, that are invoked by the component that sends the message, and implemented in the component that receives the message.

3 Design Interactions

These messages, interactions between components are invented in the process of solving the requirements statement. This invention becomes easier with practice.

Example:
Write a sequence diagram for buying lunch at the food court.

At the food court there are multiple components, such as the people receiving and assembling a dish, perhaps the dish is warmed, then the checkout person determines the charge and obtains the payment.

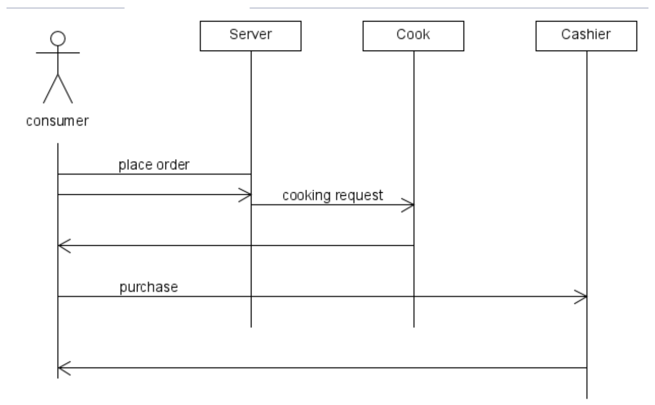

Figure 1: Familiar activities can be depicted in sequence diagrams.

3.1 Identify Parameters and Return Values

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Identify some parameters and return values for the food court sequence diagram.

   placeOrder(food, drink) -> order
   cookingRequest(food) -> waitTime
   purchase(paymentType, amount) -> change, food

3.2 Design Data Structures

Use typedef struct and typedef enum to design datatypes that support the components identified earlier, and the functions to be implemented in those components.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Write a typedef struct that holds the data for a parameterized cup of coffee or other item you purchase.

```
typedef stuct{
int sugar;
int cream;
}coffee;
```

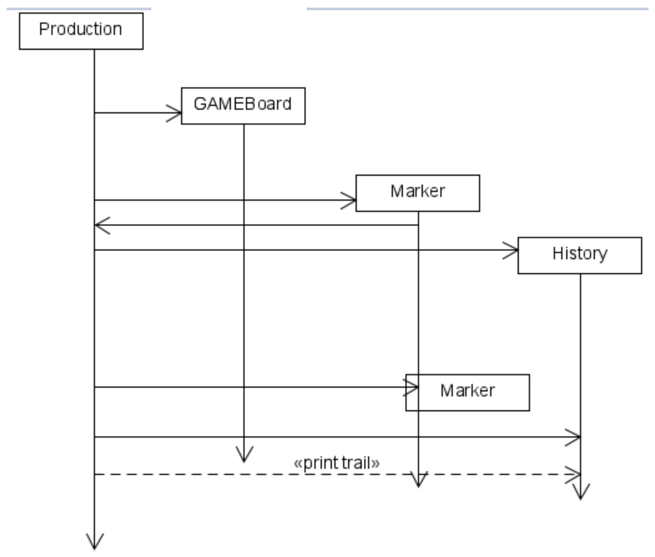

Figure 2: Some components and interactions for placing some markers, sequentially, on a gameboard, and printing successive board configurations.

4 Sequencing Functions for Development

Example:

The order in which functions are called, in execution of a program, is depicted in a sequence diagram. Choices are made about the order in which the several functions of a program are developed. The order in which functions are developed does not have to be the same as the order in which the functions are used in production. It can often be the case that a display or other output function, needed at the end of execution, provides useful insight during development, and for that reason might be developed early.

Exercise: (As always, if you have questions, ask. After this occasion, this material becomes background knowledge.)

1. Suppose you are developing a program to play checkers. The requirements only call for the final board configuration to be printed. Would you choose to develop the function that displays the board configuration later, or earlier, so that you could observe your application in intermediate states?

display earlier to be able to check for errors

5 Implementation

The presence of a message in a sequence diagram prompts us to infer a production function. For that production code, we expect to produce a function prototype, a function invocation and a function implementation. Moreover, the existence of the production function implies there will be a test function for it. That test function will have its three manifestations: prototype, invocation and implementation. Within the test function, the production code will be invoked, usually more than once, so that two different answers are elicited. Thus the test function can observe that the right answers are not received from the stub (which can only send one answer). During the creation of the test function and test cases, success criteria for the corresponding production function must be considered. Right answers must be known, to create the test function. Once this work has been done for the test function, it serves very well in extending the stub implementation into the full implementation.

6 Sequence Diagram Implementation

If, as is desirable, the sequence diagram is complete at a high level of abstraction, then once each of the production functions is developed in the test-driven manner as described above, the invocations sequenced in the diagram should be invoked in the production code as the sequence diagram shows.

7 Command Line Processing

Previously we saw how the main function receives the command line parameters, and passes them on as arguments to production.

```
int main(int argc, char* argv[])
{
    if(tests())
    {
        production(argc, argv);
    }
    else
    {
        puts("Tests did not pass.");
    }
    return EXIT_SUCCESS;
}
```

Next we will see how production works with these parameters.

```
bool production(int argc, char* argv[])
{
```

```
bool answer = false;

if(argc <=1) //no interesting information
{
    puts("Didn’t find any arguments.");fflush(stdout);
    answer = false;
}
else //there is interesting information
{
    printf("Found %d arguments.\n", argc);fflush(stdout);
    char filename[FILENAMELENGTHALLOWANCE];
    char* eptr=(char*) malloc(sizeof(char*));
    long aL=-1L;
    int maxRooms=100;
    int maxTreasure= 100;
    for(int i = 1; i<argc; i++) //don’t want to read argv[0]
    {//argv[i] is a string
          switch(i)
          {
              case 1:
                  //this is filename
                  printf("The length of the filename is %d.\n",strlen(argv[i]));
                  printf("The proposed filename is %s.\n", argv[i]);
                  if(strlen(argv[i])>=FILENAMELENGTHALLOWANCE)
                  {
                      puts("Filename is too long.");fflush(stdout);
                      answer = false;
                  }
                  else
                  {
                      strcpy(filename, argv[i]);
                      printf("Filename was %s.\n", filename);fflush(stdout);
                   }
                  break;
               case 2:
                     //this is maximum number of rooms
                     aL= strtol(argv[i], &eptr, 10);
                     maxRooms = (int) aL;
                     printf("Number of rooms is %d\n",maxRooms);fflush(stdout);
                    break;
                case 3:
                      //this is maximum amount of treasure
                      aL= strtol(argv[i], &eptr, 10);
                      maxTreasure = (int) aL;
                      printf("Amount of treasure is %d\n",maxTreasure);fflush(stdout);
                     break;
                default:
                      puts("Unexpected argument count."); fflush(stdout);
                      answer = false;
                     break;
              }//end of switch
        }//end of for loop on argument count
        puts("after reading arguments"); fflush(stdout);
```

The integer variable argc gives a count of the command line arguments. The zero-th command line argument is the name of the program that was called. If there are any arguments after that, the first one will be in argv[1], with any others following.

An executable has a "Usage", that is, there exists an understanding of which arguments are for which purpose. Arguments are matched to purpose by the order they are found in the sequence. In the example above, the first argument is expected to be a string of characters, that express a filename.

To protect against a common error, called buffer overflow, in which characters continue to be written beyond the space allowed for them, a check is made for how many characters are to be copied. This check is done using strlen. The space allowed for characters has been controlled using a defined constant, called FILENAMELENGTHALLOWANCE. It is good practice to use all capital letters for defined constants. If the character string is not too long, it is copied, otherwise a message is printed out. Other choices in programming avoid copying the characters; this opportunity was taken to show you a method of checking before copying a character string in case you ever need to do it.

Cases 2 and 3 process a string that represents an integer. The strtol function is used in preference to atoi, for error handling reasons. You should not use atoi. If you do use atoi on any graded material, you will lose points.