Topics

1. Naming
2. Extensibility
3. TicTacToe Testing
4. Commenting
A Word on Naming and Commenting

Use the course materials to guide you through assignments. These are my thoughts and perspective. Your assignments adhere to certain standards and conventions.

● You are probably concerned about:
  ○ How much time do these things take?
● What do you propose as an alternative?
  ○ Every alternative I have heard is just “cost shifting” and results in higher costs (time).
● “Why motivates how and what”
Naming

What about short names? Like single letters?

i, j

1. Does i refer to row or column? Is there a convention?
2. What about x, y?
3. We SAY row and column. Is that what we mean?
4. What if we mix i, j and x, y?

Do YOURSELF a favor. Make variable names meaningful and unambiguous.
Single Letter or Short Names?

Consider the scope.

Also, consider:

Do you have to take a mental step to look at other parts of the code to determine the functionality of a variable or is it in a context where it doesn’t matter?
Short Names Can Be OK

// please forgive my mixed language use
(for int i = 0; i < MAX_ITEM_COUNT; i++) {
    listVal[i] = 0;
}

You KNOW that “i” is just iterating over this data. What meaning could it have?
We can readily see two things with this example

1. a named value MAX_ITEM_COUNT instead of a literal

2. The variable name “i”. The context allows the use of a meaningless (but conventional*) variable name

*if I had used “x” in place of “i”, it might make you wonder if this were a coordinate because “x” has existing conventional use.
Let me draw your attention to what this does for you in terms of naming.

```java
public enum Evaluation {
    InvalidInput, /* The string provided does not describe a valid tic-tac-toe board */
    NoWinner, /* valid board, but the game hasn't been won by either player yet */
    Xwins, /* valid board, and the game was won by player X */
    Owins, /* valid board, and the game was won by player Y */
    UnreachableState /* string describes a board, but not a board state that is reachable from a valid game */
}
```
Extensibility

We discuss this on the current assignment much more for the purpose of focusing your attention on the concept than on the expectation that your code will be fully extended with ease.

In general terms, writing your code to be flexible/extensible is always good practice. It is difficult to predict when an application will need to be extended or a piece of code will need to be reused because it was originally written for a narrow purpose and fits better somewhere else.
“My code is tailored to 3 by 3 boards and bigger boards are invalid.

1) Does that mean the board length, m, n, and k variables should be static because they aren't unique to a board object?

2) Or should it be not static because code should be easily modifiable for bigger boards?”

*m, n, k (rows, columns, in-a-row-to-win) - define both the size of the board AND imply some rules
I discussed three levels of extensibility last week:

1. Recompile
2. Initialize values at startup
3. Reconfigure while running

\*m, n, k impact the rules of the game so they impact the evaluation of the game and even the tests you would write.
Making a truly general solution is worth extra credit because it is beyond the intended scope of the assignment.
If you are hosting a TicTacToe server or you have a game that keeps expanding as the player reaches a certain level, you would want game objects to be that flexible.

- In general, you should still use names instead of literals (avoiding m, n, k because they have no intrinsic meaning) throughout your code.
- Your code could be relatively fragile still, because some things would not translate directly, but it would be more easily modified to a new case.
- It does NOT have to be the general case solution for meaningfully named values to be used.

*fragile code - breaks when you make simple modifications
*brittle code - resists change and breaks when you do
TicTacToe Testing

This bit of code below that was provided to you and gives you clues.

```java
public enum Evaluation {
    InvalidInput, /* The string provided does not describe a valid tic-tac-toe board */
    NoWinner, /* valid board, but the game hasn't been won by either player yet */
    Xwins, /* valid board, and the game was won by player X */
    Owins, /* valid board, and the game was won by player Y */
    UnreachableState /* string describes a board, but not a board state that is reachable from a valid game */
}
```

What about all of the variations of different characters?
What about variations based on mixed cases? (I did touch on this briefly last week)

What are the potential return values?
What conditions produce them?
What about combinations?
    Xwins AND invalid
    Xwins AND Owins
More Resources

You Still Don’t Know How to Do Unit Testing (and Your Secret is Safe with Me)

Create Tests in IntelliJ

Generically speaking: Java Assert();
https://www.geeksforgeeks.org/assertions-in-java/
Kinds of Comments (worst to best)

1. Repeat of code
2. Explanation of code
3. Marker in code
4. Summary of code
5. Description of code’s intent

* “Information that can not possibly be expressed by the code itself”*
1) Repeat of Code

Tells what the code does in words. Basically useless. This is what I call “comment pollution”.

```java
// assign 100 to a
int a = 100;
```
2) Explanation of Code

If code is so complicated or confusing that it needs explanation then it probably needs rewriting

Code:

```c
memcpy(tstr,f_rec,8);
i=0;
while ((tstr[i] != 0x20) && (i<8))
    i++;
    tstr[i]=0;
```

Explanation(s):

// copy the first 8 chars of the directory entry data
memcpy(tstr,f_rec,8);

// scan the 8 chars above for a space, don’t go over 8
i=0;
while ((tstr[i] != 0x20) && (i<8))
    i++;
// terminate with a 0
    tstr[i]=0;

*note: 0x20 and 8 are magic numbers. Name them!
3) Marker in Code

// TODO: make changes here

// MJW012111: Matt, I commented out the line below and replaced it with the line that follows it
4) Summary of Code

Simple summary of code into one or two sentences

// Function -
// read the file indicated by the current dirEntry and rebuild // it on the local disk by successively reading and writing // clusters that make up the file on the disk being recovered

*this is still not the best comment.*
5) Description of Code’s Intent (Most useful)

// extract the root of the filename from the directory entry

memcpy(tstr, f_rec, 8);
i=0;
while ((tstr[i] != 0x20) && (i<8))
    i++;
tstr[i]=0;

// * still, this should be rewritten
*Information that can not...

- Copyright notices
- Confidentiality
- Comment blocks assigned by instructor
- Citations of sources

Example:

```
//---------------------------------
// Copyright 2007, eV Interacitve, Inc.
// Modification of this code is not permitted without express written
// consent of Matthew Harmon ***********@ev-interactive.com
//---------------------------------
```

Example:

```
//---------------------------------
// Lambert Conformal Conic Project
// Formula on page for the sphere, pg. 106
// U.S. Geological Survey Professional Paper 1395
//---------------------------------
```
Excuses for Bad or No Comments

“It Takes Too Much Time”

- It takes more time later
- harder to debug which then takes more time
- Commenting after the fact is more difficult and takes more time
- My experience and that of EVERY professional that I have spoken to says otherwise
Style: Comment Blocks

Example 1

/*--------------------------------------*/
/* here's a difficult style of */
/* comment block to maintain */
/* You'll spend too much time */
/* keeping the right hand column lined up */
/*--------------------------------------*/

(keep in mind that your code might not be viewed with a color-coded syntax highlighting system)

Example 2

//----------------------------------------------------------------------------------
// this style is easier because you don't
// have to align the right hand column
// you can just copy and paste the dashes
// to start and end the block
//----------------------------------------------------------------------------------
Commenting Techniques

Write comments at the level of the code’s intent
Focus on the code itself so that comments enhance good code
Focus paragraph comments on why rather than how

1. Endline comments
   a. Tend to be cryptic
   b. Hard to maintain
   c. OK for data declarations
   d. OK to denote bug fixes
   e. OK for marking ends of blocks
   f. Some people use this method to pair up curly braces
      i. e.g. – “} // end while”
   g. Probably not very useful
2. Commenting individual lines
   a. Complicated line of code (which you should just rewrite)
   b. Bug repair
   c. Overuse creates “speedbumps”
3. Commenting Paragraphs (blocks)

```c
//==============================================
// get next cluster fat_index=0;
//==============================================
if (cluster >= (FAT_ENTRIES/2))
{
    cluster = cluster-(FAT_ENTRIES/2); fat_index++;
}
cluster = *(F_tbl[fat_index]+cluster);
```

The better style would be:

```c
cluster = getNextCluster(F_tbl, cluster);
```
Comment the Why, not the How

Example from "recover.c"

```c
if (track >= 1024)
    printf("%i WARNING (cylinder) in %s\n",status, filename);
else
    write_count = write(outFile,io_buff, bytes_2_write);
```

Preferable:
// this BIOS can not read tracks greater than 1024 without special drivers
// print warning but keep recovering the file anyway

```c
if (track >= LAST_LEGAL_TRACK)
    printf("%i WARNING (cylinder) in %s\n",status,filename);
else
    write_count = write(outFile,io_buff,bytes_2_write);
//*notice the literal or magic number “1024” replaced with “LAST_LEGAL_TRACK”
```

Commenting Routines

- Say what the routine WON’T do, mention permissible input values
- Document global effects (if any)
- Side effects (are dangerous)
- Create or destroy anything?
- Document source of algorithms
- Avoid enormous comment blocks
  - I like some comments before every routine for visual delineation at the very least
Balance

- useful amount of commenting
  - If it takes too much time to wade through comments then there are too many comments
  - If there’s as much or more comments than code then there are too many comments
  - Rule of thumb: 1 line of comment for 10 lines of code
  - Do not count lines and then add comments every 10th line!

- Avoid "comment pollution"
  - inane commenting
    - *inane* - adjective
      1. silly; stupid.

- Avoid "speed bumps" - comments with no redeeming value, they do not improve comprehension

- Metrics or "rules of thumb" such as "comments per 10 lines of code" are not particularly useful but they are intended to give some idea as to what is considered reasonable by the "ruling body"