Software Engineering

Writing code is just one aspect of crafting software and applications. You need to understand design, testing, deployment, licensing, security, monitoring, managing, and support of software, including the processes and procedures used in teams. One who does all (or most of) that is often called a *software engineer* (as opposed to a programmer or developer).

Generally, *software engineering* is more about the management of and process of software development, although it definitely encompasses development too.

While you will need some experience before you can get hired to design and develop some business-critical software (entry-level jobs are usually in programming and support), you should become familiar with a few concepts and terms. (Qu: Why? Ans: to communicate with software engineers you work with and to understand their needs so you don't become frustrated. Also, to grow into a more senior position.) See monster.com for a sample SE job description.

> Note that the U.S. Federal government allows the title "software engineer" for almost any computer-related field, but some states (and some other countries) have high standards for any job title including the word "engineer". Florida is one state where this is true. Professionals (in the legal sense) are held to a higher standard of work and conduct than non-professionals.

In COP-2805C (Java II), we will learn more about these topics, focusing on professional software development techniques and tools. As we begin to develop more complex classes and programs, you need to know a few of the concepts and terms before next semester:

JavaBean Standard

Some classes define *getter* and *setter* (*accessor* and *mutator*) methods, for each of the object's properties. In general, this is a bad idea. Fields are only there to support the methods, and as an implementation issue are usually declared `private`. Sometimes, however, it is a good idea. For example, Graphics.setXXX methods.

> Too many such methods are usually a symptom of a poor design. Avoid the mistake of have "dumb" objects, each with getters/setters (essentially, just database records wrapped in objects), and one "brain" main method that does all the work. This is not object-oriented!

The ***JavaBean*** standard for objects requires a pair of methods for each *property*, with names **`get`**`Propname` and **`set`**`Propname`. Boolean properties may use **`is`**`Propname` for the getter instead. Read-only properties don't have a setter.

Enterprise frameworks and other advanced Java software use classes that follow this standard. Objects of such classes are known as Java Beans, or more commonly, just ***beans***. By following the standard, these objects can be treated as "block-box" components, added to applications without knowing the code. They can communicate

with other beans using events (the *observable* pattern), be saved and restored as XML text, and even edited in a GUI (that is, a GUI bean editor can automatically detect all the properties and show them in an editor view. With a bunch of beans, you can actually assemble a Java program without writing any code! (Note Swing components are beans, allowing building user interfaces with drag and drop in IDEs.) See the JavaBean Trail in Oracle's Java Tutorial for more.

When to Define Methods

Start with class and object design (below), which comes from the software requirements. The initial methods you invent should correspond to the *messages* (or *commands*) invoked on objects. These methods define the possible *behavior* of the objects and should be generally declared `public`. Should these methods be unwieldy, you can define methods to remove duplicated code or to simplify or shorten the methods. Such methods should be declared as `private` as they are just implementation methods, and you should feel free to change them.

Rule of thumb: Keep methods focused on a single task, and shorter than a screenful or "pageful". (Actually, even shorter is better: good methods are often 10 or fewer lines long.)

Some methods are *basic utility methods* (such as `Math.cos`) that don't depend on any object. Such methods should be defined as `static`. General utility methods that can be reused in other projects should be `public` and be coded to be general purpose, if convenient.

> Actually, the initial design is to decide what classes to have, and the responsibilities of each. For each class, then you decide what public methods are needed to meet those responsibilities.

In class program: Show **MsgBox.java**, **MsgBox2.java**, **MsgBox3.java** programs that has main ask the user for a line of input and then prints it within a box. Have students complete similar program that invokes a method *boxMsg(String message)* to draw the box around the String using `System.out.print()` and/or `System.out.println()` for output. (Hint: Use `String.length()` and ignore Unicode issues.)

```
+----------------------+
| Danger Will Robinson! |
+----------------------+
```

Qu: Consider the `drawLine` method of `MsgBox`. What parameters should it take to make it more generally reusable in other applications? Ans: indent, char to draw with.

Design Guidelines for Methods and Classes: Abstraction and Encapsulation

Abstraction: the process of finding the essential (and **relevant for the current problem**) methods (behaviors) and fields (properties) of classes. You *abstract* the objects in the problem domain to decide what the classes should be; you can write a verbal description and examine all the nouns.

Each class should be concerned with a single entity (or set of related operations, when creating *utility* classes such as `java.lang.Math`). Note that a single entity in the problem domain may be represented by several objects in your program (ex: HCC new registration system, what are the objects?). The reason is to keep each class focused on a single concept ("separation of concerns").

A class is ***cohesive*** if its methods and properties relate to a single abstraction.

Use **top-down design** for the methods discovered above. This may lead to additional `private` methods.

The instance (and class) variables define the properties of the objects. (Q: What does an object have to "know" or "remember" between method calls?)

Encapsulation is putting related items in a single place, and then hiding (making "private") the implementation details. (Most properties are private.)

Coupling is the amount classes are related to each other (that is, how "spread out" your abstractions are). Proper encapsulation reduces coupling.

The methods define what the object's behavior, what it can do. Usually there are public methods for reading and changing the values of the private or package-private properties. (Better than having public properties, since that increases ***coupling*** and lowers the ***data-hiding*** and ***encapsulation*** that is so desirable to decrease debugging and maintenance costs. Also mention ***cohesion***.) The public interface of a class, package, or module is known as its API. You want code in one class/package/module to only use the public API of another. Keep in mind that a public API usually lives (and must be maintained) forever! You can always make something public later, but can never hide something that was public initially.

There are other design principles you should know. In the Java II course, you will learn about the SRP (single responsibility principle) and DIP (dependency inversion principle). For now, I will leave you will a simple illustration of these: separation of concerns. The `main` method should not both be responsible for creating and wiring-up (or hooking-up) the dependencies of your objects, and also responsible for running the application. It is better (and common) to have a separate class just for the `main` method (the class is often called "`Main`"), which creates the basic objects and connects everything together, then calls some other method (often in another class) to actually run the application. As much as possible, all methods (and classes for that matter) should only have a single responsibility.

> As it happens, not all Java classes are templates for objects. Some are templates for data structures, or *structs*. What's the difference? Objects are black boxes with public methods. A struct generally has no methods and all public fields (or equivalently, public getter and setter methods). (Remember class `IntWrap`?) An example might be class `Point`, with two public fields `x` and `y` and no methods. A more common example is working with real applications that store data in files or databases. When transferring data to/from your program, you put the data in a struct, often called a *data transfer object* (or DTO).

When designing a class as a template for objects, make the data private and avoid any get/set methods if possible. When defining a struct or DTO, avoid instance methods and either make the fields public or have public getter and setter methods.

Java SE since 14 include a preview feature for this purpose, called *records*. A Java record is similar to a Python tuple. Look for it in Java SE 16. *(Mention Lombok if time.)*

What makes for good comments?

Know your audience; generally experienced programmers. The important thing about comments is that you don't write them for yourself. You write them for future maintainers (possibly your own future self). They are not intended for programming novices (except for teaching demos). All comments should be spell-checked and clear in what they convey to the reader. In general, comments should appear near the code they describe, such as at the end of a line of code or in a separate line just above the code.

A goal of writing good documentation is to anticipate the readers' perspectives.

There are different types of comments:

- ***What* comments refer to the actual steps taken.** Use *what* comments to describe what a chunk (also called a code *fragment* or *snippet*) of code is doing. Simple, clear code needs fewer *what* comments. If during a code review you ask the author what a chunk of code is doing, it either means that their code is unclear and/or the *what* comments are deficient. Complex formulas, algorithms, Regular expressions, and formats (file formats, date/time formats, and others), all need *what* comments. When brain-storming using pseudocode, the steps you discover can become *what* comments which are place-holders for code you intend to add, and you fill in the code later. (These are often called ***TODO*** comments, and often you put that word in to find such spots easily. Once you added the code, if the result is short and clear you then remove the comment. (It is redundant "noise" at this point.)

  Unclear code should not have *what* comments added. Instead, fix up the code! (This is known as *refactoring*.)

  Linus Torvalds (the inventor of Linux and of Git) said this (in the kernel code style document): "Comments are good, but there is also a danger of over-commenting. NEVER try to explain HOW your code works in a [*what*] comment: it's much better to write the code so that the **working** is obvious, and it's a waste of time to explain badly written code."

  While I agree with that, if you cannot come up with elegant solutions that are short and obvious, *what* comments are better than nothing. Maybe your comments will inspire others to invent more elegant code for your task. Here's an example of a good *what* comment for a regular expression that is not obvious:

```
// Format standard email dates, for example:
// Fri, 13 Mar 2020 11:29:05 -0800:
String RFC5322DateTimeRegExPattern =
"^(?:\s*(Sun|Mon|Tue|Wed|Thu|Fri|Sat),\s*)?(0?[1-9]|[1-
2][0-
9]|3[01])\s+(Jan|Feb|Mar|Apr|May|Jun|Jul|Aug|Sep|Oct|No
v|Dec)\s+(19[0-9]{2}|[2-9][0-9]{3}|[0-9]{2})\s+(2[0-
3]|[0-1][0-9]):([0-5][0-9])(?::(60|[0-5][0-9]))?\s+([-
\+][0-9]{2}[0-5][0-
9]|(?:UT|GMT|(?:E|C|M|P)(?:ST|DT)|[A-IK-
Z]))(\s*\((\\(|\\)|(?<=[^\\])\((?<C>)|(?<=[^\\])\)(?<
-C>)|[^\(\)]*)*(?(C)(?!))\))*\s*$";
```

(I bet even Linus would agree to have comments here!)

- ***Why* comments refer to the reasons for writing the code in a particular way.** A *why* comment is for explaining a particular implementation decision or the programmer's intent, especially if it's not the "obvious" design choice. If the obvious choice is to use an `int` but you use a `double` or (even stranger) a `String`, a *why* comment is useful.

  There can be many reasons why some code was written in a non-obvious way. Some examples include formulas that were rearranged to reduce round-off error or to avoid overflow, code you are not allowed change, and code requiring non-intuitive design for security reasons (say to eliminate side-channel attacks).

  Why comments are also used to document the reasons for the code, often listing a reference to an issue (or ticket) in some issue-tracking system.

  Here's an example of a good *why* comment, adapted from Robert Martin's *Clean Code* (p. 59):

  ```
  // The following trim is required, since it
  // removes leading space that would cause the
  // the item to be recognized as another list:
  String listItemContent = match.group(3).trim();
  ```

- **A third type of comment is a *how* comment.** These are for libraries and reusable modules, and tell the reader how to use your code with their code. The Java "doc" (or "API") comments for Java SE is an example of this. (Such docs are usually extracted from comments in the source code.) *How* comments are used to describe how to use methods, fields, and classes of your code. Generally, only `public` methods and fields need these, but never hurt on private methods too. *How* comments might describe method argument types and ranges, return value type and range, method semantics, pre- and post- conditions, and use cases (example code). They can include warnings (such as *this method is not thread-safe*), examples, and references to other documentation. *What* and *why* comments are generally not included in such API docs.

The *how* or *API* comments (also called *doc comments*) are specially treated in Java, and are converted to HTML. These will be discussed at a later time.

- **A fourth type of comment is a *required* comment.** These are the comments you must have at the top of each source code file. If your code is under any sort of license or copyright, you can either list the license in the comments or include a link to the license. (I've seen source code file with over 100 lines of such comments at the top!)

  What else is required depends on circumstances. You might need to identify the author(s) who wrote the code and purpose of the code. If any of the code came from others (taken from StackOverflow for example), include references. For our class, you must include the purpose of the file ("project x, to do such-and-such"), the names of all authors, and any collaborations you used ("I found an example of this at URL xyz").

  Required comments include compliance information, such as what standards the code must meet. For example, security standards, financial standards, military standards, etc. There are standards for code that runs on airplanes and boats. A common example today is compliance with the EU's GDPR (a European privacy standard). Such comments may include a notice about who and when a compliance audit was performed.

  If the code is online or you use online systems to manage the software development process (code review, issue tracker, wiki), have links for those as well.

  Every organization will have different rules for required comments.

Bad comments should be eliminated. Some examples of bad/useless comments include "`//initialize variables`", "`x = 2; //set x to 2`", "`num = num + 1; // add 1 to num`", "`//Default constructor:`", etc.

Often you can eliminate such bad comments by choosing better names. For example:

```
p = n * e + t; // Compute price
```

can be written as:

```
price = quantity * priceEach + tax;
```

The same applies to method names.

Avoid ambiguous comments, for example:

```
// No file found means all defaults are loaded
```

That must have meant something to the author, but what? Is this a placeholder to remind the programmer to do something later, or a statement that some other code was supposed to load defaults, or something else?

Comments are a vital part of your code. Always keep them clear, correct, and useful. Do not forget to update comments whenever you update the related code.

Programming Style and Program Development Process

Programs are read more often than they are written. Others must read your code. Do not program sloppily, thinking you'll "pretty it up later". That never happens!

First, design your program; do not design "at the keyboard" (writing code before you know what the code is supposed to do). Once you have the main design worked out (not necessarily all the details), only then should you start writing code.

> Every organization has a definition of acceptable "program style". Fortunately, each language has a generally accepted style documented someplace, and usually organizations follow that closely. Code that violates the style rules is deemed unacceptable (and there are automatic style checkers used). The exact style followed by a team of developers does not matter as much as the fact that they all follow the same style rules.

While many workflows are possible when crafting software, here's a procedure that saves time and works well with the few tools and techniques you already know (as beginning programmers):

Start with the "what" comments. Decide on (public) class and method names to begin. Put in empty public methods (except for their comments), with the correct *method signatures*. These are called *stub methods* (or simply "***stubs***"). At this point you have a do-nothing ***skeleton*** program that will compile without errors. (Non-void stub methods also need a "fake" `return` statement or they won't compile, e.g. "`return 0; //Stub`".) The comments here are *what* comments: a list of tasks done by that method, in the correct order. **If you cannot add complete *what* comments to your skeleton program, you do not understand the task yet.**

These what comments are a form of *pseudocode*. At the start, you are merely brainstorming, putting in the comments where you think the matching code should go, changing them around, and trying alternatives. Each such comment should be simple enough that you can implement it in a few lines of Java code. (And if the code is clear, you could remove the *what* comment; but please don't in our class, so your instructor can see your design.)

> Sometimes you may not know what the steps should be. It is perfectly reasonable "to play with code", to see what might work. But once you have the right idea, throw away that experimental, non-commented and non-designed code, and add in the *what* comments to your skeleton that you now know to be correct. This is how you should document your design of your methods in our course. (In the Java2 course, you will learn better ways.)

It is far, far easier to follow the correct style rules as you type, then to use sloppy programming and try to make it nicer later. Remember that you may need to show others your code before you are done, and if they cannot understand it or it is too difficult for them to read it, you will not get any feedback other than "*clean up this mess*". (Some IDEs can enforce style rules for you.)

Use indenting and blank lines to improve readability; they do not slow down your program! Use the proper indentation style shown in class and in the examples. For most programs, you should indent 4 spaces each time. (Use spaces, never tabs. Most editors have settings to convert tab to spaces as you type.)

Keep your lines shorter than 80 characters. Remember you can always break up long lines into two or more shorter ones.

Now that you have a design in pseudocode comments, you are ready to write actual code. Add declarations for the variables and objects you have decided to use. Choose an appropriate type for your variables, planning for the future if the values become large (to avoid overflow; see below). Choose descriptive names and use a consistent naming style.

To write the Java code for the methods of your classes, start working on one class and one method at a time. Look at the *what* comments in that method, and pick one of them to implement for now. Put the code for that one task below the comment you already have. As you code, you can add other comments of course. And if your design changes ("*Drat! This won't work!*" or "*Ah-Ha! A brainstorm!"*), you can alter the *what* comments as well.

After each *chunk* of a few lines is coded (using correct style!) that implement the one *what* comment, **save the file** and then check for *syntax errors* by compiling. **Do not move onto the next chunk until what you have done so far compiles without error.** Finding and fixing bugs in a few lines is far easier than finding and fixing bugs in dozens or hundreds of lines!

> After working for a while, save your work to a backup such as a flash drive or cloud storage. Nothing is more frustrating than losing hours or days of work because your file got corrupted or deleted!

As you code, you need to pause every so often and reflect on the big picture:

Math related logic errors include overflow/underflow (ex: byte b = 100 + 100;), floating point divide by zero (+Infinity, -Infinity, NaN), and floating-point round-off errors (which require special comparison techniques). **A programmer must ensure this doesn't happen.** In fact, Java 8 added `Math.addExact` method (and other `Math.*Exact` methods), which throws an exception on over/under-flow. You should consider using these methods.

As you finish a method's code, you should pause and spend some time to reflect on the code before moving on, to see if you missed any error checking or other issues, or just to make sure the code is as clear and simple as possible: Is you code free of logic errors? Have you made your code safe and secure? Are there any compliance issues you need to address? Can the code be improved? Always allow time for this and to change the code if necessary, a process known as ***refactoring***.

> In 11/2012, the Stockholm Stock Exchange received an order for 4.3 billion futures at a unit price of 107,000 SEK. This order would have cost nearly 460 SEK trillion, or 131 times the entire GDP of Sweden. The bug that caused this false

order brought the exchange to a halt. It appeared to be an integer underflow that created the 4.2 billion future order.

Other issues to watch for include security concerns and compliance with various laws and regulations.

Debugging

Just because your code compiles does not mean it is error (bug) free! You need to run the program with carefully selected input and verify the output is expected. This is known as testing. But what if it isn't right and the tests fail? You need to locate the point in your code where you did something wrong (or forgot to do something), then fix it. This process is known as *debugging* and there are many different tools and techniques that can help with this. For now, here are a few methods that do not require learning any additional tools or the ability to understand low-level system details. (As you grow as software developers, you will learn other tools for debugging specific types of issues.)

To start with, **ignore all compiler error messages except the first**. The reason is that the first error may cause the other error messages, and can hide additional errors. So always solve the first syntax error and ignore the rest. Repeat until no syntax errors remain and your code compiles.

To solve crashes or tools (such as IDEs) not working, you can **copy the error message and paste it into a search engine**. The odds are very low that you are the first person to have an issue with some existing software. Be sure to strip out of the error messages any specific data from your code, such as your file name and line number of the error. If you don't find any results, it is likely the error was caused by something else, such as missing software or incorrectly configured software. The various stackexchange.com sites are generally considered reliable, especially stackoverflow.com, so when searching, see if you can get an answer from there.

The final debugging technique you can use right now (as beginners) is known as ***scaffolding***. When your program runs but gives incorrect results, you need to narrow down the part of your code where the error lies. Add `System.out.println()` statements to print the values of variables and expressions at different points in your code. The bug will be between the last print statement showing correct values and the next print statement showing incorrect values. These print statements are meant to be removed from the final working program. You can put scaffolding before some formula to show the values used, and afterwards to show the result; in loops to see how many times to loop goes around, at the top of methods to show the values passed in, etc.

If you are unsure where the bug is (and assuming there is only one!), put the print statement showing the input data and the results near the end. If the results are wrong, the error is earlier in your code. You can move the print statement to half-way between the last position and the top. (If you have multiple scaffold statements, move the one showing the wrong results half-way between its current position and the previous print statement showing correct results.) This is known as ***bisection***.

It's not always the case a developer has source code they can modify with scaffolding. In some cases, you need to debug software from a third party without source code available. With source code, you can use the debuggers built into IDEs such as Eclipse or NetBeans. There are stand-alone debuggers as well that don't require source code. Some include the primitive command line tool `jdb` (part of the JDK), Jswat, and YourKit.

Demo JDK 8 monitoring and debugging tools: Start some Java app (Davmail is good), then run `visualvm`. (Add some plugins esp. visual GC.) Other tools: `jps`, then `jmap -heap` *`PID`*. Also: `jmap -histo:live` *`PID`*. To browse the heap, use `jmap -dump:live,format=b,file=dump.dat` *`PID`*, and then `jhat dump.dat` and open browser to `http://localhost:7000/`. Also show `jconsole` *`PID`*.

Do not re-invent the wheel! Reuse an existing class if one exists that will do the job. When creating your own classes, think about designing in some features that will allow you and others to reuse your class in future applications. This is what the huge number of Java packages (of classes) is for.

Try to write efficient, readable code that minimizes round-off errors (by re-arranging formulas). For example: Calculate a common expression once and store the result in a variable. **Declare variables as needed** and initialize at the same time.

Testing: Most classes represent objects such as Person, Color, Button, etc., and are not full programs. You can add a `main` method to such classes, solely to test each method of that class: have your main method create an object of the class and call its methods with test data. Then print the result and the expected result. This sort of testing (sometimes called ***ad-hoc***) is better than no testing at all, but better testing is very important in the real world and will be discussed in the future.