Lesson: Software Engineering Beam

Software engineering is the study of the design, construction, and maintenance of large software systems. As the hardware capabilities of computers have increased, so have the expectations for the performance of software. We expect programs to be friendly, easy to use, reliable, well documented, and attractive. Meeting these expectations often increases the size and complexity of a program. Thus, over time, the average size of programs has tended to increase.

Many software systems represent a significant number of person-years of effort and are written by large teams of programmers. These systems are so vast that they are beyond the comprehension of any single individual. This is most likely unlike the kinds of programs that you have worked on (probably by yourself) so far. As computer systems become more and more intertwined into the fabric of modern life, the need for reliable software steadily increases. As an example take the cellular telephone network. This system consists of millions of lines of code written by thousands of people over decades; yet, it is expected to perform with complete reliability 24 hours a day, 7 days a week.

Unfortunately, whereas the scaling up of hardware has been a straightforward engineering exercise, software production cannot be so easily increased to meet escalating demand. This is because software consists of algorithms that are essentially written by hand, one line at a time. As the demands of increasing reliability and usability have led to software systems that can not be understood by a single person, questions concerning the organization of teams of programmers have become critical.

These managerial issues are complicated by the unique nature of software production. Programming is a challenging task in its own right, but its complexity is increased many fold by the need to divide a problem among a large group of workers. How are such projects planned and completion dates specified? How can large projects be organized so that individual software designers and programmers can come and go without endangering the stability of the project? These are just some of the questions addressed by software engineering.

One of the cornerstones of software engineering is the software life cycle.

Definition: *The **software life cycle** is a model of how software is developed, used, maintained over time, and eventually discarded.*

This model is helpful in predicting costs and allocating programming resources, but it suffers from inflexibility. This inflexibility has led to the adoption of alternative software development models in recent years, such as rapid prototyping. Newer models tend to focus on improving user satisfaction with the software (e.g., making programs more usable and user-friendly).

In addition to managing the development of individual software projects, software engineers also design tools for automating portions of the software development process. For example, tools to assist with the creation of program documentation and testing are now common. Such tools are often referred to as CASE (Computer Aided Software Engineering) tools.

Analysis and design

When working on large software projects, rarely do we ever begin in front of a computer monitor and keyboard, and begin to code. For simple programs that you are likely to design during your educational experience, perhaps this is fine. The kinds of projects that computing professionals work on are rarely, if ever, so simple. Just as understanding the problem is crucial to ultimately solving it, so is understanding the requirements of a project in order to ultimately design a solution that does what we want. A large part of software engineering is analyzing what we want in order to ultimately design what we want.

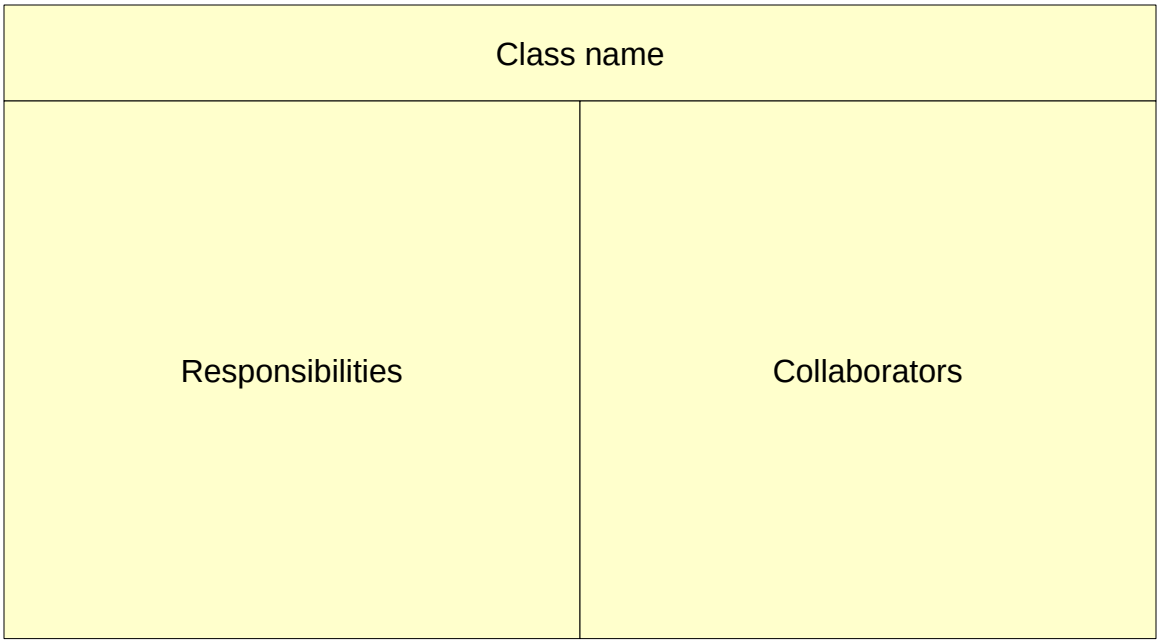
The process of analyzing what we want in order to specify the requirements of a system can often be time consuming. In fact, this part of the process often involves a customer or client who will ultimately be the recipient of the produced system. You may be thinking that the knowledge base of the customer is probably very different from the knowledge base of a programmer (or an organization that is tasked with designing the system for the customer). And you would probably be right.

The process of analyzing the requirements of a system is often done in a game-like manner, playing out scenarios. One method, called the **verb-noun method**, attempts to identify things or entities involved in the system. These entities are nouns that may ultimately translate to classes and objects in a programming project. In addition, playing out scenarios by having the entities interact with each other helps to discover actions that involve them. These actions, or verbs, may ultimately lead to interactions between objects in a programming project. In fact, this may lead to the discovery and implementation of methods and functions.

One way to help discover entities in a system is to use CRC (Class Responsibility Collaborators) cards.

Definition: ***CRC cards*** *attempt to capture the classes (nouns), their responsibilities (verbs), and their potential collaborators (other classes that are specified on other CRC cards).*

The goal of using CRC cards is ultimately to play out scenarios involving the classes. This helps to analyze system requirements and to discover object interactions. In addition, it helps to see if the problem description is clear and complete. Ultimately, class interfaces can be designed from the CRC cards while playing through various scenarios. Below is a sample CRC card:



Documentation
Documentation is arguably very important; yet, it is often the most despised aspect of engineering software. In fact, many developers put it off until the end of a software project. This often leads to inconsistent and incomplete documentation of the project! Usually, it is good practice to write class and method comments that describe the **purpose** of each. This ensures that the focus of the comments is on *what / why* instead of *how*. It also ensures that we don't forget what something does or is responsible for. As a quick example, the following is what NOT to do:

```
# add one to count
count += 1
```

This comment is pointless since anyone who can read Python code can clearly see the variable count is being incremented. The comment is really just noise and doesn't add anything of benefit. Instead, try something like this:

```
# account for the new user being added to the system after user
# registration
count += 1
```

This is far more useful as it describes the reason why count is changing and the overall purpose and logic behind it being incremented. This adds information that a fellow developer wouldn't know by just looking at the code.

When working on a software project, it is easy to remember what something does, why decisions were made, and so on. But after being removed from the project for some time, it is often difficult to recall programming decisions. In fact, many programmers often start over on small projects rather than go through existing code that is not commented properly to try to understand it.

Nowadays, there are many tools that can automate the generation of a user manual or documentation of an application using embedded comments in the source code. The use of special, meaningful tags are interpreted by this type of software, which allows it to render the documentation.

Prototyping
Often, we wish to observe and test an application before it is entirely finished. That is, we would like to observe its use in order to perhaps drive further design decisions. Prototyping involves generating a preliminary model of an application to help support early investigation. It allows us to test certain functionality while ignoring other details. Usually, a prototype is not complete; that is, it is not a complete, finished version of the application. Often, the incomplete components are simulated (or sometimes *hard-coded* with temporary values or behaviors that will be replaced later on). In prototypes, we try to avoid random behavior, as it is difficult to reproduce.

Iterative software development
The process of creating software often involves many iterations. In iterative software development, the idea is to implement the basic functionality of a software project and subsequently test to make sure that it works. The next iteration will add more features to the project. Further testing will be done to ensure that the implemented changes work properly; however, retesting the features of the first iteration (all previous iterations, actually) is crucial, as changes made to the application may have inadvertently *broken* things.

Using early prototyping by developing prototypes early in the development process is key in iterative software development. It allows developers to interact frequently with the customer. In fact, iteration is not just performed during coding. It is also performed during analysis, design, prototype generation, and feedback.

For simple programs (where you work alone), iteration typically occurs during design and prototype generation. In the end, the goal is the same: to take small steps towards completion of the project. The end of each step is marked with a period of testing. The goal is to fix errors early. If necessary, earlier design decisions can be revisited.

Generally, robust software requires thoughtful processes to be followed with integrity. The goal is to analyze carefully, specify clearly, design thoroughly, implement and test incrementally, review, revise, and learn. Errors found are treated as successes rather than failures. In fact, there is no such things as error free software!

Errors
Undoubtedly, you will encounter many errors when designing and implementing software projects. Some will be easy to find and fix; others won't. In fact, some will be undetectable (and therefore, never fixed!).

When writing programs, *early* errors are known as syntax errors.

Definition: ***Syntax errors*** *are errors that the compiler (or interpreter) spots; as such, they are easy to detect.*

Fixing syntax errors is usually straightforward, because today's compilers are pretty good about locating the source of syntax errors.

Later errors are usually logic errors.

Definition: ***Logic errors*** *are bugs that the compiler cannot spot and can lead to incorrect results or abnormal program behavior.*

In fact, some logic errors are not immediately obvious. Logic errors can be, for example, making a mistake in a calculation (e.g., adding instead of subtracting), making an incorrect assumption about input data, and so on. Logic errors are hard to find because a program still runs even with the logic error. The end result may be incorrect; however, that may not be so obvious (depending on the purpose of the application).

Definition: ***Runtime errors*** *are also later errors that occur during the execution of a program.*

If your program attempts to open a file that isn't found, for example, it will generate a runtime error. The error doesn't occur if the file is found; therefore, runtime errors may or may not occur (depending on runtime environment).

Most commercial software is loaded with later errors. This begs the question, are errors truly preventable? If not, what can we do about detecting them? Arguably, we need tactics for both preventing and detecting errors. If we operate under the assumption that errors will never be entirely preventable, then we must also consider detecting them. With software engineering techniques, we can lessen the likelihood of errors and improve the chances of error detection.

In software projects, there are several methods of error prevention and detection.

Definition: ***Error reporting*** *involves providing information about errors that have been detected.*

In error reporting, how is this information provided? Is it just printed out to the console? In fact, is there even a human user? Perhaps the application is automated and never executed by a human in front of a computer! A technique, then, may be to send an error message to an object. A better technique may be to return a diagnostic value after completing some action. The preferred technique, however, is to generate exceptions. This will be discussed later in this lesson.

Definition: ***Error handling*** *is the process of dealing with errors programmatically. Should an error occur, the application internally handles it.*

Error handling requires anticipating failure. A typical point of failure involves obtaining user input. Users vary, and predicting what input they provide is not always possible. Therefore, the sanitation of user input is crucial to preventing many errors.

Definition: ***Sanitizing input*** *often requires ensuring that the provided input is of the proper type and within the proper range. It usually manifests itself in source code as checking the type and value of arguments passed in to methods.*

Exceptions and exception handling

Definition: *In programming, an **exception** is the interruption of the normal flow of a program. It is something abnormal, yet sometimes expected. Exceptions are a language feature that is predicated on the fact that errors can't be ignored and can often be recovered from.*

In Python, exceptions can (and often do) occur. They can be detected and programmatically handled through a **try-except** block. The **try** block identifies statements that, when performed, may cause an exception. The statements in the **except** block are *only* executed if an exception occurs in any of the statements in the try block. It provides code that handles the exception. You may recall this in some of the RPi projects that involved the GPIO pins. The behavior of some of the applications in these projects was to run forever (well, until the user presses Ctrl+C). It is important to ensure that the GPIO pins are reset before terminating a program. Pressing Ctrl+C generates a KeyboardInterrupt exception. The **KeyboardInterrupt** exception is defined in Python as some keyboard activity that interrupts the normal flow of the program. Pressing Ctrl+C does exactly this.
Several of the activities that you worked on detected (and handled this exception by cleaning up the GPIO pins). Here is a snippet of code from one such past RPi activity:

```
try:
    while (True):
        # do stuff
except KeyboardInterrupt:
    GPIO.cleanup()
```

The statements in the try block occur forever (i.e., in the `while(True)` loop). Pressing Ctrl+C generates an exception. The except block detects this exception and handles it by cleaning up the GPIO pins. Since the exception is programmatically handled, the interpreter doesn't terminate the program and dump information to the console about the exception. If desired, multiple exceptions can be handled at the same time. In fact, we can create our own exceptions. In Python, these are classes that inherit from (and are subclasses of) the Exception class.

Error recovery can be managed by considering a few important things. First, check return values. Implementing a robust return value technique can provide a lot of information about the behavior of a program and aid in detecting and preventing errors. Second, use (and don't ignore) exceptions. Attempt to programmatically handle them. Of course, this means to try to anticipate all possible errors (which is a difficult thing to do). Lastly, include source code to attempt recovery. This may, for example, mean to loop to recover and retry an action that generated an exception. However, an exit strategy may be needed if the error continues to occur. This often happens with input/output (e.g., opening files, accepting user input, etc).

Testing and debugging
A large part of designing software is to ensure that it is free (or nearly free) of errors. This requires testing and debugging.

Definition: ***Testing*** *searches for the presence of errors. It merely indicates that errors exist.*

Applications are made up of many units. The preferred tactic is to test each unit separately. This is called unit testing.

Definition: ***Unit testing*** *is the process of testing each unit separately, in isolation from other units.*

Unit testing is performed often during development. The goal is to find errors and fix them early before they become overwhelming.

If ideal coupling and cohesion are maintained, it is easy to identify what a unit should do. This is called its **contract**. Programmers often call this the unit's **interface** (i.e., how it can be interacted with). Most programming languages provide this information via a signature (or header), such as a method's signature. To test units, we often look for potential contract violations. This can be done with positive and negative tests. That is, we test boundaries or boundary conditions (e.g., zero/empty parameters, one, full, etc). We provide values for arguments that aren't necessarily expected to see if we have properly dealt with such an eventuality.

Since testing is often repetitive and time-consuming, it can be automated. In fact, tests are often re-run, over and over again (called regression testing) after we have modified a program.

Definition: ***Regression testing*** *is the process of running tests after a change to the program to see if anything previously working has now broke. A previously working part of the code that is now broken is called a* ***regression****. A* ***test harness*** *is an entire class (or classes) whose sole purpose is to test programs in an automated way.*

Through testing, we can detect the presence of errors. Ultimately, however, we must find and fix them. This process is known as debugging.

Definition: ***Debugging*** *is the process of finding the source of an error.*

It is important (and sometimes quite frustrating) to note that errors can occur far (sometimes very, very far) from their source. Debugging helps to lessen the likelihood of errors and to improve the detection of errors.

That being said, it is important to develop good code reading skills and to play "computer." In fact, there are several techniques that can help with debugging.

Definition: ***Manual walkthroughs*** *involve getting away from the computer and running through the program by hand (on paper). This is a relatively underused method, but it can be very useful.*

An object's behavior is largely determined by its state. Therefore, incorrect behavior is often the result of incorrect state. Walkthroughs involve tabulating the values of key fields and documenting state changes after method calls. Sometimes, walkthroughs are delivered verbally to other programmers. Sometimes, a method known as rubber duck debugging is used.

Definition: ***Rubber duck debugging*** *involves talking to a "bobblehead" or a rubber duck (basically, some sort of inanimate object sitting on one's desk). Verbally explaining source code and the behavior of a program often helps to reveal errors and inconsistencies.*

A very popular approach when debugging is to use **print statements** placed in strategic places in the source code. This technique adds output statements throughout the code to show state, position, and sequence. Note that this approach can be overwhelming, as the output can be plentiful. However, the use of one or more debug Boolean variables can help with this. For example:

```
DEBUG = False
...
if (DEBUG):
    print("Something meaningful.")
```

Print statements enclosed in an appropriate if block will only be executed if the variable DEBUG is set to true. Sometimes, the need for varying levels of debugging output is needed. In this case, the variable DEBUG can be an integer rather than a Boolean. For example:

```
DEBUG = 3
...
if (DEBUG == 1):
    print("Debug level 1 output")
elif (DEBUG == 2):
    print("Debug level 2 output")
elif (DEBUG == 3):
    print("Debug level 3 output")
```

Most major software projects are designed using IDEs (Integrated Development Environments). You have used IDLE (for Python programs) and the Scratch IDE (for Scratch scripts). Many IDEs integrate a debugger as one part of its functionality (in addition to source code editing, syntax highlighting, compiling or interpreting, etc).

Definition: *A* ***debugger*** *is an interactive debugging tool that helps programmers test and debug their programs.*

Debuggers are language and environment specific, and they usually work together with the compiler. Debuggers support breakpoints (i.e., setting places in the source code to stop execution). In addition, they provide the ability to step into, through, and over one or more statements. This is useful to see how a single statement (or a group of statements) affects execution, and to identify the source of an error. Lastly, debuggers provide the ability to *watch* and *trace* variables as statements are executed. That is, a programmer can see the values of variables change as statements are executed. You can see why debuggers are very useful debugging tools.

What if your program is running unusually slow? If your code is doing unnecessary work or is just not optimized, but is producing correct output, a debugger might not be able to help. You need a profiler.

Definition: *A **profiler** is a tool that can show you the runtime and memory usage of your program. It can show you how long each function is taking to run, how much memory each part of your program is using, and more. It is mainly used to fix performance issues.*

So use a profiler for performance issues and use a debugger for bugs.

A few remaining important points, concepts, and ideas

Before leaving this overview of software engineering, there are a few final points to make concerning the development of software projects. Specifically, these refer to generating source code. First, code duplication is an indicator of bad design. Multiple instances of the same snippet of code scattered throughout a program makes maintenance harder and can lead to errors. To remedy this, we can encapsulate the duplicated code in a method that is called when it is needed.

Second, software engineers should practice responsibility driven design. This is the idea that each unit should be responsible for manipulating its own data, and that the unit that owns the data should be responsible for processing it. Intuitively, this leads to loose coupling and localizing change. If and when a change is needed, as few units as possible should be affected.

Third, much of successful software development involves constantly thinking ahead. When designing units, software engineers try to think about likely future changes. Doing so leads to design choices that make it easier to maintain units in the future.

Lastly, refactoring is an integral part of software development and maintaining coupling and cohesion. When units are maintained, new code is often added. This can lead to units becoming longer, responsible for more tasks, and more dependent on other units. Therefore, they may need to be split up into multiple units to maintain ideal coupling and cohesion. This is known as refactoring. Of course, units must then be tested (including regression testing).

Definition: ***Refactoring** is the process of modifying code without introducing new features. This often entails adding, modifying or even deleting units in an effort to make your code more maintainable (such as focusing on better coupling and cohesion).*

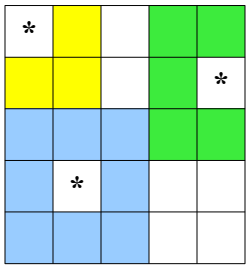
The game of life

The Game of Life was invented by John Conway in 1970. Generally speaking, it begins with a square 2D matrix of *cells*. Each cell either contains a living organism (or not), forming a population. We sometimes consider the matrix a virtual Petri dish for this reason. The goal is to see how the population evolves over time, according to a few simple rules:

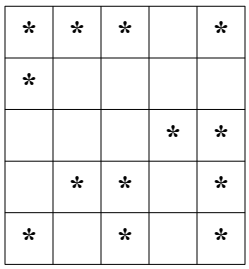
(1) Any living cell with fewer than two neighbors dies (ostensibly from loneliness);
(2) Any living cell with more than three neighbors dies (ostensibly from overcrowding);
(3) Any living cell with exactly two or three neighbors continues living, unchanged, to the next generation; and
(4) Any dead cell with exactly three neighbors becomes alive (ostensibly from reproduction).

A cell's neighbors are located above, beneath, on either side, and at the diagonals. Therefore, a cell may have up to eight neighbors. The exception, of course, is a cell that is on an edge or in the corner of the matrix. In the table below, the cells highlighted in yellow identify the three neighbors of the cell in the top-left corner of the matrix, the cells highlighted in green identify the five neighbors of the cell in the

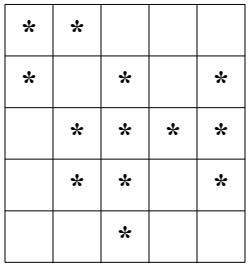
right-most column, and the cells highlighted in blue identify the eight neighbors of the cell near the bottom-left of the matrix:



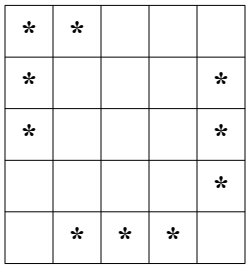
Initially, the matrix is randomly populated with living organisms and dead cells. This initial generation is called the **seed**. The rules are then applied for all cells simultaneously. That is, the number of living neighbors for each cell is calculated from the current generation in order to generate the next generation. Here is an example of the random seed of a 5x5 matrix, forming generation 0:



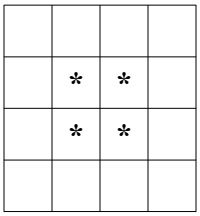
If the rules are applied simultaneously, then the next generation (generation 1) becomes the following:



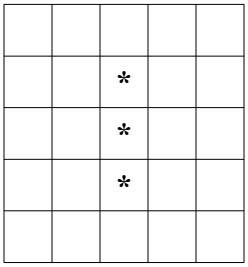
As one more example, the next generation (generation 2) becomes the following:

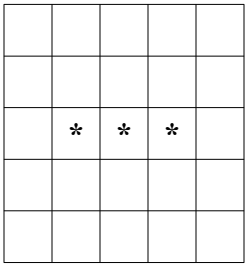


The game continues until either the generation stabilizes and stops changing (becomes a still life), flips between two states called oscillators, or expires into nothing (i.e., all cells are dead). Here's an example of a still life:



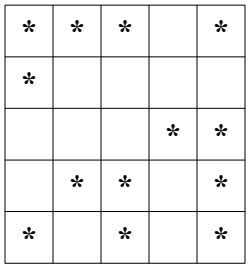
No matter how many more generations are produced, the matrix does not change. The following are examples of oscillators (that continuously flip from one to the next, from generation to generation):





Coding the game of life

The first issue to tackle is the data structure that will be used to represent the matrix. One obvious choice is to use a list of lists. That is, a list will be used to represent the matrix (or board). Each element of the list will be a list (i.e., a sublist) representing each row of the matrix. Here's an example of the following matrix represented as a Python list:



```
[['*', '*', '*', ' ', '*'],
 ['*', ' ', ' ', ' ', ' '],
 [' ', ' ', ' ', '*', '*'],
 [' ', '*', '*', ' ', '*'],
 ['*', ' ', '*', ' ', '*']]
```

Note that it has been formatted to appear somewhat like the matrix. Python actually represents it on a single line as follows:

```
[['*', '*', '*', ' ', '*'], ['*', ' ', ' ', ' ', ' '], [' ', ' ', ' ',
'*', '*'], [' ', '*', '*', ' ', '*'], ['*', ' ', '*', ' ', '*']]
```

The first iteration will be to accept the seed from standard input (which can be redirected by the operating system from a file). Additionally, we will just display the board (via a **print** of the list):

The remainder of this lesson has been purposely omitted. The goal is to follow along with your prof as the game of life is implemented in Python using iterative software development and other

software engineering principles. Although this is not a particularly difficult or large software project, it is enough to show how software is designed, step-by-step!