Week 07 Research Assignment

Note: All answers are a synthesis of what I've learned from the class materials, unless a source is linked specifically.

Prompt: Research the SOLID principles of Object-Oriented Programming as introduced by Robert Martin.

Response:

According to freeCodeCamp (https://www.freecodecamp.org/news/solid-principles-explained-in-plain-english/), the SOLID principles are a set of guidelines designed to help developers produce code that is easy to read/understand as well as easy to test: these principles help keep code "clean". Here they are one-by-one (all examples are my own synthesis of the information contained the previous link):

- The **S**ingle Responsibility principle states that all classes should be designed having only one purpose (as opposed to multiple purposes). An example: If a codebase is written to facilitate a business's transactions with customers (each of whom has an associated account), we would probably write a single class to encapsulate customer account data (such as names, addresses, and account numbers) and a single class that handles transactions. We wouldn't put the logic of a transaction in the customer class: then that class is probably trying to address more than one purpose. The single responsibility principle suggests that we keep something like records of a customer account and the logic of a transaction in their own classes for good reason: if something changes in how the company handles transactions, as long as we're following the single responsibility principle, *nothing about the customer class should change.*
- The **O**pen-Closed principle states that classes should be extensible but not modifiable (*open* to extension, *closed* to modification). By modifying existing classes to include new functionality, we take the risk of introducing bugs to code that is being used in production--or otherwise worked just fine (passed all necessary tests) before we modified it. If instead we *extend* the class with new functionality, the superclass can still do its job and will continue to pass all its tests. We can test and then modify our new class as much as we need until it is also ready to be used in production--at which point, hopefully, it will also be something we can *extend* instead of *modify.*
- The **L**iskov substitution principle states that if a class has subclasses, any code dealing with the primary class should be able to deal with any of its subclasses, and still work as intended. Essentially, an extending class shouldn't override the methods of its superclass to produce a different output. The Liskov Substitution principle seems to follow from the Open-Closed principle: if we use overrides in a sublcass, we're actually modifying the functionality, not extending it.

- The **I**nterface Segregation Princple states that any class implementing an interface shouldn't be forced to override methods that are irrelevant to that class specifically. Let's say you have two interfaces that both implement a parent interface. One interface is for regular customer accounts, and then there's a separate interface for customer accounts with extra privileges. You shouldn't put abstract methods to handle privileges in the parent interface because you'd then need to write override methods for them into the regular customer accounts implementation, but never use them. So instead, we'd extract our privilege handlers to a third interface, and have our customer account with privileges interface inherit from the original parent *and* the privilege handlers interface. Utilizing the polymorphic attributes of Java interfaces allows us to make our inheritance chain much easier to follow, and helps keep us to the original Single Responsibility principle.
- The **D**ependency Inversion Principle states that we shouldn't depend on plain classes as superclasses in our inheritance chain, and instead rely on abstract classes or interfaces as much as possible. This means we can extend an interface's implementing classes as much as we need to, and the original interface will remain unmodified. This principle is the means by which we follow the other four: if we use interfaces, holding to the other four principles becomes much simpler, and our code is easy to follow and we can test parts of it incrementally and separately from other parts of it without breaking them or fudging what they're meant to do.

Prompt: What are wildcards in MYSQL? How are they useful?

Response:

According to javatpoint.com (https://www.javatpoint.com/mysql-wildcards), there are two wildcards in MYSQL: `%` and `_` (a percent symbol and an underscore). Both of them rely on their position in a string to tell a mysql query what strings to match: "%Q" is not the same as "Q%".

`%`

is used to match any number of characters, using a LIKE clause. If we ran a statement such as `SELECT * FROM table WHERE column LIKE "Q%"`, it would return all rows where that column's value is Quentin, Quicksilver, or Qwertyuiop, but not quaestor, inQyAndQlyde, or McQuack.

`_`

is used to match a single character. If we ran a statement such as `SELECT * FROM table WHERE column LIKE "_e"`, it would return all rows where that column's value is Me, ye, or ee, but not she, Thee, or ewe.