# Chain of Responsibility

This pattern does a good job of encapsulating logic and decoupling the concerns of different handlers. Since each handler is a separate class, testability and as a result maintainability are made easier. It’s just much easier to test components in isolation. I think this would be the best choice for handling choice inside a library and you want to allow the user to custom select the handlers they deem appropriate. This would allow a user to effectively construct their own chains, given access to a variety of functionally unique links. That also means that extendibility and flexibility are easily managed, as you can create as many link implementations as necessary and plug and play the various components as necessary. I do think the biggest potential drawback to this pattern is the concern with function calls. For a logically intensive chain with multiple potential sidechains, it is possible that it will take many method calls to eventually reach the correct handler. In a modern computing environment, you are likely to encounter an interrupt and context switch eventually, and that would be significantly more costly in this pattern than the others. Ultimately, I think this approach succeeds in loosely coupling handler logic while providing a flexible design, but it may have drawbacks with regards to computational efficiency, readability and familiarity.

# Dispatch Table

Of the implementations for this assignment, this is my favorite. I think the pattern is very clean both in the setup and execution. The code is maintainable and extendable, because adding another case is as easy as adding another entry to the dictionary. The ability to use some sort of enumeration as the keys also provides consistency throughout the program, as well as contributing to the readability of the code, as you can use actual event descriptions for cases. This was also substantially less code than the chain of responsibility, and since you go directly to the handler instead of stepping down through a series of possibilities, is also significantly less computationally expensive. One limitation is that dictionary misses must be handled in a try/catch, as there is no standard default action as with the other two implementations, but this could be potentially mitigated by wrapping the dictionary in a method encapsulates the lookup and the try/catch, such that passing the event code to the method handles all scenarios.

# Switch Statement (If-Else)

As the most familiar method, I don’t think this is a bad implementation, but it does have limitations. Maintainability and extendibility are not much of a concern, as adding additional cases is trivial, but given the format, adding cases may be more prone to copy/paste errors and poor ordering logic than other patterns. It is easily the most concise in terms of lines of code, and if encapsulated in a method, could be implemented in other files as easily as the other patterns. As far as clarity, this has a bit more syntax than a dispatch table but given the widespread familiarity with switch and if/else statements, I believe that concern is mitigated slightly, while conceding that as the logic grows more complicated, this becomes a less appealing pattern.