Jacob Gillenwater

Dr. Phil Pfeiffer

CSCI-5300-201

March 15, 2020

HW5: Chain of Responsibility

Chain of Responsibility

A chain of responsibility's primary strength is its flexibility. Through a chain of responsibility, a third-party developer can simply inject a custom handler for an event into what is essentially a linked list. However, this high degree of flexibility comes at the cost of clarity, concision, and performance.

For example, a chain of responsibility requires that each handler in the chain be wrapped inside of a unique class. Even while using inheritance, this approach requires a lot of boilerplate code to handle the basic setup of a new handler. Additionally, it is usually to be best practice to place each class into a unique file. Therefore, any chain of a reasonable length dramatically increases the number of files in the project, obscuring the clarity of the logic. In short, even when following otherwise good design principles, a chain of responsibility forces the implementor to violate the DRY principle through an abundance of boilerplate code, and it obscures the logic across multiple files.

As a side note, a chain of responsibility is inherently inefficient. Upon the initial setup of a chain, a computer must spend a significant amount of CPU cycles initializing each object and allocating memory for each of these new objects. Furthermore, whenever a request is sent down the chain, each call between the functions will quickly fill up the call stack. This situation can be very problematic for environments with limited memory space, or in any situation where a context switch is likely to take place during the processing of a request.

Chain of Responsibility is a design pattern proposed by the Gang of Four in their landmark book *Design Patterns*, with the express goal of promoting Object-Oriented Design. Through this lens, the origin of the pattern is clear, as it is the most object-oriented and SOLID method for managing choice in programs. Unfortunately, when viewed with a lens offering any perspective of practicality instead of only paradigm, the chain of responsibility pattern provides no benefit that cannot be achieved more easily through an alternative design pattern.

Dispatch Table

A dispatch table is everything that a design pattern should be: easy to understand, easy to implement, flexible, and concise. The pattern allows flexibility by simply allowing third-party developers access to the dispatch table. The access allows the developer the ability to not only add new actions but override existing actions. Also, should either addition or overriding of actions be unwanted, one can easily revoke them via a wrapper class.

The dispatch table requires no boilerplate code, as each line is necessary to perform the action, which follows the DRY principle. Furthermore, each key and action pair define your logic into a single reference, keeping the code concise. The only failure is a lack of all-encompassing clarity. Barring some

interpreted languages, the implementation separates each action from the rest of the dispatch table. This problem is not unique to dispatch tables, as a chain of responsibility, and a switch case can easily run into the same situation. However, a dispatch table typically forces this approach. In conclusion, a dispatch table is a strong all-rounder, offering a high degree of flexibility, conciseness, and limited obscurity.

Switch Case / Nested If

A traditional switch case is the most rigid of these three design patterns. It offers no support to third-party developers without exposing all other code within the namespace. What the pattern lacks in flexibly, it more than makes up for in clarity. If statements and switch cases are ubiquitous tools within a programmer's arsenal, so reading and understanding the flow and logic of a switch case or a nested if is trivial for an experienced programmer. Because of its ubiquitous nature, the clarity of this design pattern is easily the highest of the three being examined.

This pattern is also potentially the most concise of three patterns examined, because no code is wasted developing wrapper classes, or initializing data structures. The logic and the associated actions are presented precisely where they are needed, and nowhere else. Depending on the complexity of the action, it may become necessary to abstract its implementation out into a separate method, as the dispatch table requires. However, in this situation, it becomes no more concise than a dispatch table. In summation, this pattern is the clearest, besides also being the least flexible. Additionally, this pattern is at least as concise as a dispatch table, both of which being more concise than a dispatch table.