Team Subway
CSC 403 – Software Engineering
10/15/2023
Team Members: Will Shepherd, Eli Payton, John Parks, and Chris Perry

Chris

Name: State
Category: Behavioral
Description:
The idea behind the State behavioral programming pattern is to have an object whose behavior changes depending on its current state. The state can be described as a status that determines how an object performs an action that it's being asked to do.

For example, consider the play/pause button on a music-playing application to be an object and one of its actions is to display an icon for the button. In this scenario, a play or pause button icon is displayed, depending on whether the music is playing or is paused, so it could have the status options: "musicPlaying" or "musicPaused". If the status is currently "musicPlaying", it'll show the pause button, if the state is currently "musicPaused", it'll show a play button.

An implementation of the pattern would have a Context class, a State interface, and a number of ConcreteState classes implementing the State interface. The Context class acts as a way for a user to interact with the state object, the State interface acts as a template to standardize how to build the different ConcreteStates, and the ConcreteState classes are implementations of that template that define the specific behavior of actions for each state.
If we were to implement the play/pause button scenario, the Context could be a box containing the pause/play button that users click and define what state it needs to switch to when a user clicks it, the State could be a template that includes an action, "DisplayIcon()", and the ConcreteStates could be "Playing" and "Paused", where the "Playing" state's "DisplayIcon()" implementation displays the pause icon, and the "Paused" state's "DisplayIcon()" implementation displays the play icon.lic

When to Use:
When you have an object whose behavior needs to change dynamically depending on a state, especially when the alternative is a large collection of conditional statements that can be divided in a modular way

Advantages:
Overall:

- Since each state is implemented in its own subclass, it can be easier to read

If the state transition logic lives inside of each ConcreteState subclass:

- The logic becomes modular and it's easier to add new states since the logic determining the transition patterns for a state is isolated to the subclass
- The transition logic also becomes easier to read since it's partitioned out, where the alternative could be a large switch statement or something similar that becomes confusing very quickly

Disadvantages:

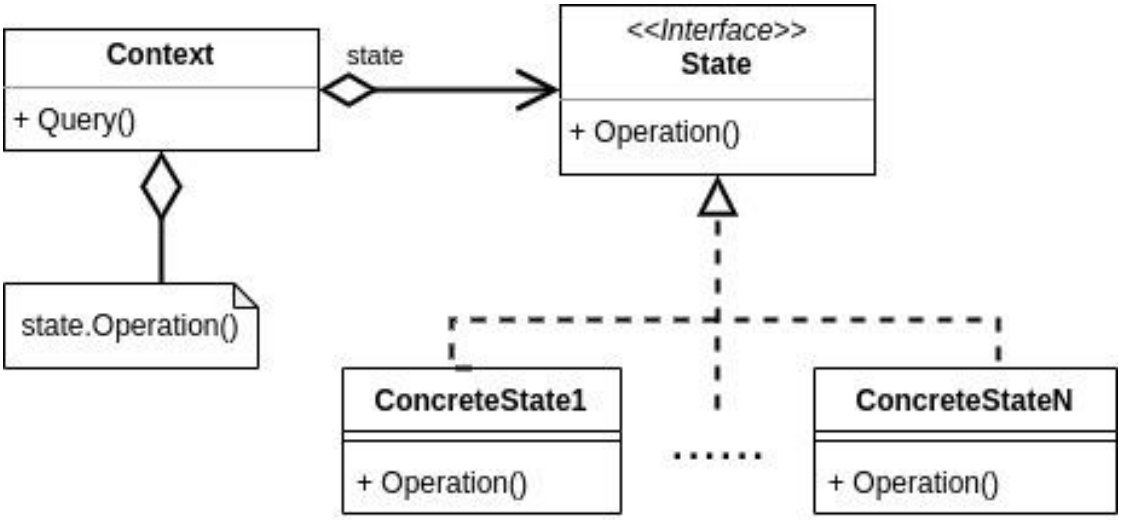
Overall:

- Since each state is implemented in its own subclass, the code is less compact and can be harder to keep track of

If the state transition logic lives inside of each ConcreteState subclass:

- The state subclasses will be dependent on each other since they'll need rely on each other to determine the current state

General UML:



Eli

Name: Proxy

Category: Structural

Description:

The idea behind the Proxy design pattern is to control access to an object by using a proxy object. The user can only access the obfuscated object via the proxy. This provides a layer of security to the object.

In my example implementation, the proxy is a "CreditCard" object that has access to the proxied "BankAccount". The user cannot directly access the bank account's money. But the credit card can spend money via the bank account and checks the credit card pin before executing the spend.

When to Use:

When you have an object whose access needs to be controlled for security reasons/monitoring. Proxies allow caching and can access remote resources like authentication services without affecting the requested data. Proxies can also hide the complicated interfaces of the objects they access.

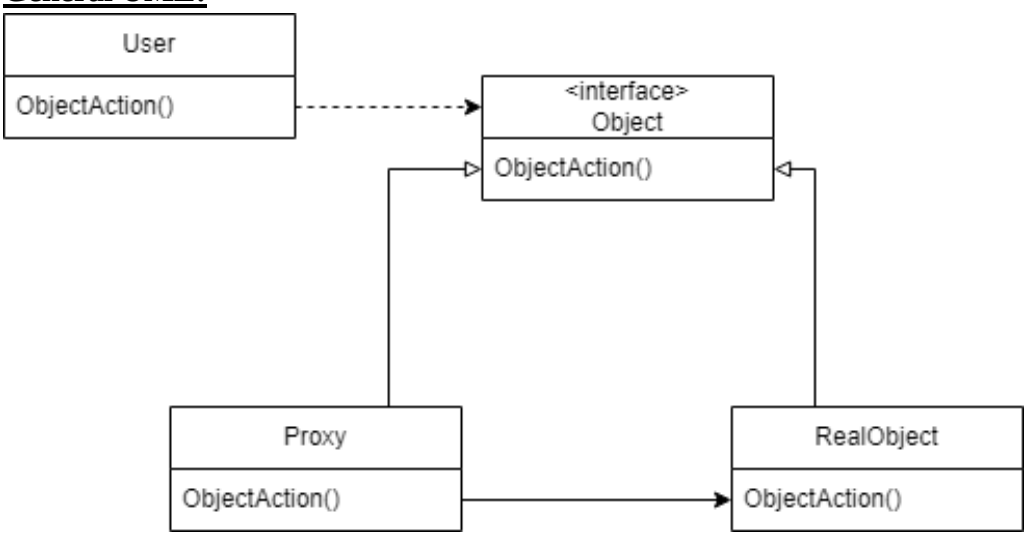
Advantages:

- Security - A proxy can allow you to control access to an object by checking permissions of the requesting user before allowing access.
- Resource Management - Proxies can delay creation of real objects until needed also known as lazy loading.
- Caching - Proxies can store data from previous requests allowing the system to save time when hit with repeated requests.

Disadvantages:

- Complexity - Introducing a proxy can add complexity to a system, especially when multiple layers are being used.
- Increased response times - Navigating through multiple proxy layers for a request can add latency.
- Coupling - Using a proxy can lead to tight coupling between the proxy and the object it protects meaning the proxy might need changes when the object is changed.

General UML:



John

Name:
Iterator

Category:
Behavioral Pattern

Description:
The Iterator Pattern is sort of a "middleman" client between an object accepting inputs and the inputs themselves. For example, if there was a large collection of objects in a collection, but we don't want to show the implementation on how we use them or care about their types, we can simply iterate through them using this pattern. Another example is if we want to generate a **massive** list of numbers and calculate which numbers are prime and which aren't. The iterator pattern allows us to see immediate results for the beginning numbers, rather than wait for the entire collection calculation to finish to see the results. Many programming languages have a native interpretation built in for this pattern. For C#, their built-in methods are the enumerator and yield.

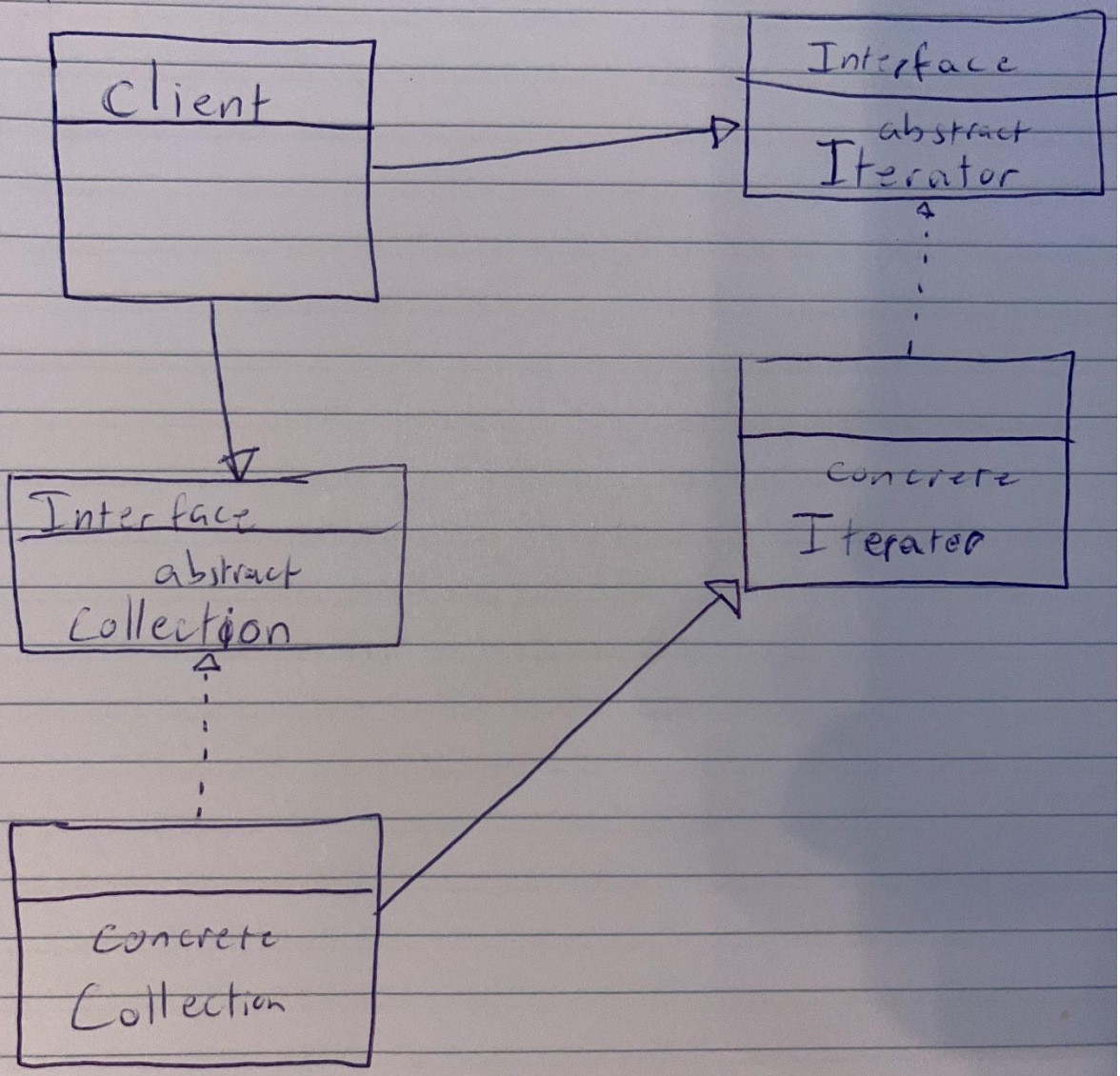
When to Use:
Let's say we wanted to iterate through the different results if we took a list and used the elements to transform a variable. We could iterate through the list and do a yield return to show what each element does to our variable, one at a time.

Advantages:
The iterator pattern has many advantages, such as the ability to allow for more abstract code. It is also flexible, in the sense that it can have different types of iterations as well. It is also very useful for lazy loading, which is useful in web applications.

Disadvantages:
If you wanted to know and show the underlying interpretation of your code, and have it more compact, the iterator pattern is the opposite of what you may need. This would also be suboptimal and unnecessary if we used simple collections. There is also added work and more code necessary to write for this pattern.

General UML:

Iterator Pattern



Will

Name: Bridge
Category: Structural
Description:

The bridge is a structural design pattern that allows one to decouple an abstraction or a high-level interface, which is less complicated and detailed than its implementations but nonetheless lays a framework that describes how they should be or work, from its implementations. This means that implementations of that abstraction are able to vary independently, which is particularly useful whenever there is a need to change or extend an implementation without affecting other implementations of the same abstraction. Furthermore, by encapsulating code in an abstraction, it provides a way to more easily manage complex hierarchies of code or different variations of code as implementations of that abstraction. In this way, the abstraction serves almost like a template and its implementations are fleshed out versions of that template; you can think of a form that is filled out by different people – all are of the same abstraction, but their details vary.

A good example of the bridging design pattern in use can be seen in how chargers have evolved. Before, chargers were mostly a specific cable fixed to a specific power adapter, usually designed exclusively for just one device. Since then, though, we've begun to see chargers as separated charging cables and power adapters. They all work more or less in the same fashion, connecting one USB-2.0, USB-3.0, USB-C, or similar connective interface to the power adapter, but their implementations vary; some charging cables may support only USB-C to USB-C, while others may support only micro-USB to USB-2.0. This allows a variety of cables to be bridged to a wide range of devices while still adhering to the abstraction that is a charger; overall, this has increased the versatility and functionality of chargers. However, note that for this example, you would likely create multiple abstractions, one for charging cables, power adapters, and chargers, which includes a combination of a compatible charging cable and power adapter.

When to Use:

You should use the bridging structural design pattern whenever you want to split a large class or a set of closely related classes into two separate hierarchies: abstraction and implementation. This allows the abstraction and implementation to be developed independently of each other.

Advantages:

One obvious advantage to bridging is that – as it was previously mentioned – it allows for the decoupling of abstractions from their implementations; this makes it easier to make changes to one implementation without affecting others. It also increases flexibility and extensibility; more abstractions and implementations can be made to accommodate evolving needs or requirements. Similarly, abstractions can be reused as new implementations that do not already

exist or repurposed as new abstractions for eventual implementations. Furthermore, it improves the maintainability of code; in most cases, a problem lies in either the implementation of its abstraction, and bridging makes it easier to identify which may be the culprit long-term.

Disadvantages:

Bridging also has its disadvantages. One such disadvantage is that it may add a layer of complexity to a project; it may not always be easy to identify which abstractions are needed, or it may not even be feasible to create abstractions, especially for smaller projects, wherein there is not a need for many different similar implementations. It also requires more forethought and technical overhead long-term, especially if multiple abstractions are being used in any one implementation. Finally, it can – over time – contribute to an excessive volume of code, which can be tedious for developers to both maintain and keep track of.

General UML: