Structural Patterns

Tan Cher Wah
cherwah@nus.edu.sg

Structural Patterns

Adapter	Allows classes with incompatible interfaces to work together.
Decorator	Attaches additional responsibilities to an object. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Provides a simplified interface to a complex subsystem.
Flyweight	Reduces the cost of creating and manipulating a large number of similar objects.
Proxy	Acts as an intermediary between a client and the real object.

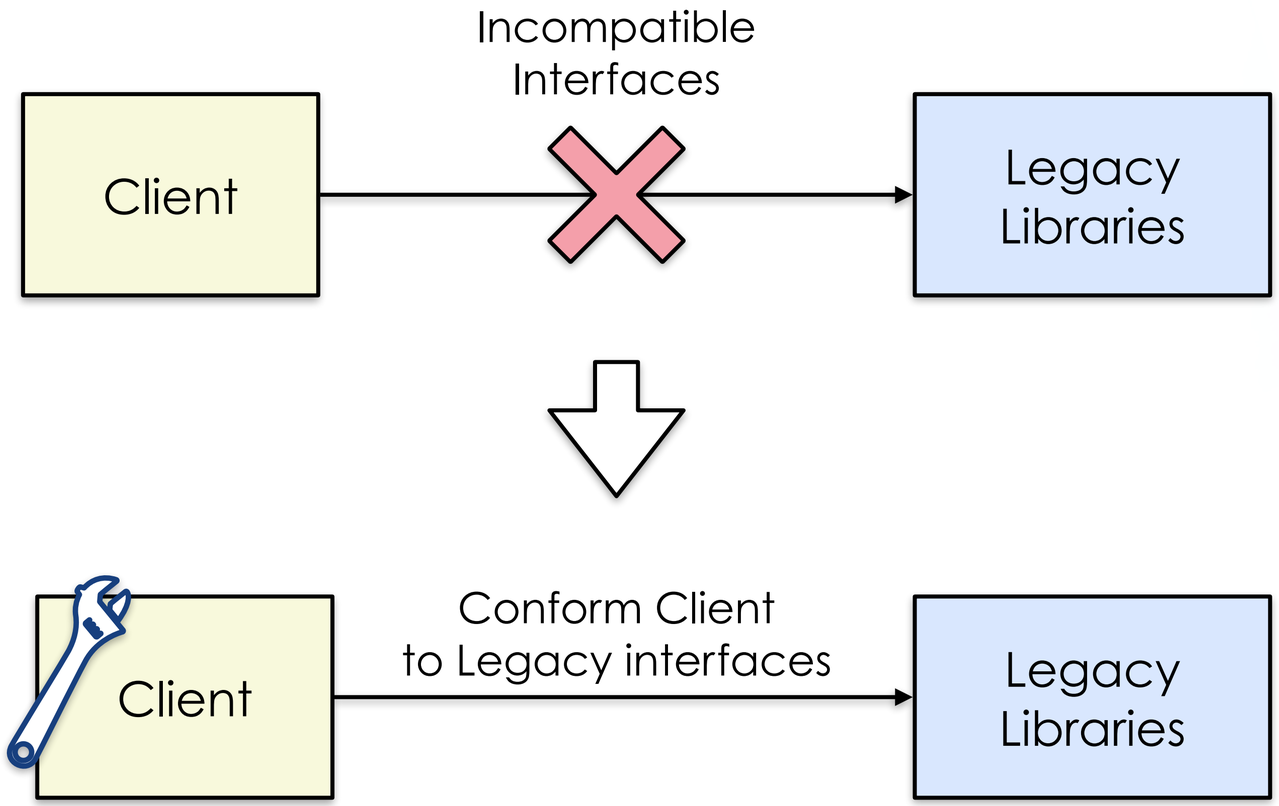
Adapter

A Design Challenge

- Consider the scenario where we need to use multiple pre-existing classes or libraries that have incompatible interfaces or methods.
- Often, it is difficult or impossible to integrate them and make them work together.
- A solution is to create an intermediary that translates or maps the interface of the client to the interface expected by the pre-existing classes or libraries.
- This allows the client to interact with the adapted, pre-existing classes and libraries, and allows for code reuse.

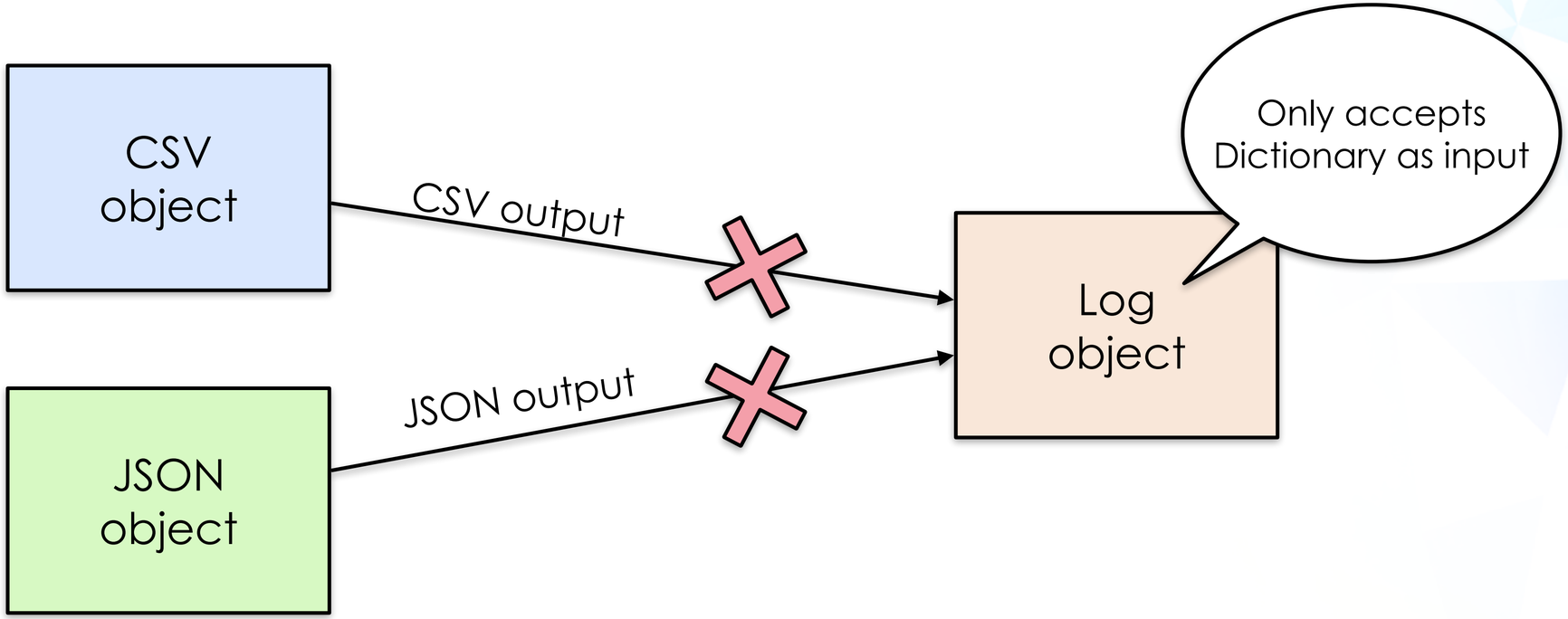
A Lesser Approach

The simplest way is to modify existing Classes to talk to legacy libraries



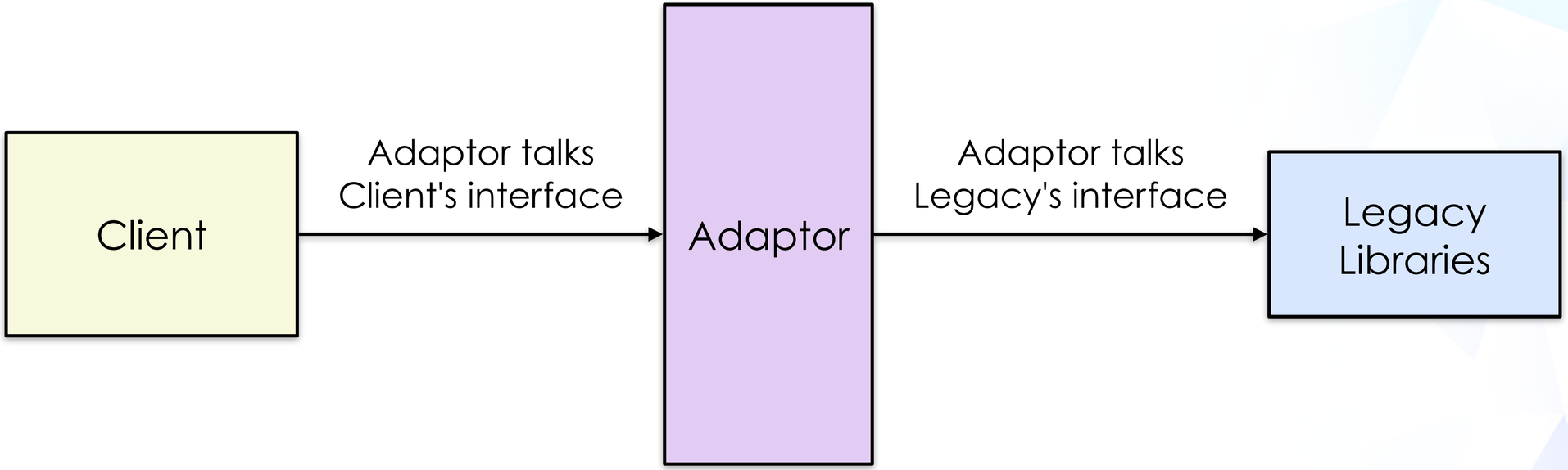
Problem

Given two objects that produce outputs in distinct formats, CSV and JSON, respectively, how can we utilise an existing Log() function to log their results?



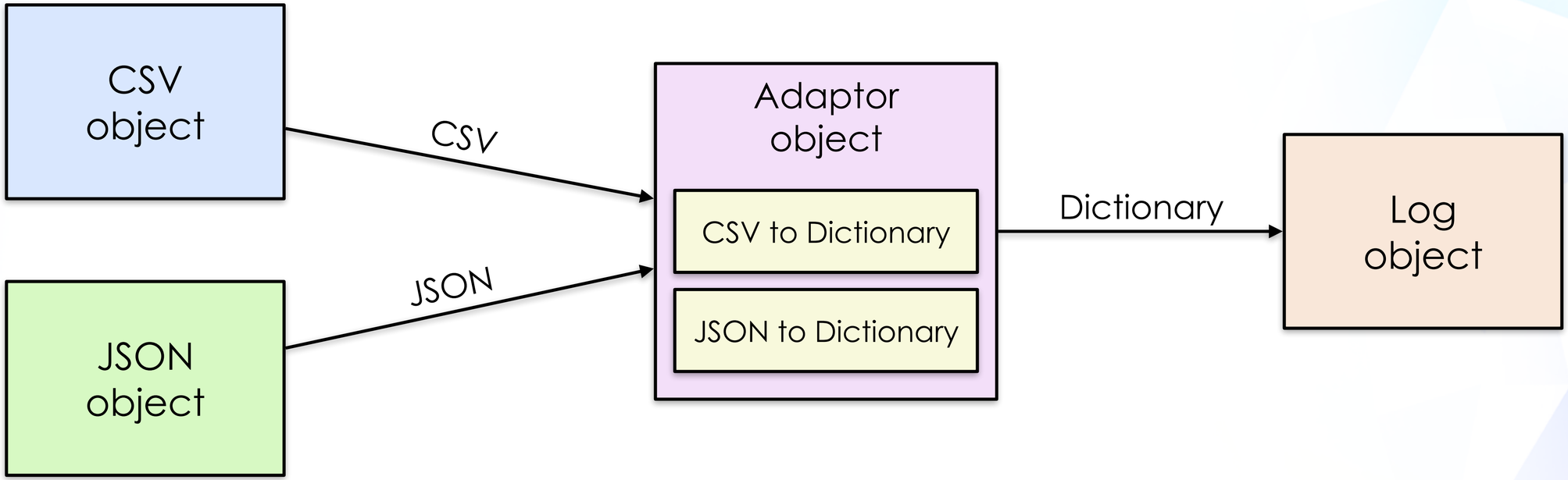
A Better Approach

Provide an Adapter object to adapt differences between interfaces



Problem Solved

Provide an Adapter to convert the CSV and JSON to a Dictionary as inputs to the Log() function



Go Implementation

- Review the Go implementation of the Adapter design pattern located in the Adapter folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code in order to determine
  - The number of Adapters in the program
  - The adaptations that were done
  - The function that can be reused due to the adaption

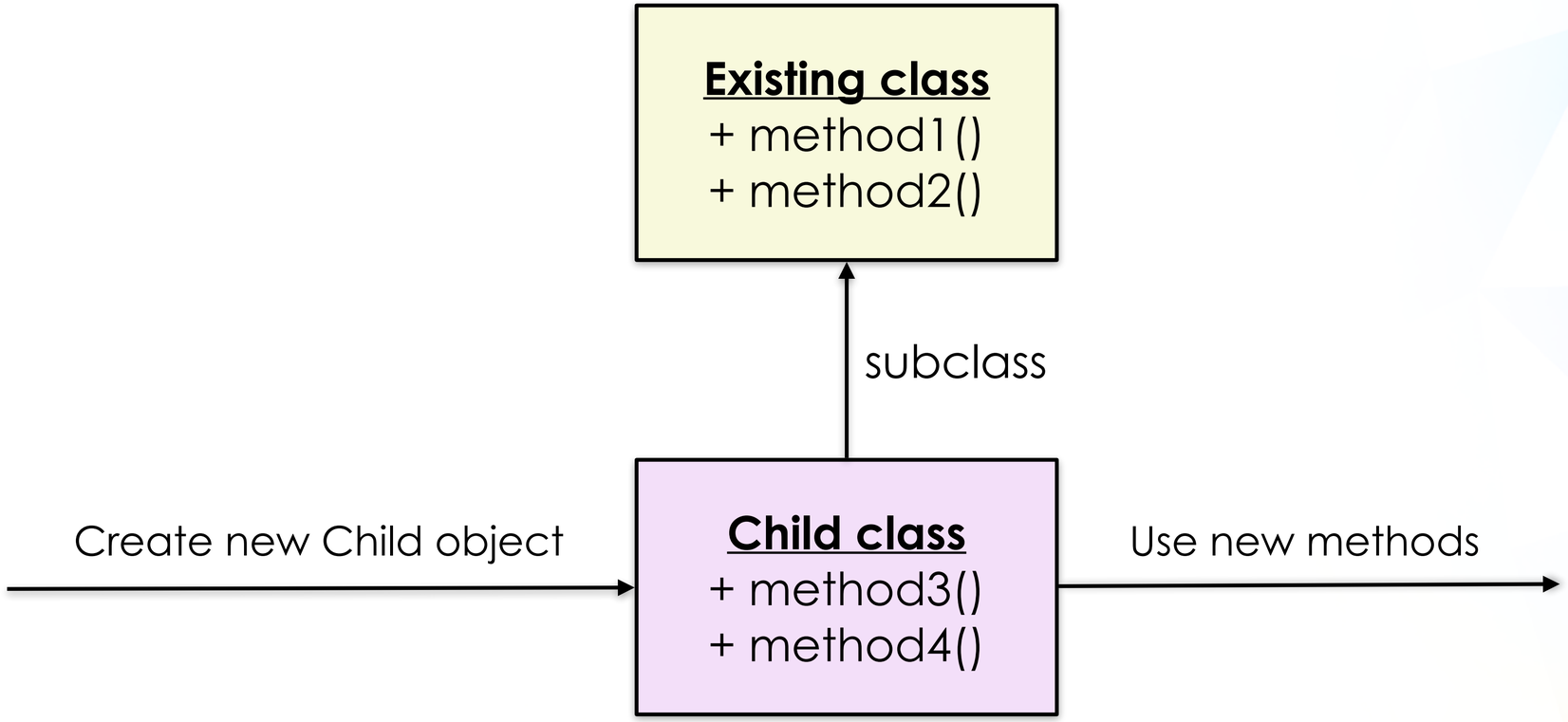
Decorator

A Design Challenge

- Consider a scenario where there is a need to add new functionality to an object without modifying its class.
- Instead of creating a new sub-class for each new functionality, as traditional inheritance would require, we need a way to add the new behaviour to the existing object.
- This provides a more flexible and maintainable approach, as the existing object's class remains unchanged and new functionality can be added or removed at runtime without affecting the rest of the system.

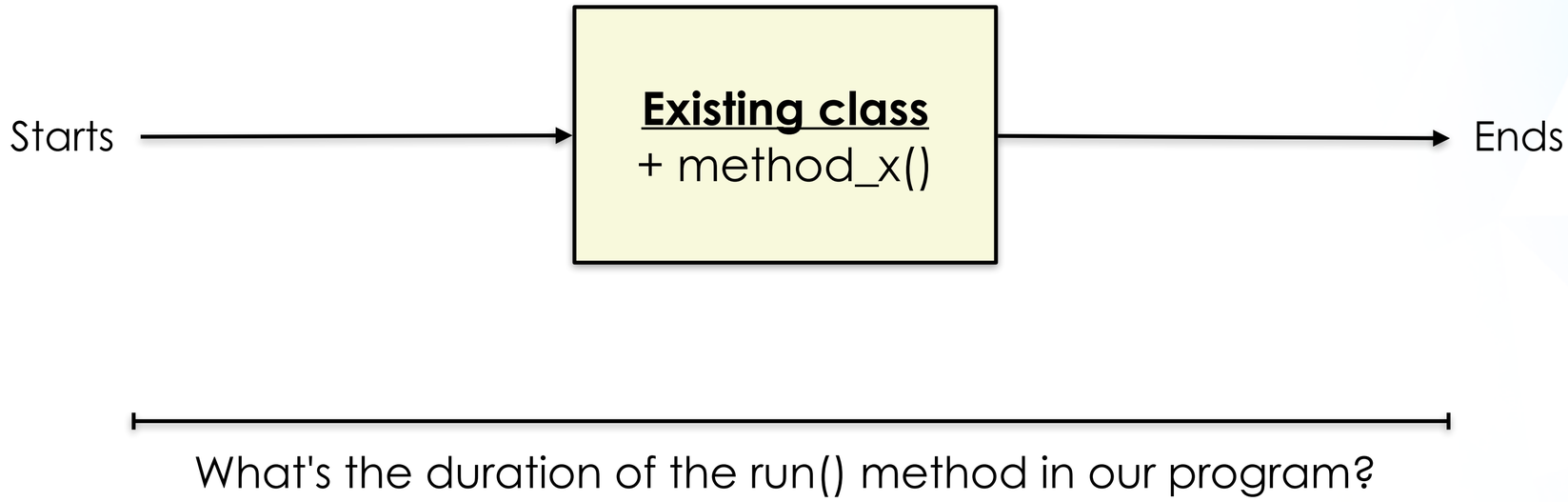
A Lesser Approach

Subclass the Existing class to add new methods into the more specialised Child class, thereby adding new functionalities to the overall system



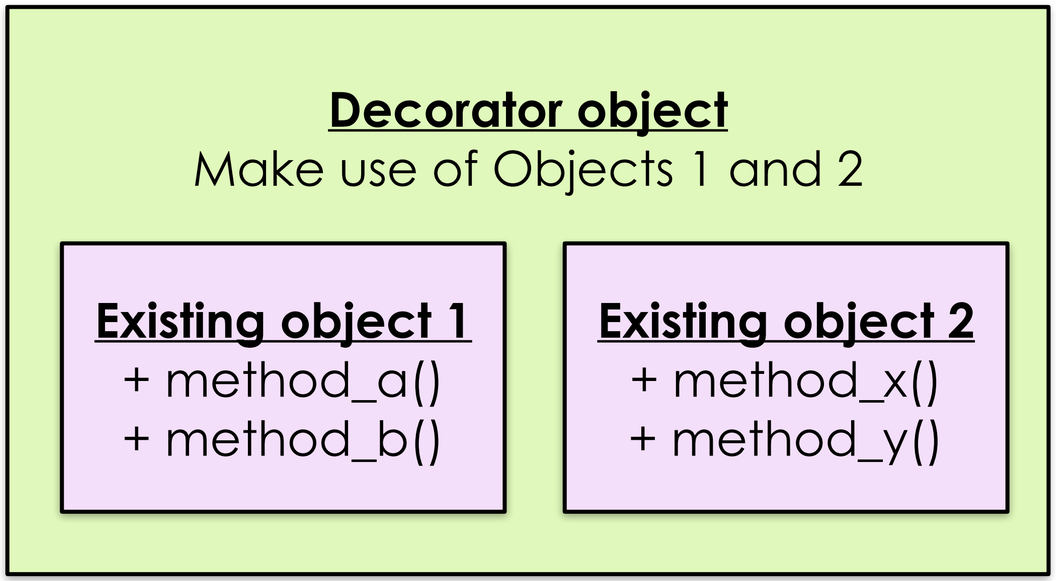
Problem

Given an object, how can we measure its elapsed time without modifying it?



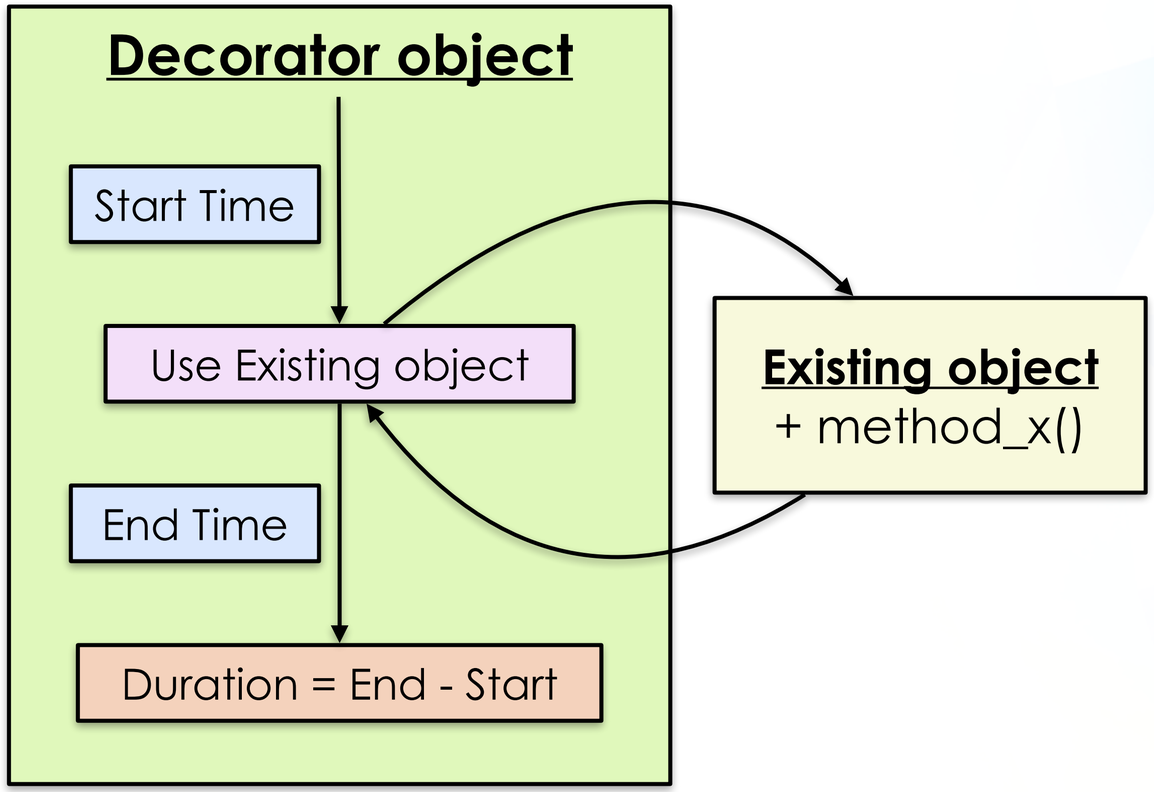
A Better Approach

Take a compositional approach by wrapping the original object. Then forward requests to it, and add your own behaviour before and/or after the request is handled.



Problem Solved

Wrap the object within a Decorator object that captures the start-time and end-time of the object's methods



Go Implementation

- Review the Go implementation of the Decorator design pattern located in the Decorator folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code in order to determine
  - The object that was decorated in the program
  - The new functionality that was added to that object

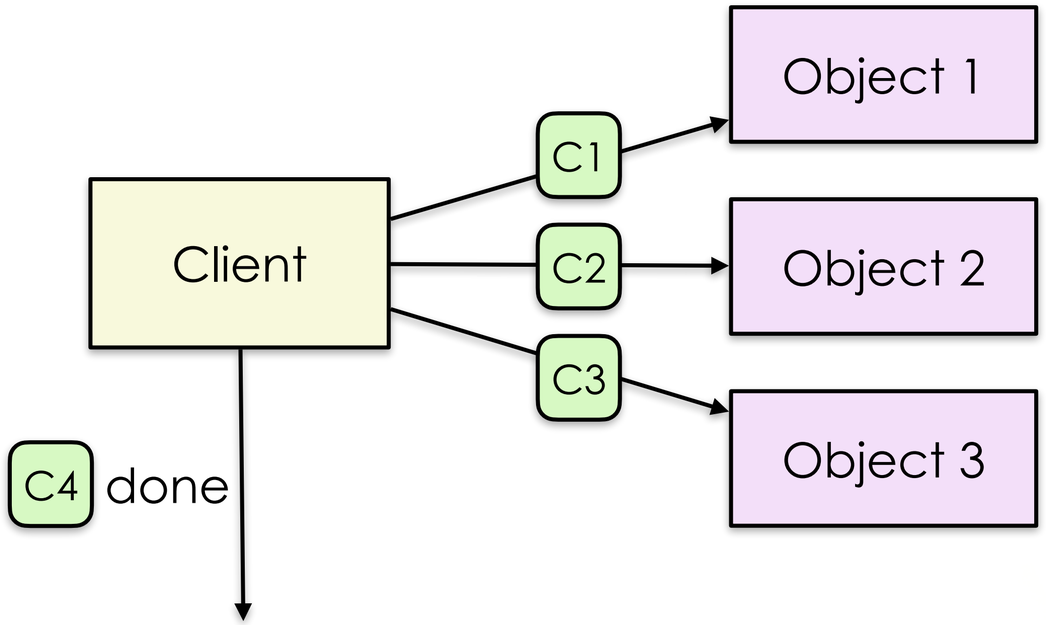
Facade

A Design Challenge

- Consider a sub-system with complex components and APIs, which can make it difficult for the client to understand and use the system effectively.
- To address this problem, a solution would be to provide a simplified interface to the complex system, hiding its internal complexity and providing a clear and easy-to-use API for the client.
- This reduces the cognitive load on the client and making it easier for them to understand and use the system.

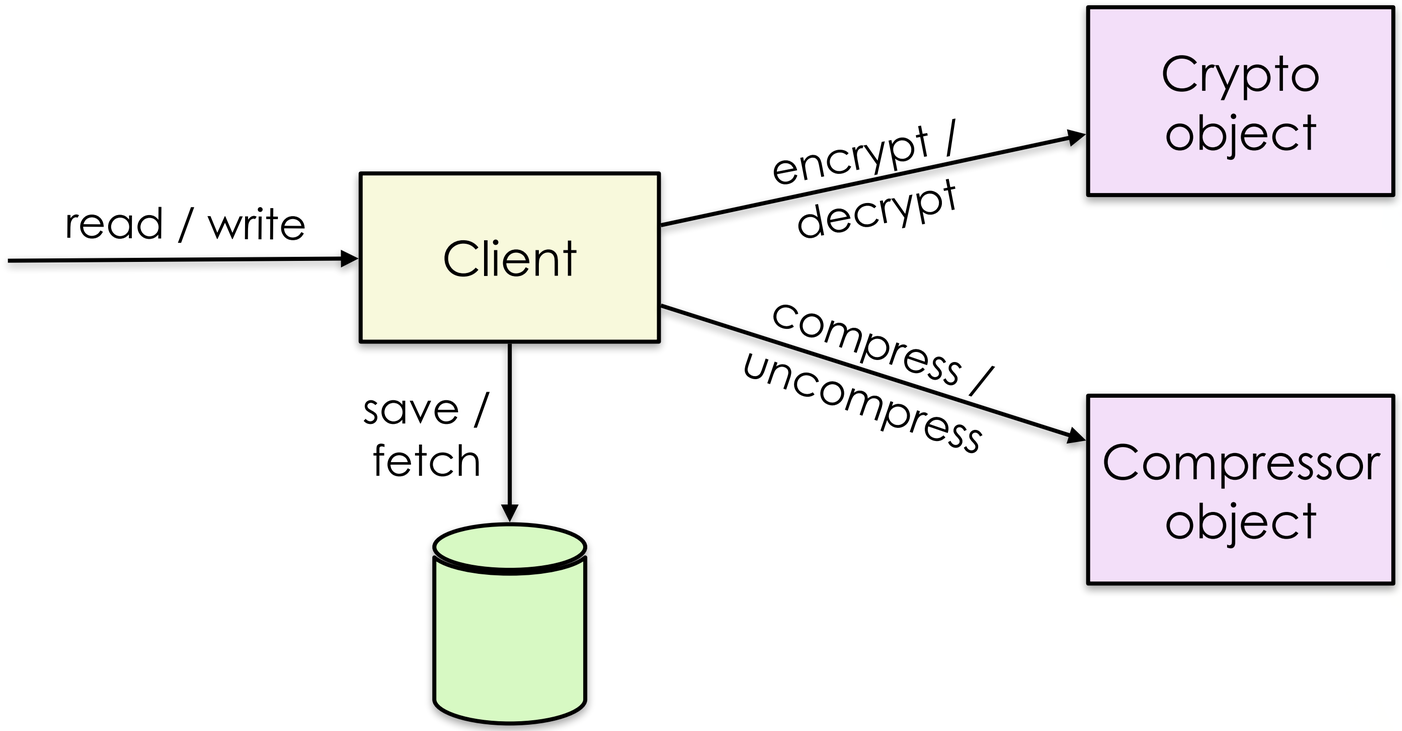
A Lesser Approach

Expose the various low-level components of the system to the Client; making it difficult for the Client to perform a straight-forward task



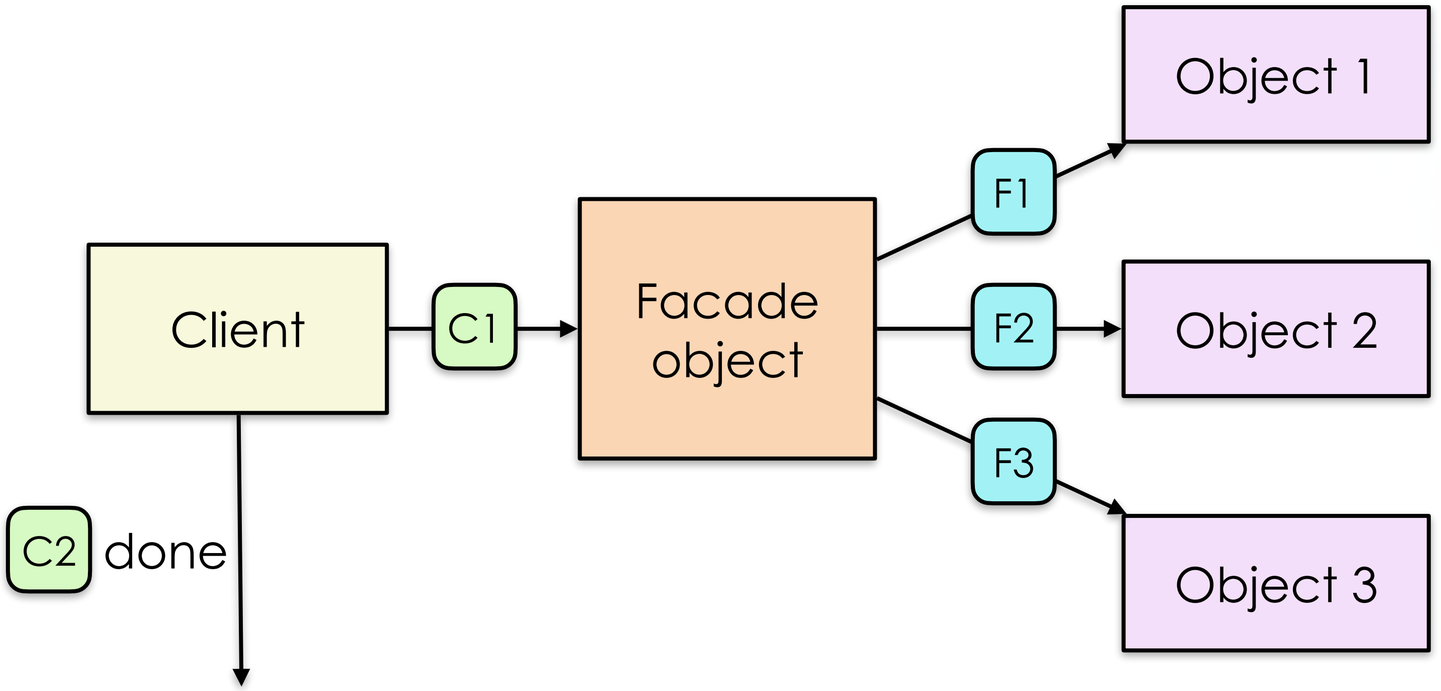
Problem

Consider a system requirement where content to be saved must first be encrypted, then compressed before committed to storage. And the reverse operations to be done for retrieval.



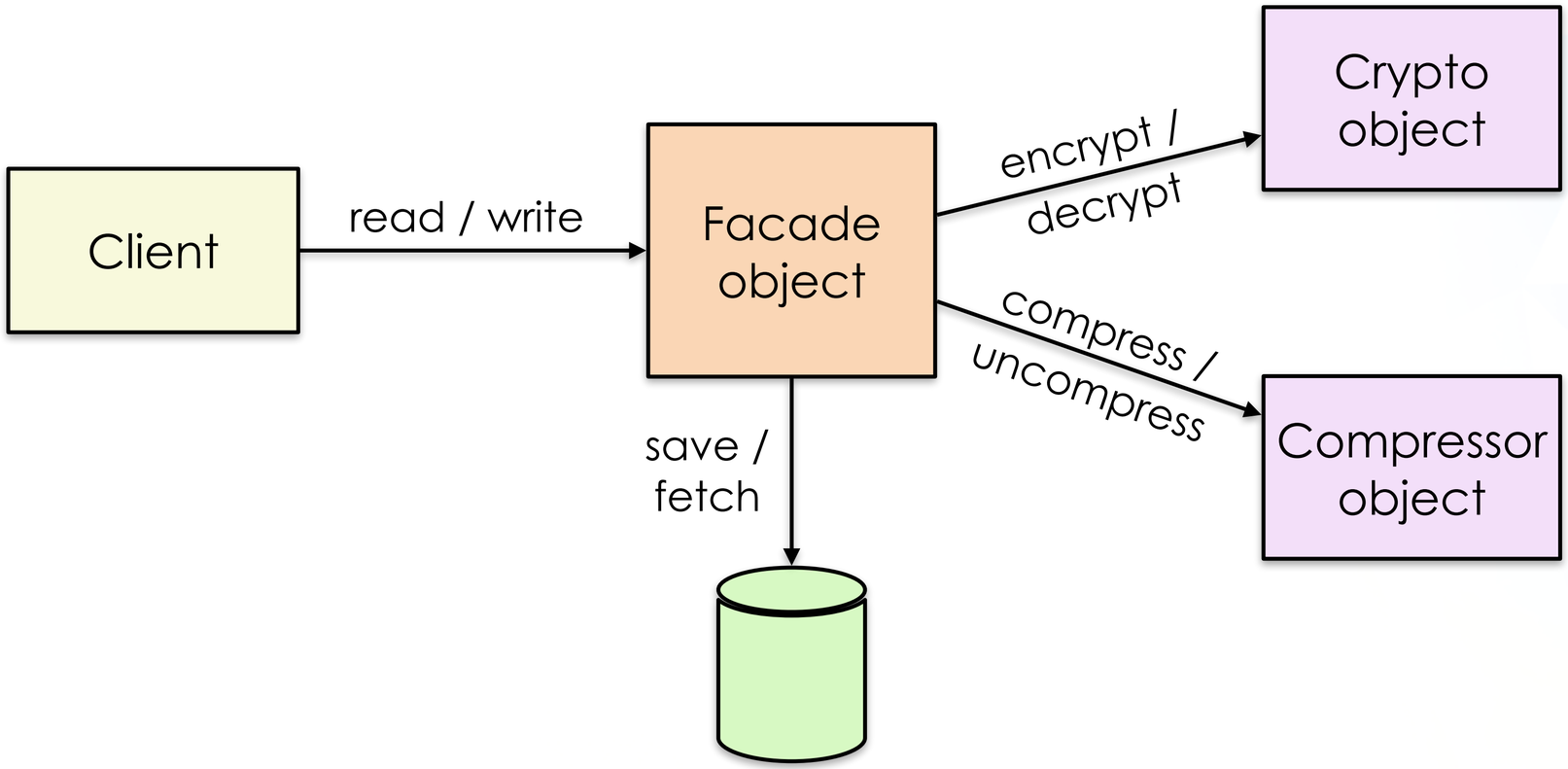
A Better Approach

Hide the complexity by providing a simpler interface, and invokes the various the low-level components on behalf of the Client



Problem Solved

Provide a simpler interface / API to the Client to shield it from the need to interact with the Crypto and Compressor objects.



Go Implementation

- Review the Go implementation of the Facade design pattern located in the Facade folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code in order to determine
  - The simplified interface offered to the client
  - The Encrypt and Decrypt logic used
  - The Compress and Decompress logic used

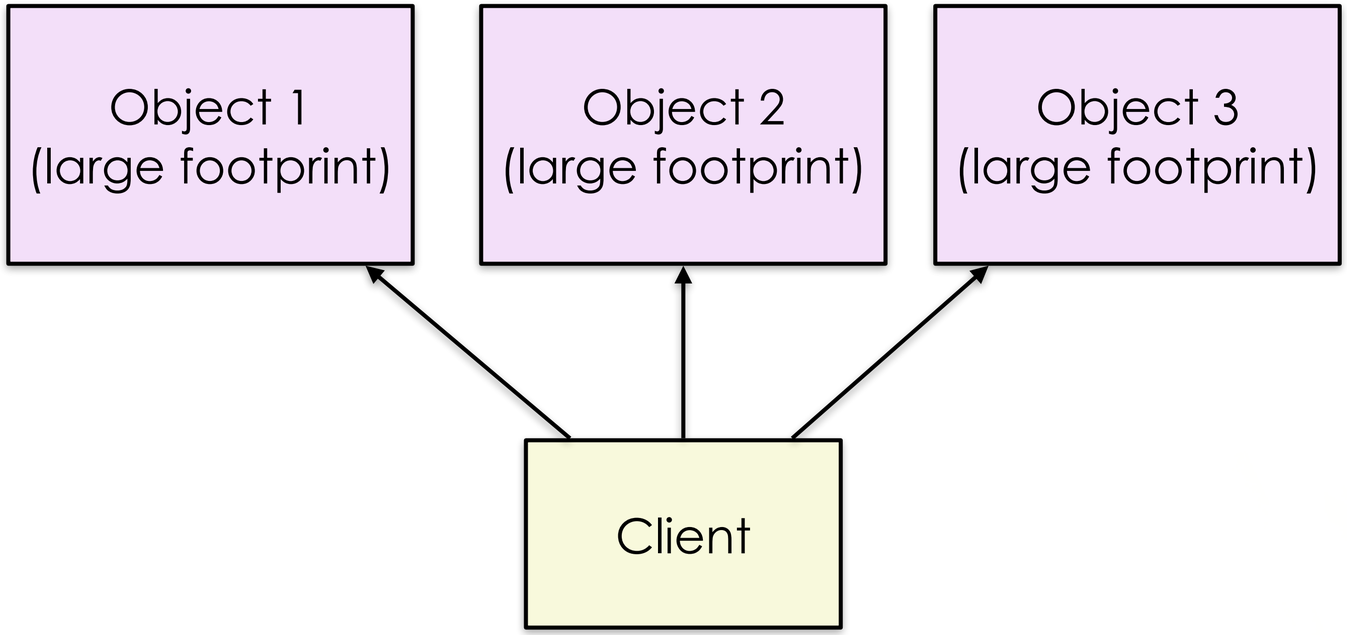
Flyweight

A Design Challenge

- Consider a system that needs to manage a large number of similar objects
- However, creating and storing each object can be costly in terms of memory usage and processing time.
- To address this problem, a solution would be to reuse existing objects instead of creating new ones, and storing the unique data separately
  - This allows for a significant reduction in the memory usage and creation costs, leading to a more efficient management of the similar objects.
  - It also enables the system to handle a large number of objects while maintaining its performance and responsiveness.

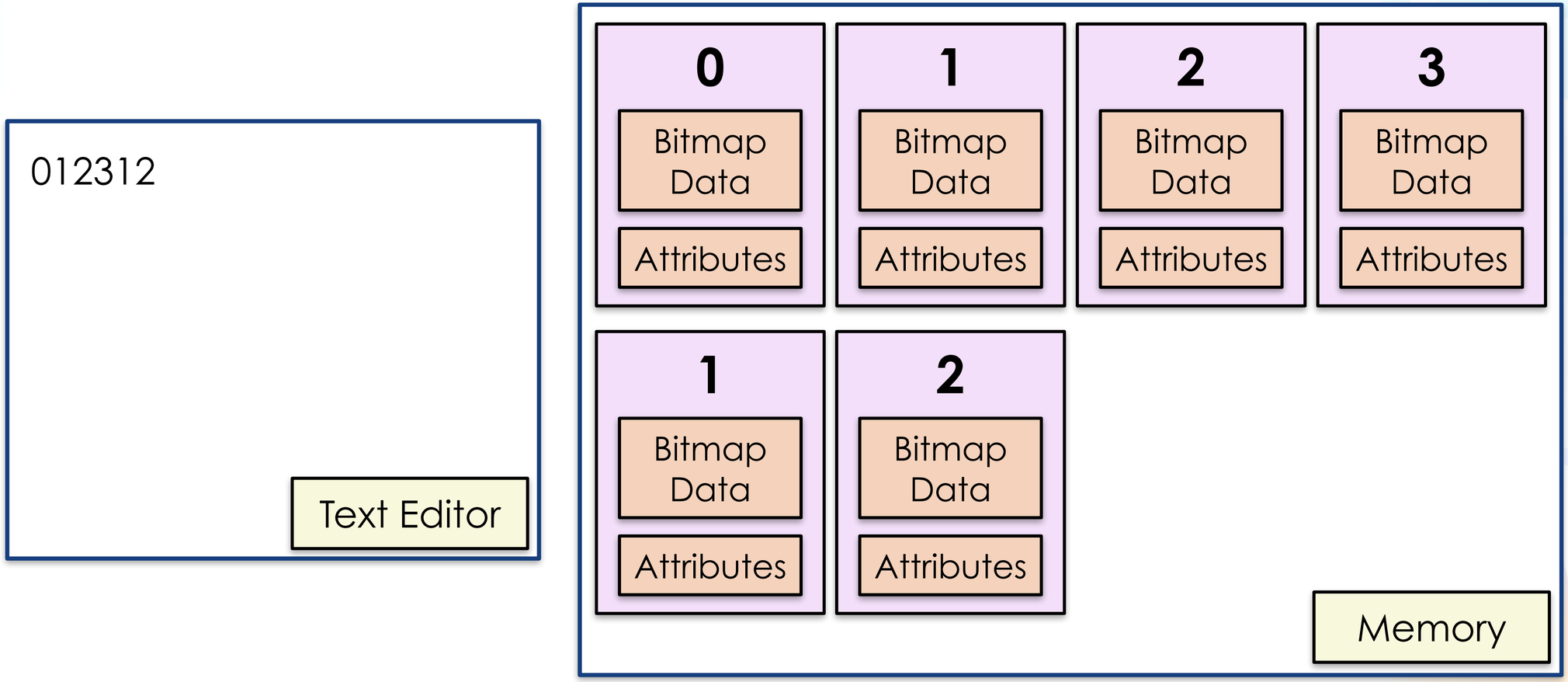
A Lesser Approach

Each object contains a large and complex data structure for itself; thus increasing the memory usage significantly each time the Client instantiates the object



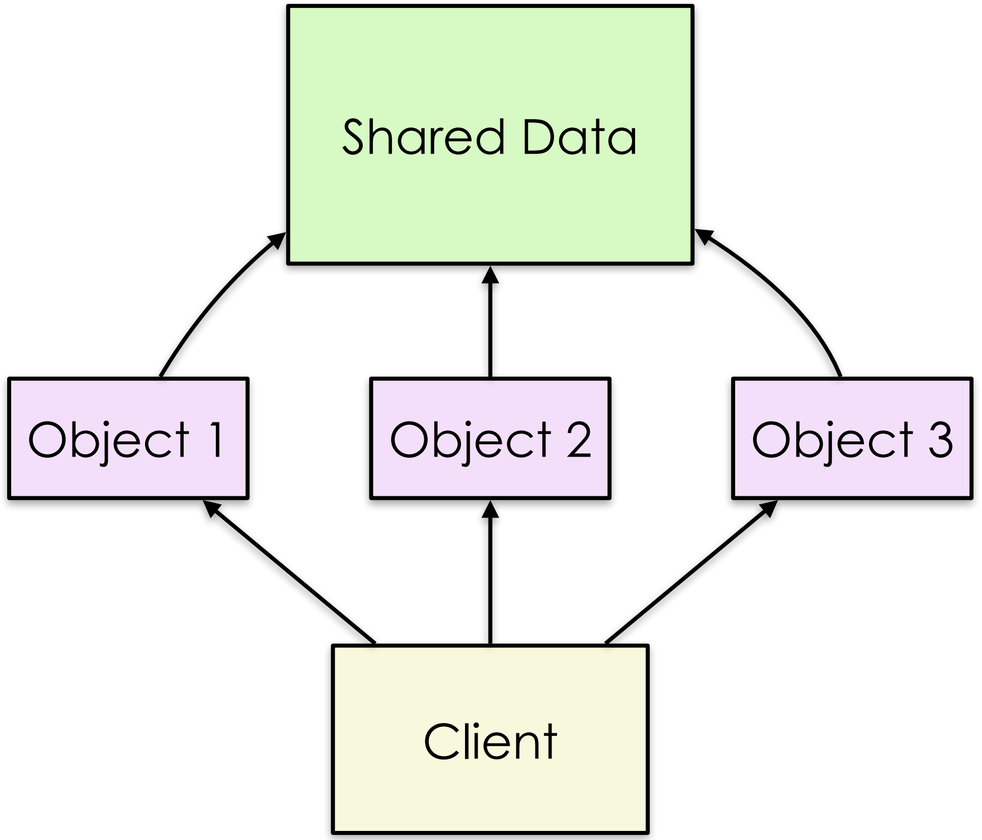
Problem

Imagine a Text Editor representing each character in its document as an individual object; the memory usage would be prohibitively large.



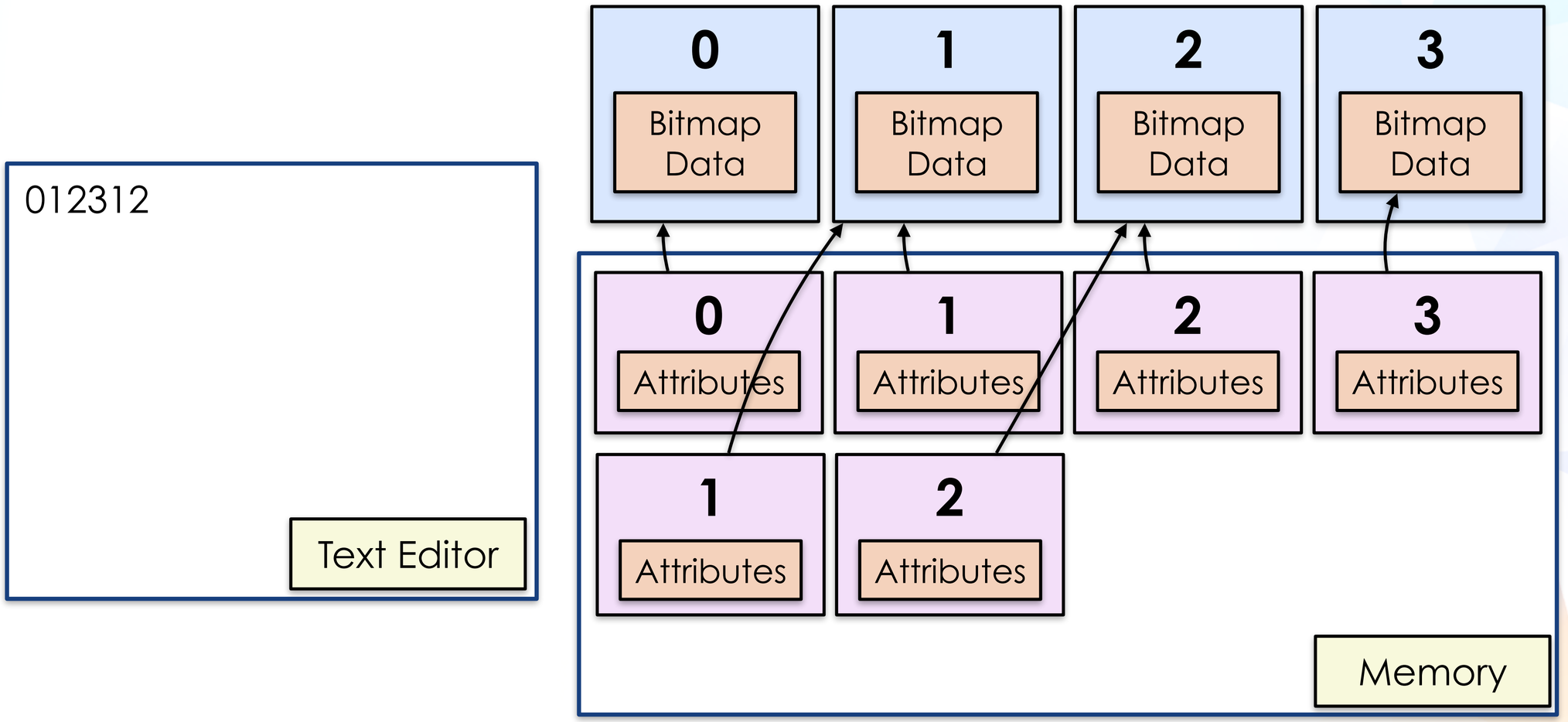
A Better Approach

Store shared data separately and only once, with each object referencing to the shared data as required



Problem Solved

Share common data (e.g. bitmap font representation); individual objects only store specific data (e.g. color of that particular character).



Go Implementation

- Review the Go implementation of the Flyweight design pattern located in the Flyweight folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code in order to determine
  - The Class that contains the core data used by other objects
  - The original code structure without the Flyweight design pattern

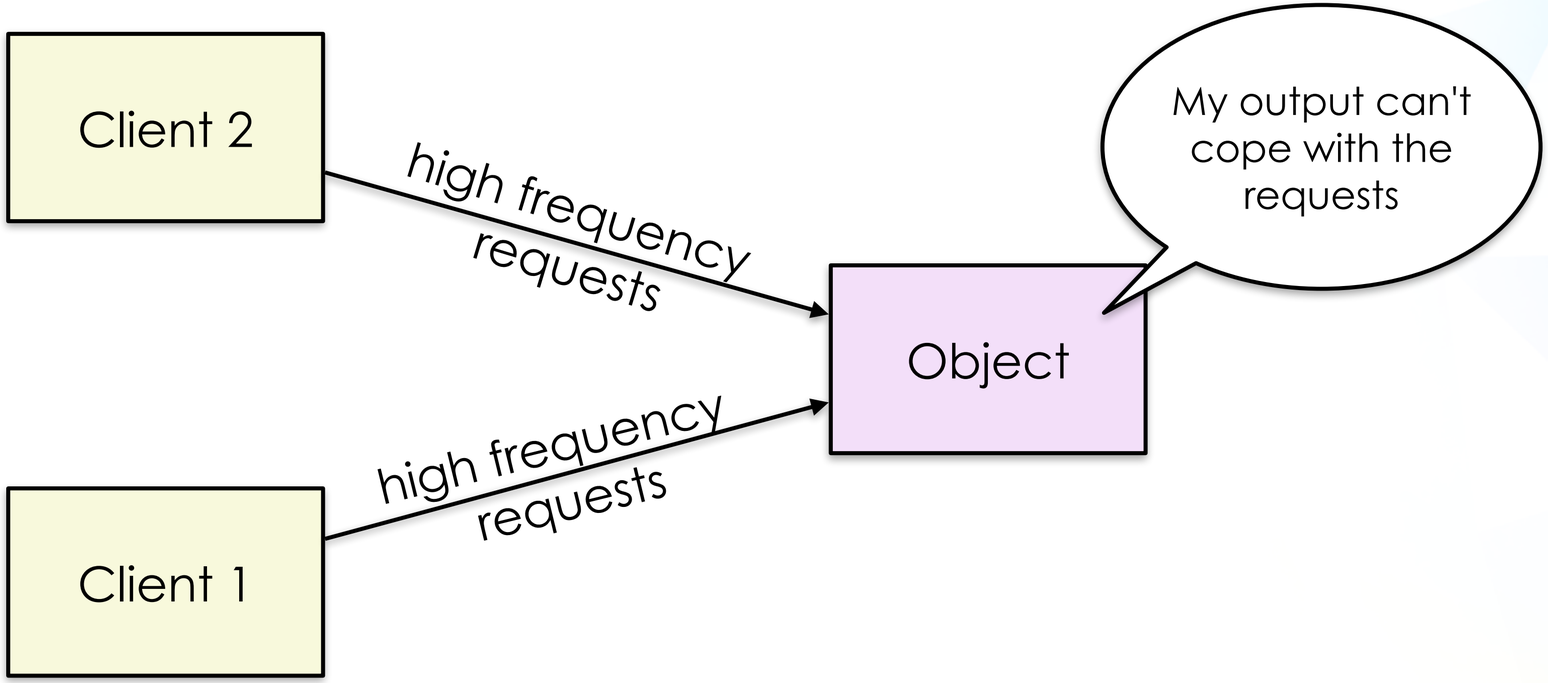
Proxy

A Design Challenge

- Consider a system where a client needs to frequently access a large dataset, but the cost of acquiring the data is high.
- The proxy provides a surrogate object that controls access to the original data and caches it for faster access, reducing the cost of repeatedly requesting the data from the source.
- Additionally, the proxy can also implement eviction and refresh policies to further optimise the access to the data.

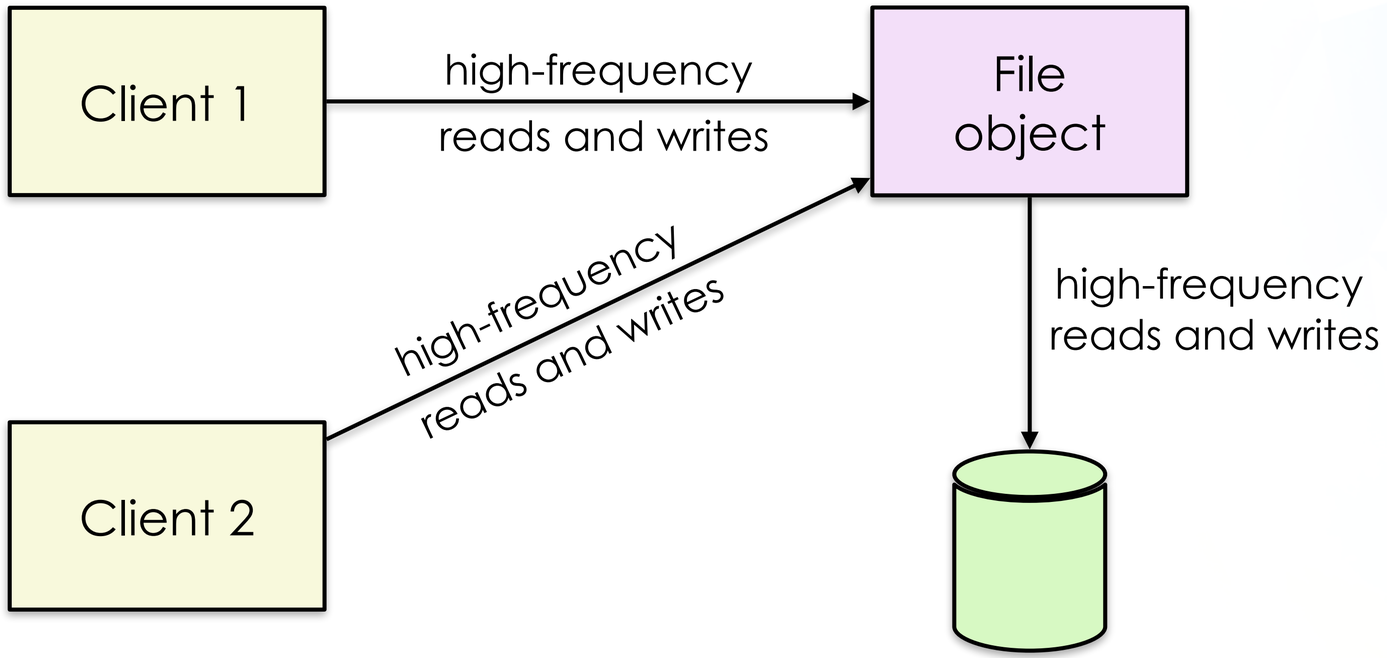
A Lesser Approach

Allow Client to access the target object directly, with high-frequency requests; potentially bogging down the target object and degrading response time



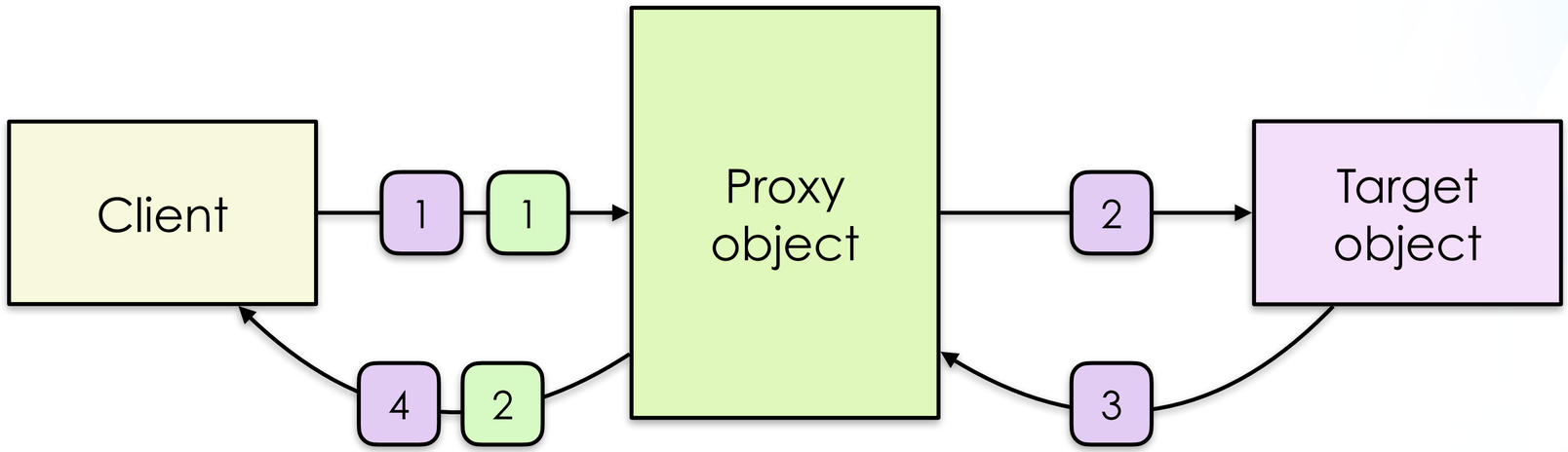
Problem

Saving and fetching data from SSDs at a high frequency can degrade the performance of the overall system and induces latency.



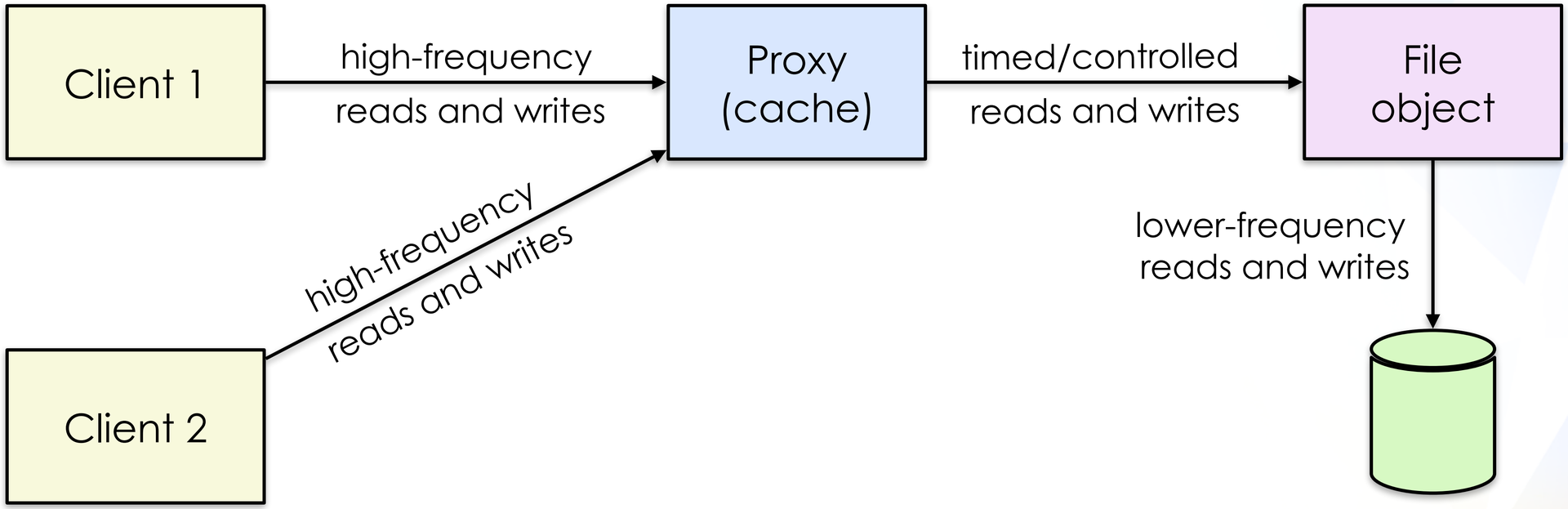
A Better Approach

Get the Client to access the source object via a Proxy object, thus providing technical benefits, such as response time, to the Client



Problem Solved

Have a Proxy object to act as a in-memory Cache; only read from SSDs when cache is stale and write to SSDs when the write-buffer (in the cache) is full



Go Implementation

- Review the Go implementation of the Proxy design pattern located in the Proxy folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code in order to determine
  - The function of the Proxy object
  - The storage mechanism implemented for the database layer

The End