Creational Patterns

Tan Cher Wah

cherwah@nus.edu.sg

Creational Patterns

Factory	The creation logic of a class is not exposed to the client; new objects can be created using a common interface.
Abstract Factory	Allows the creation of families of related objects without specifying their concrete classes.
Prototype	Allows the creation of objects by cloning existing ones instead of creating new ones from scratch.
Singleton	Provides a controlled object-creation mechanism such that only one instance of a given class exists.

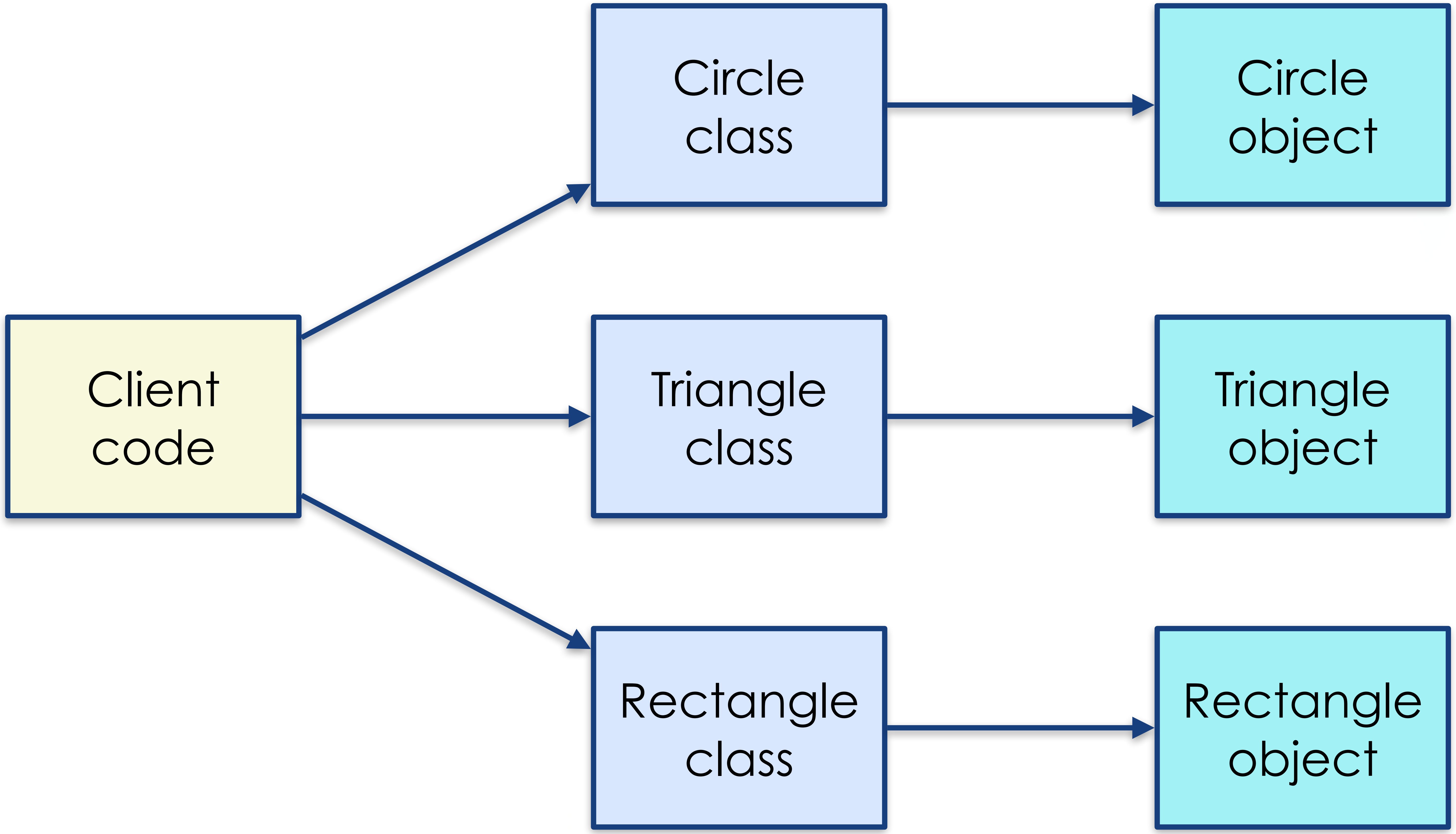
Factory

A Design Challenge

- Consider a software component that allows Shapes to be created for rendering on a graphical canvas
- Some of these Shapes are circles, triangles and rectangles
- Each Shape has its own set of properties such as color, width and height
- The various Shape classes should have some flexibility to change in future versions without affecting existing code that already uses it
- How to design such a system?

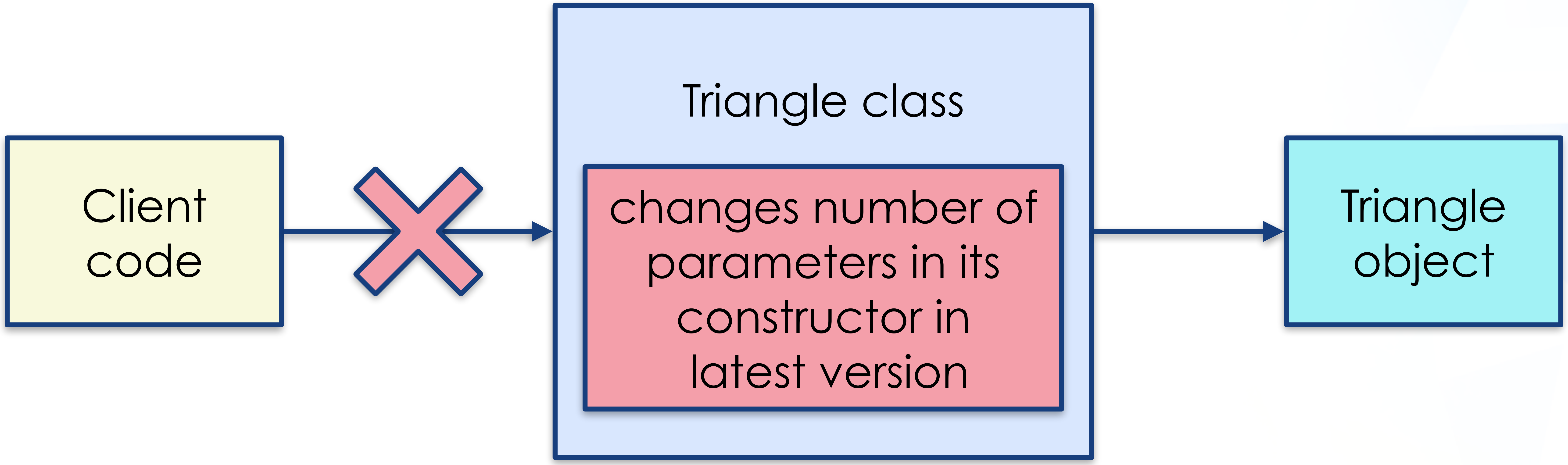
A Lesser Approach

Client **directly** invokes individual constructors to create the concrete shapes



Sample Problem

If any of the classes changed (e.g. adding or removing properties), the **Client is affected** and needs to be changed (in order to compile with the latest version of Triangle class)


Client code

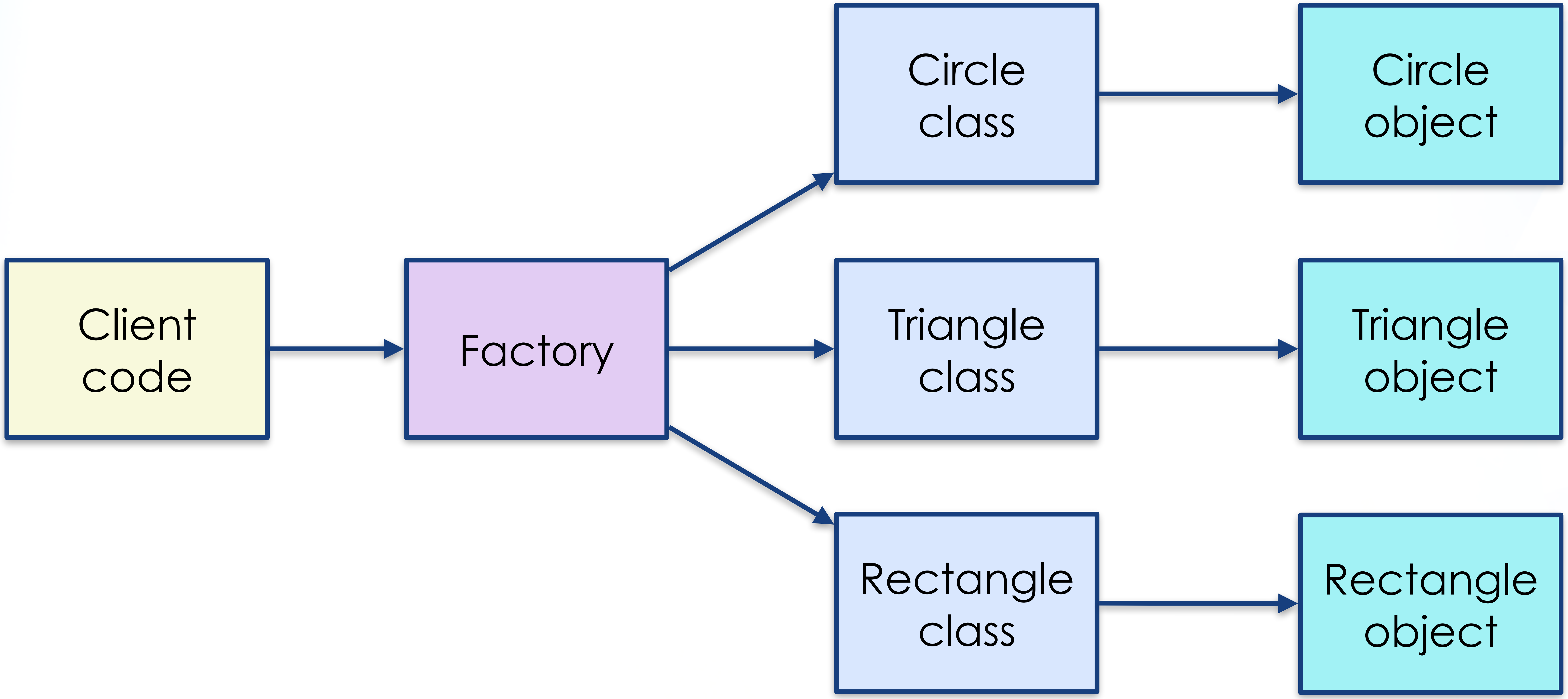
Triangle class

changes number of parameters in its constructor in latest version

Triangle object

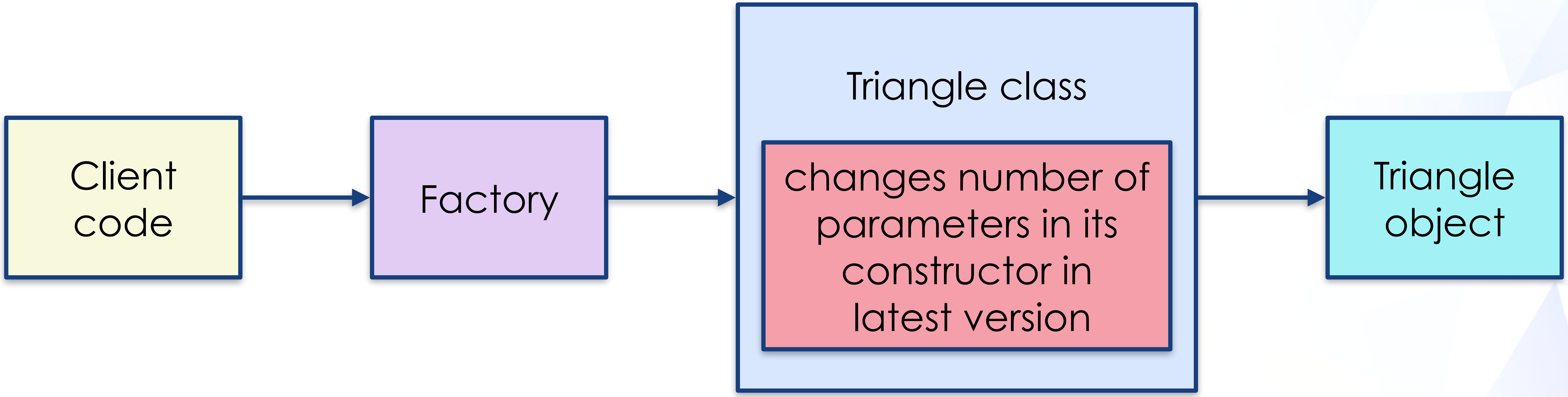
A Better Approach

Use the **Factory** design pattern such that Client creates **concrete shapes** through a Factory, which knows how to call the appropriate constructors of the different Shape classes



Problem Solved

The Factory's interface for the Client **remains the same**, but the Factory knows how to call the constructor of the updated Triangle class; Client is not affected by the change


Client code
Factory
Triangle class
changes number of parameters in its constructor in latest version
Triangle object

Go Implementation

- Review the Go implementation for the Factory design pattern under the Factory folder.
- This includes running and debugging the code to understand its design.
- Inspect the code to determine
  - How was the Factory pattern implemented?
  - What must be changed in Shape.go if the Factory pattern is not used?

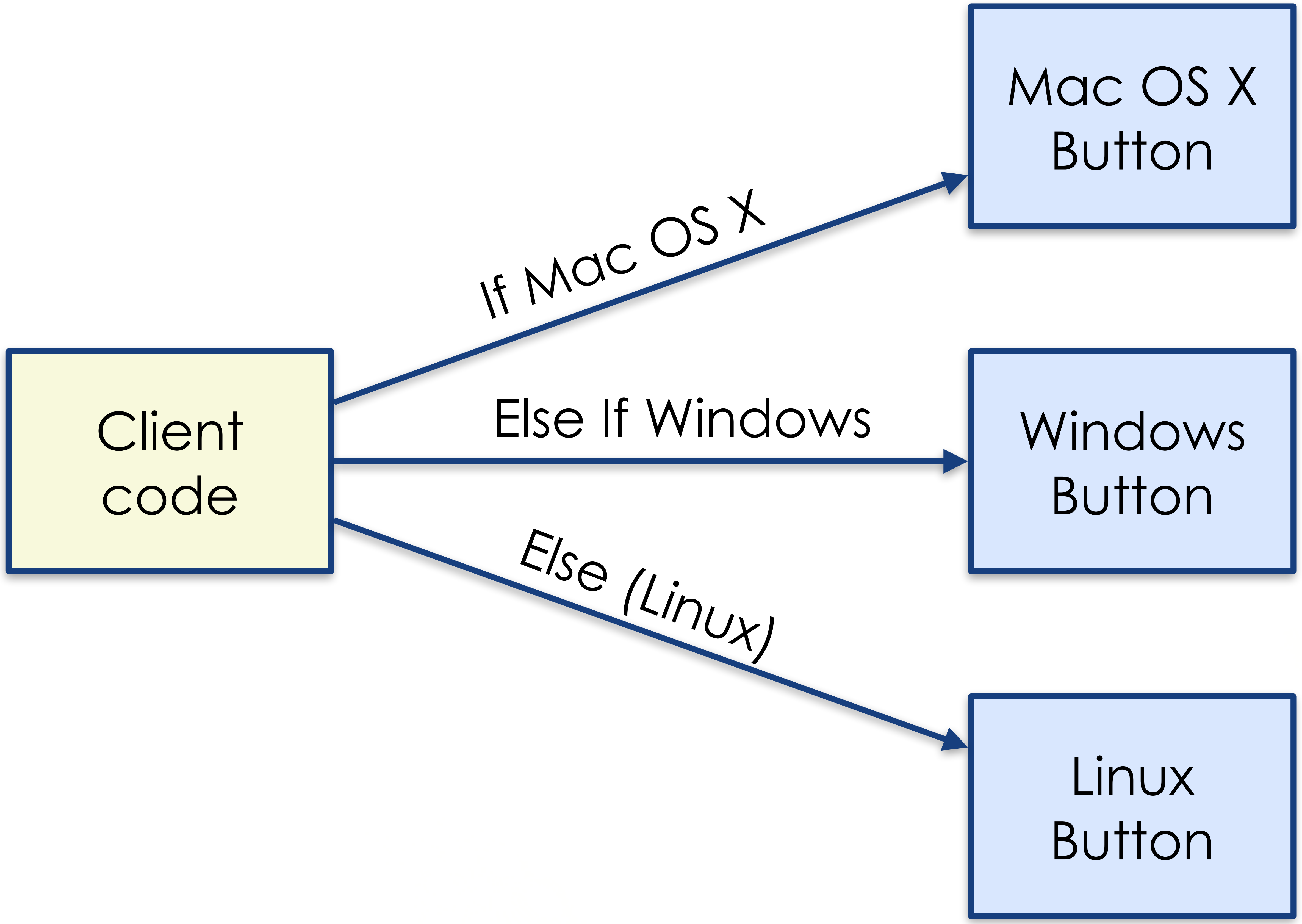
Abstract Factory

A Design Challenge

- Consider a software component that is designed to be portable among the following operating systems: Mac, Windows, Linux
- Native widgets will be instantiated depending on which OS the software is currently running on
- For example, on a Mac OS, Mac widgets should be created instead of a Windows or Linux widget
- How do we design such a system?

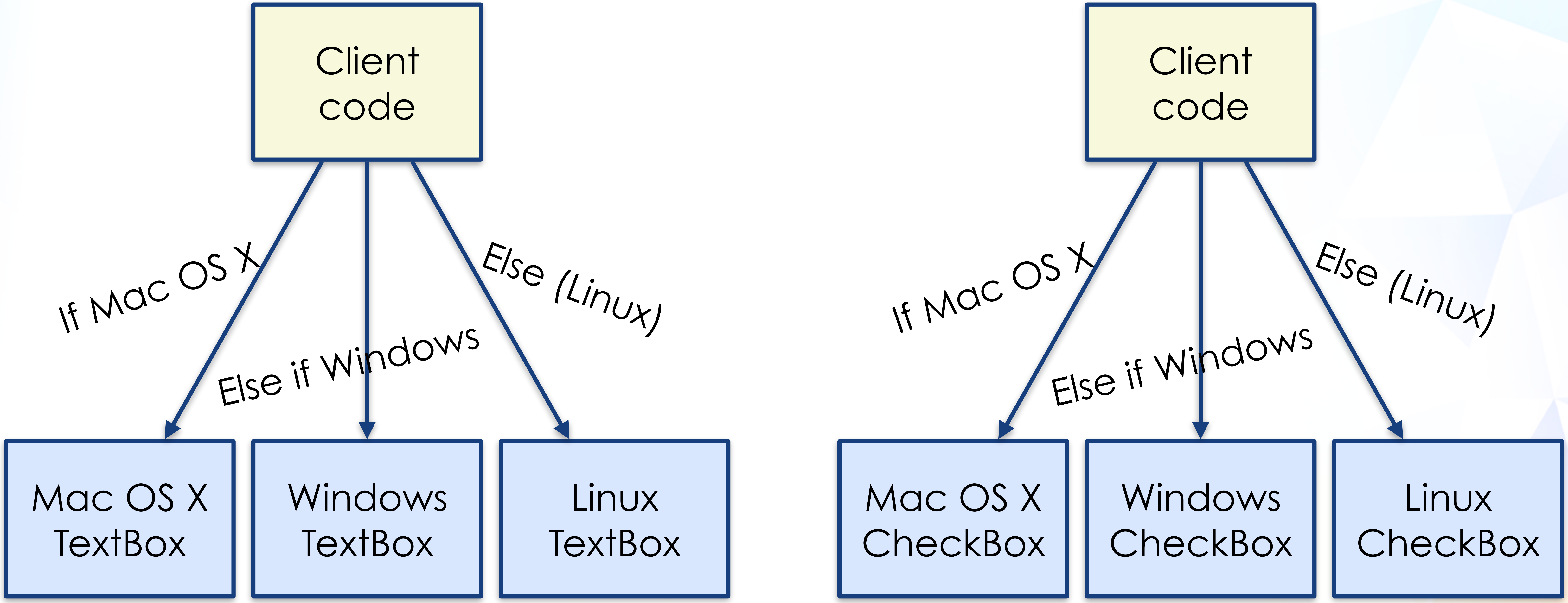
A Lesser Approach

Perform **conditional checks** to determine which OS the software is running on before instantiating the right kind of widget



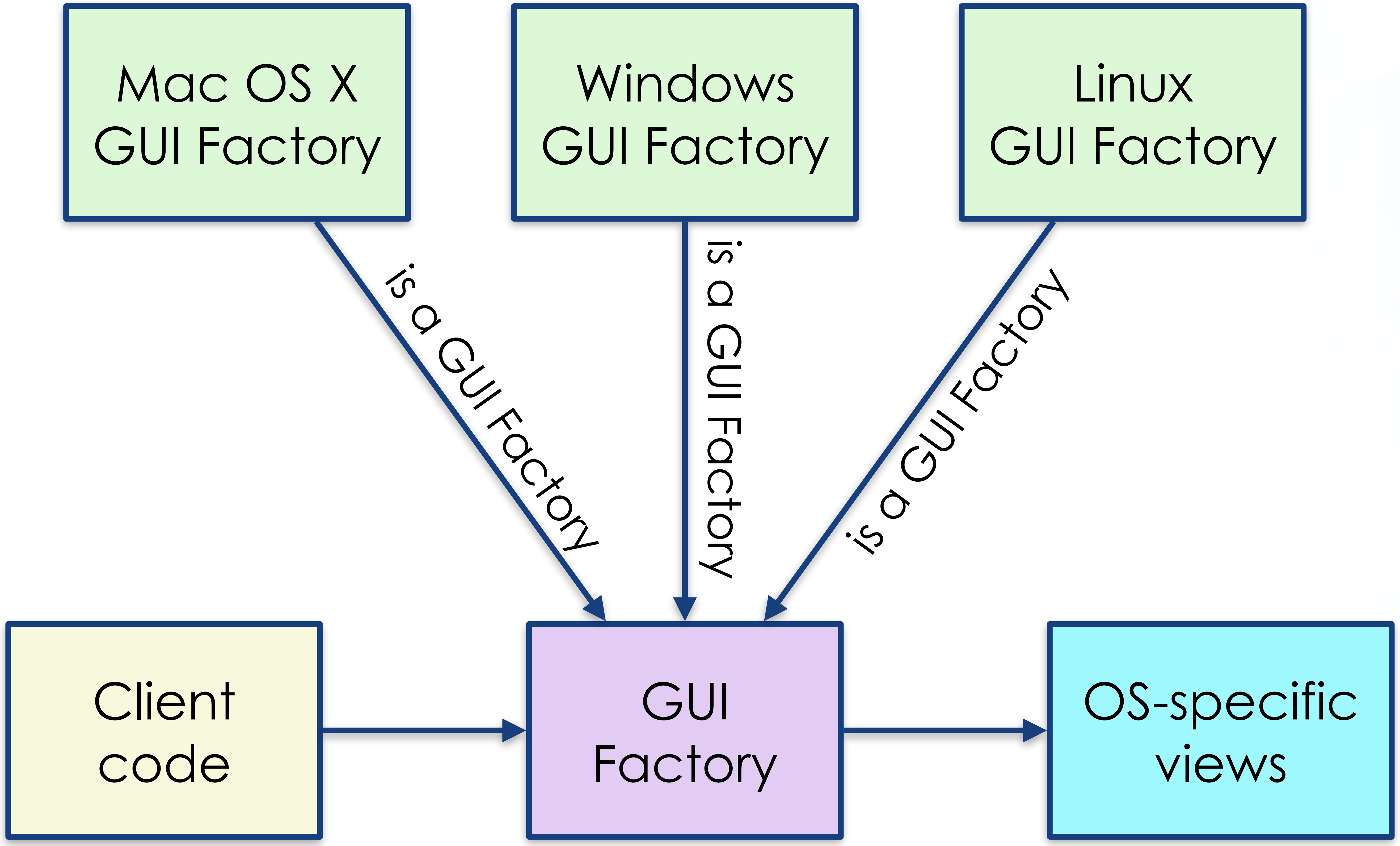
Problem

Code is filled with conditional checks; Object-Oriented Design is meant to free us from excessive if-else statements



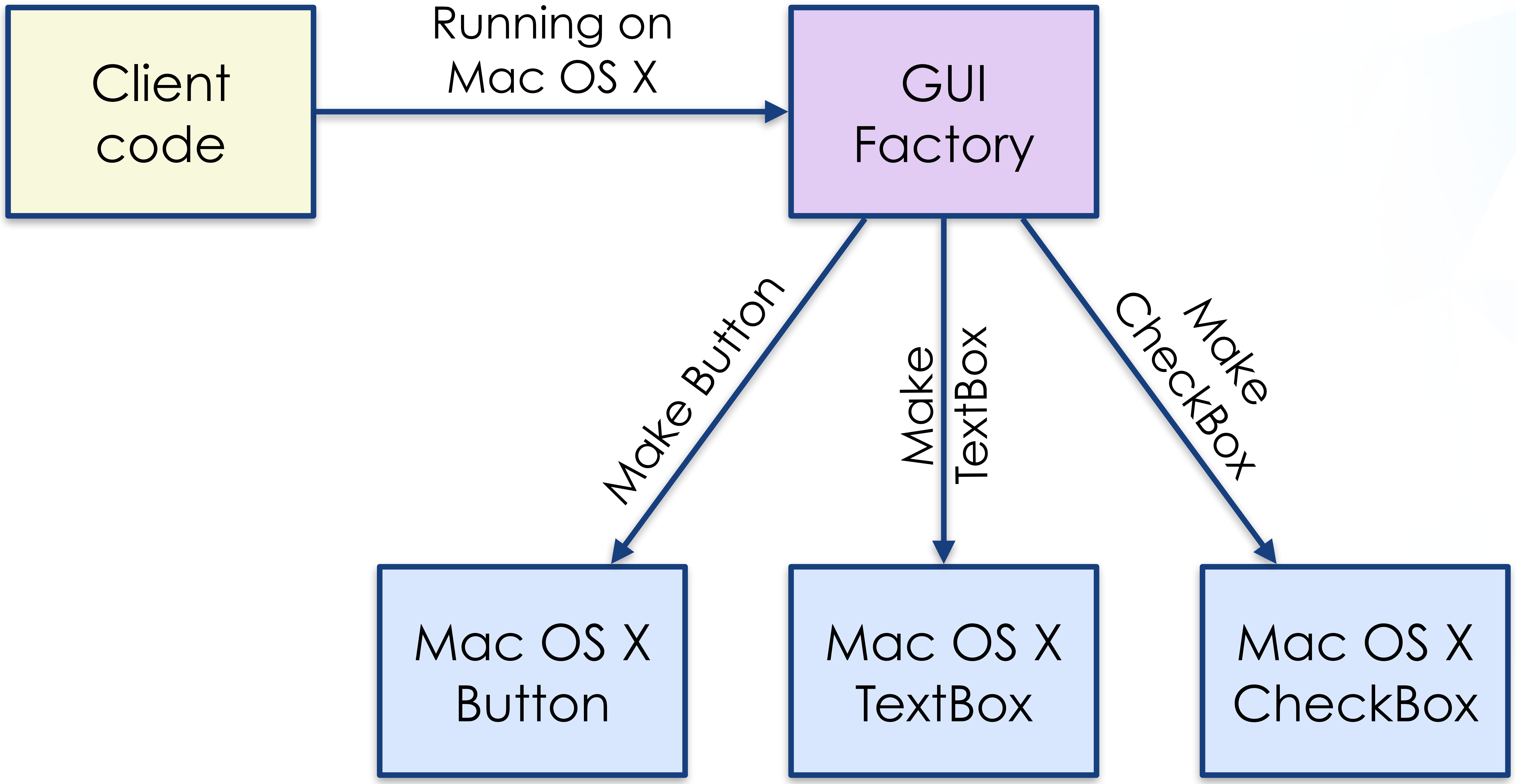
A Better Approach

Use the **Abstract Factory** design pattern such that Client creates desired **family of components** without conditional checks



Problem Solved

The appropriate views are now created based on the OS-specific GUI Factory; no conditional checks are needed



Go Implementation

- Review the Go implementation for the Abstract Factory design pattern under the Abstract_Factory folder
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code to determine
  - How was the Abstract Factory pattern implemented?
  - How was this implementation different from the Factory pattern?

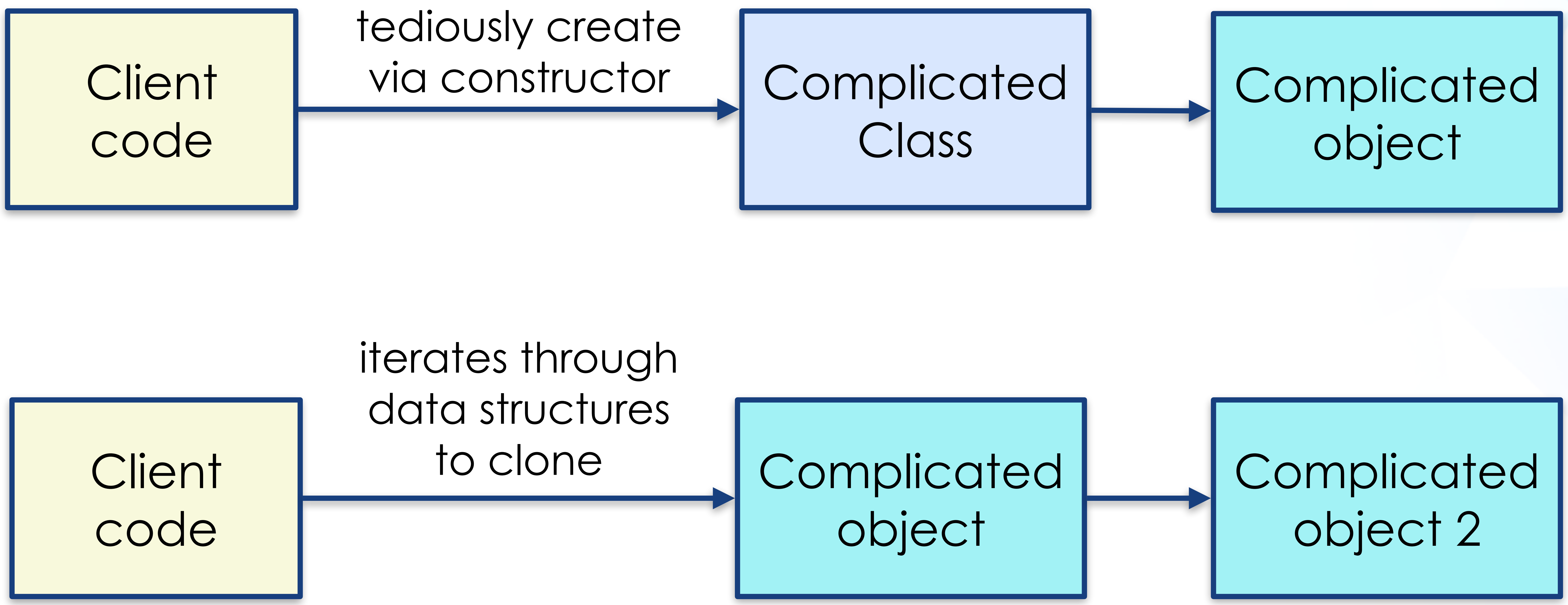
Prototype

A Design Challenge

- Consider a Class that accepts numerous parameters in its constructor and uses complicated data structures for storage
- Creating a concrete object from the Class is also **expensive** as it needs to query multiple entities like the database and network
- Making a copy of a concrete object is difficult without intimate understanding of its data structures
- Such a Class would be hard to learn and use
- How to overcome the design challenge?

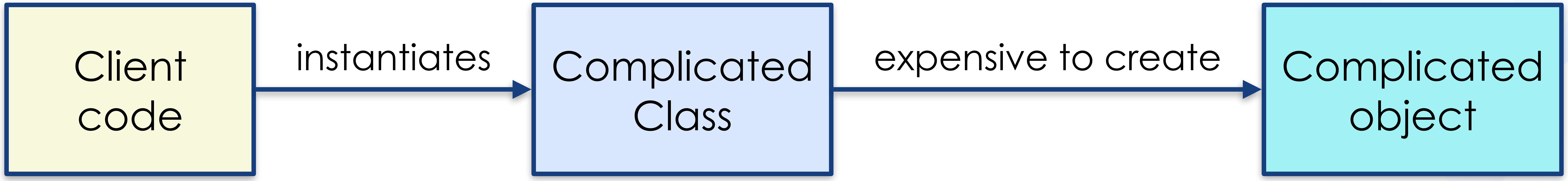
A Lesser Approach

Expose the Class's complicated constructor and data structures to Clients



Sample Problem A

Creating a concrete object from scratch from the Complicated Class consumes too much resources (e.g. time, memory); noticeable performance issue when done often



Client code

instantiates

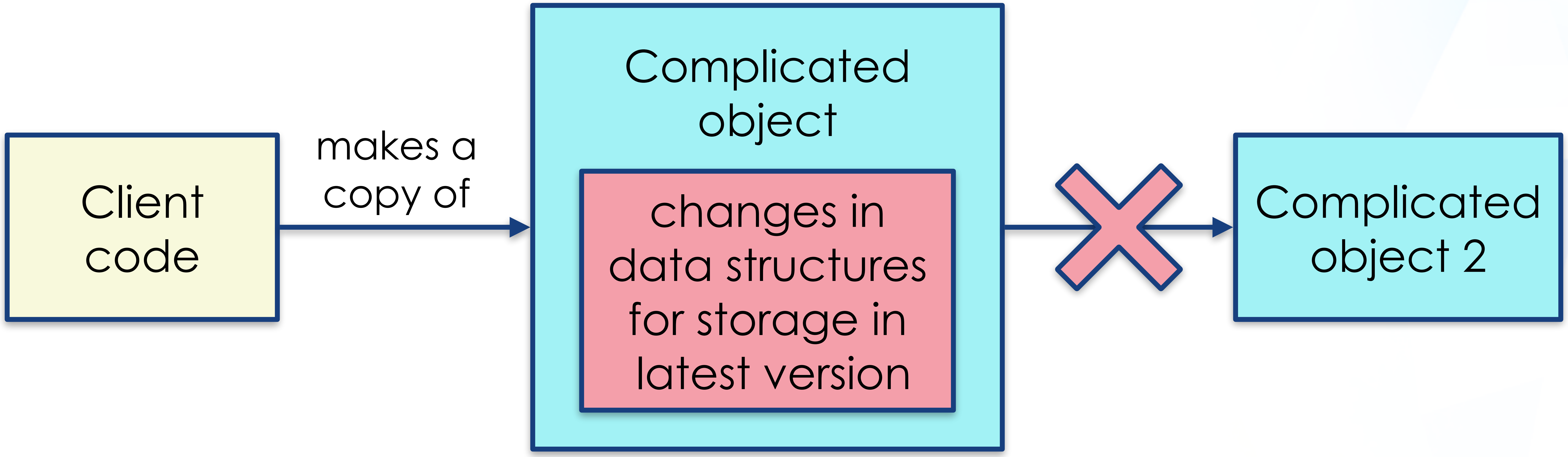
Complicated Class

expensive to create

Complicated object

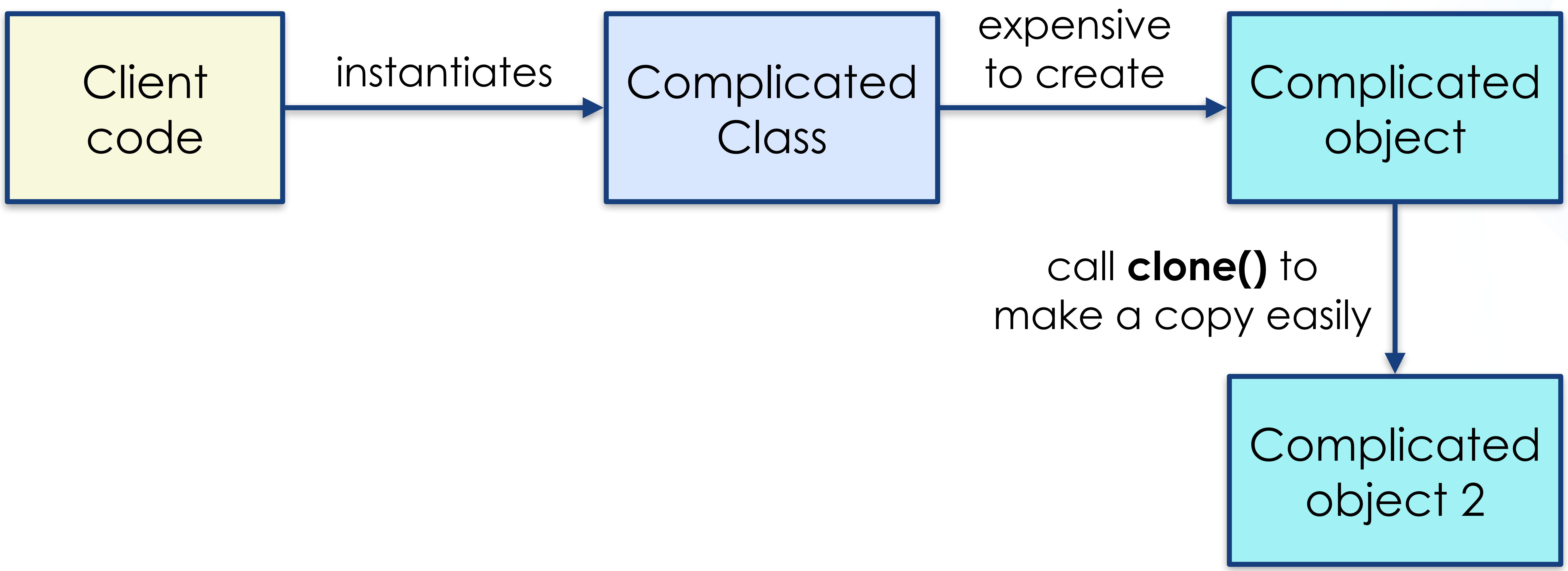
Sample Problem B

Any changes in data structures of the Complicated Class would invalidate Client's code to clone the object



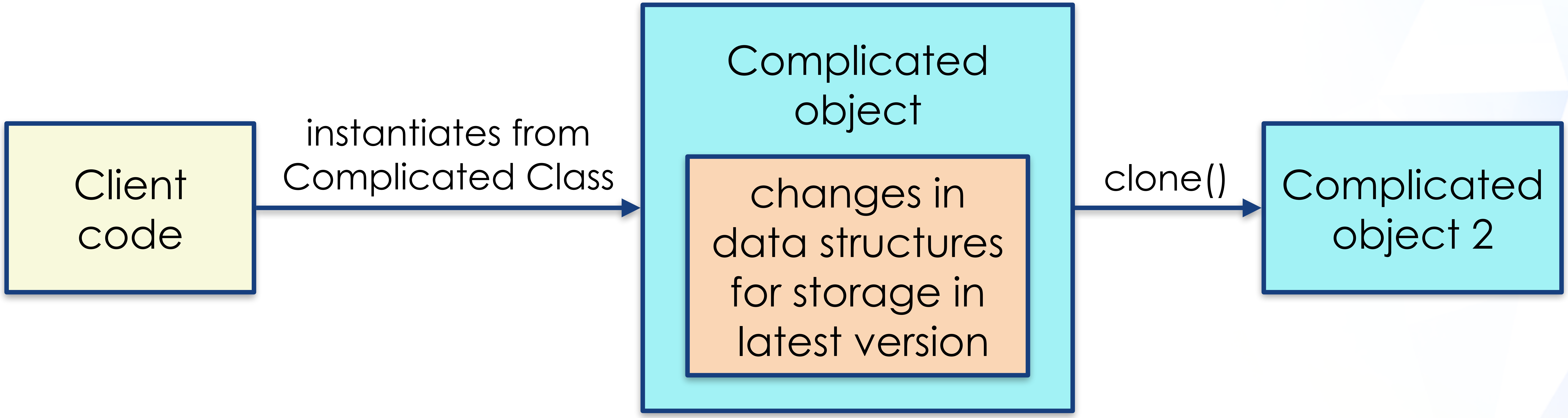
A Better Approach

Provide a **clone** method in the complicated class so that Client can call it to make cheap duplicates of the complicated object; expensive creation operation incurred only once.



Problems Solved

Client no longer needs to know the different data structures of the complicated object to perform a copy; any changes in later versions has no effect on it as it can simply call clone() to make copies


Client code
instantiates from Complicated Class
Complicated object
changes in data structures for storage in latest version
clone()
Complicated object 2

Go Implementation

- Review the Go implementation for the Prototype design pattern under the Prototype folder
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code to determine
  - How was the Prototype pattern implemented?

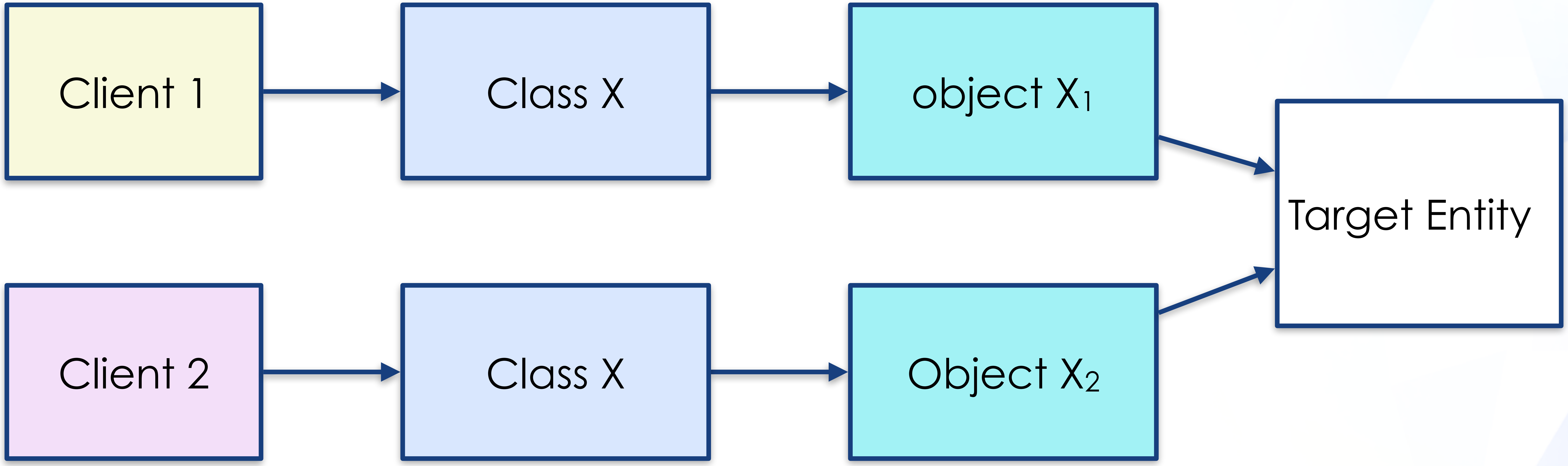
Singleton

A Design Challenge

- Consider a Class that performs operations such as logging or making database connections
- As multiple Clients in the system can instantiate the same Class, having more than one instance of the Class can have unexpected outcomes
  - Two Logger objects might be writing to the same file, at the same time, and overlaps each other's sentence
  - Two many Connection objects might be created, by multiple clients, that drain the memory resources of the target entity (e.g. database)

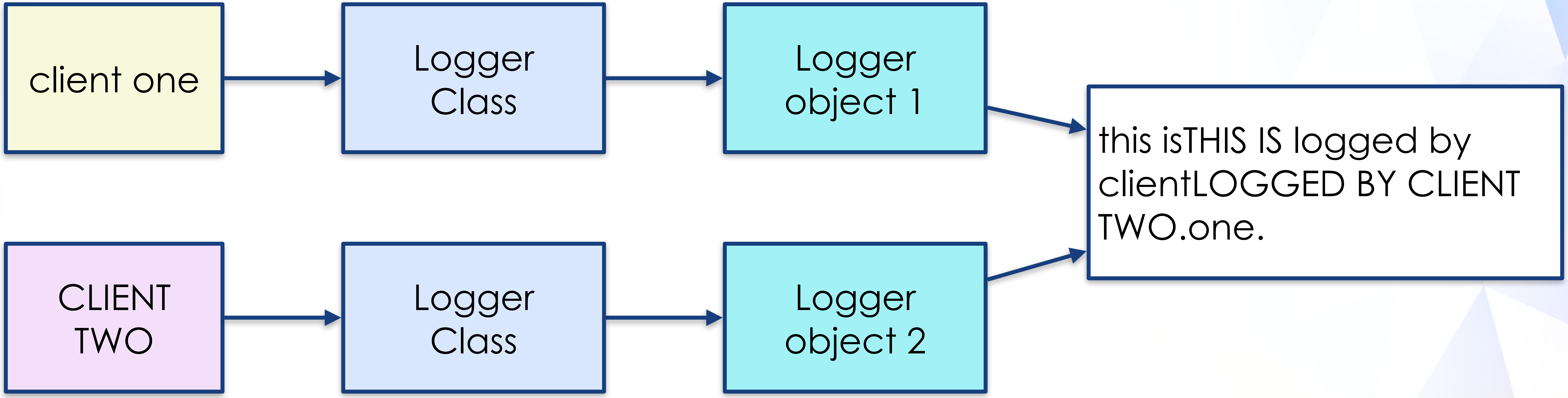
A Lesser Approach

Expose the creation of the Class to Clients and have multiple instances of itself accessing the target entity.



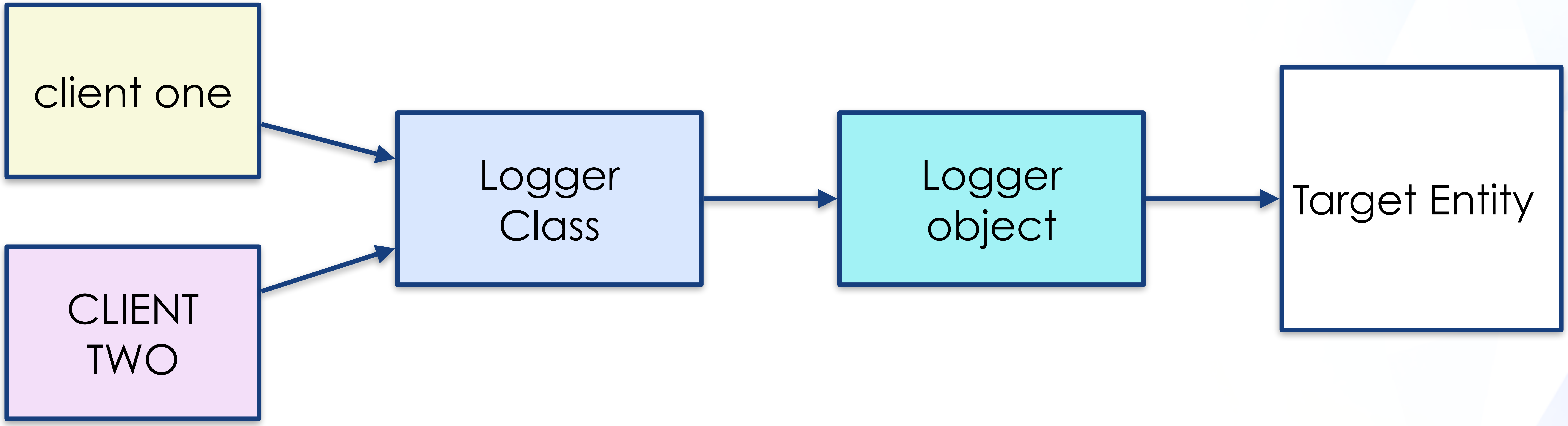
Sample Problem

This example shows multiple logger objects writing to the same file; but overstepping on each other's sentence in the process.



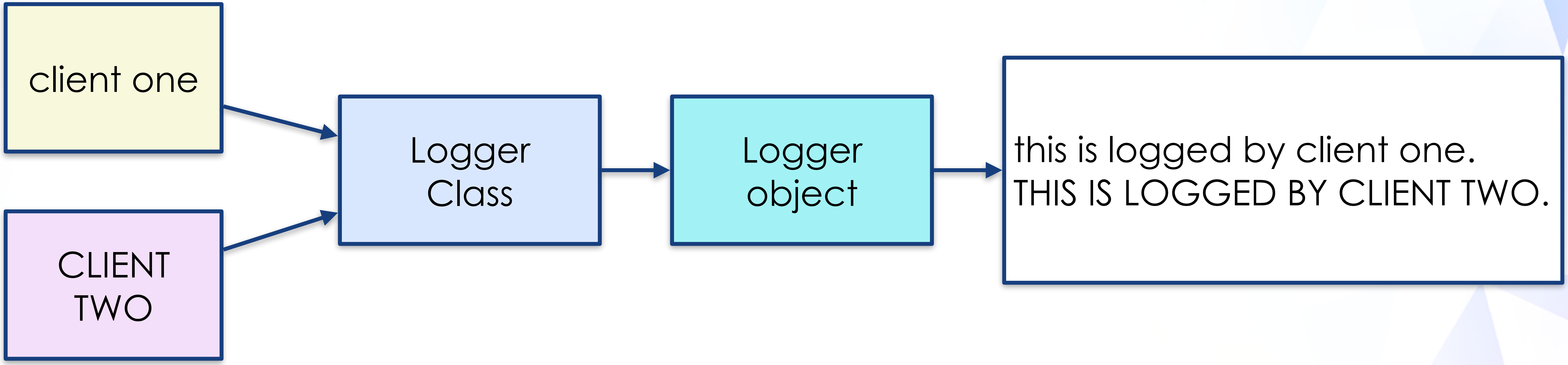
A Better Approach

Only expose a single instance of the Class to Clients, such that a single object is operating on the target entity.



Problem Solved

As there is only a single Logger object, writing to the file is now in an ordered manner



this is logged by client one.
THIS IS LOGGED BY CLIENT TWO.

Go Implementation

- Review the Go implementation for the Singleton design pattern under the Singleton folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code to determine
  - How was the Singleton pattern implemented?
  - How did we prevent the Logger from being instantiated directly?
  - Which part of the code ensure that only a single instance of the Singleton class was created?

THE END