Behavioral Patterns

Tan Cher Wah

cherwah@nus.edu.sg

Behavioral Patterns

Mediator	Defines a middle-man that controls the communication among other objects.
Strategy	Defines a family of algorithms, encapsulates each one, and make them interchangeable.
Observer	Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
Visitor	Represents an operation to be performed on the elements of an object structure

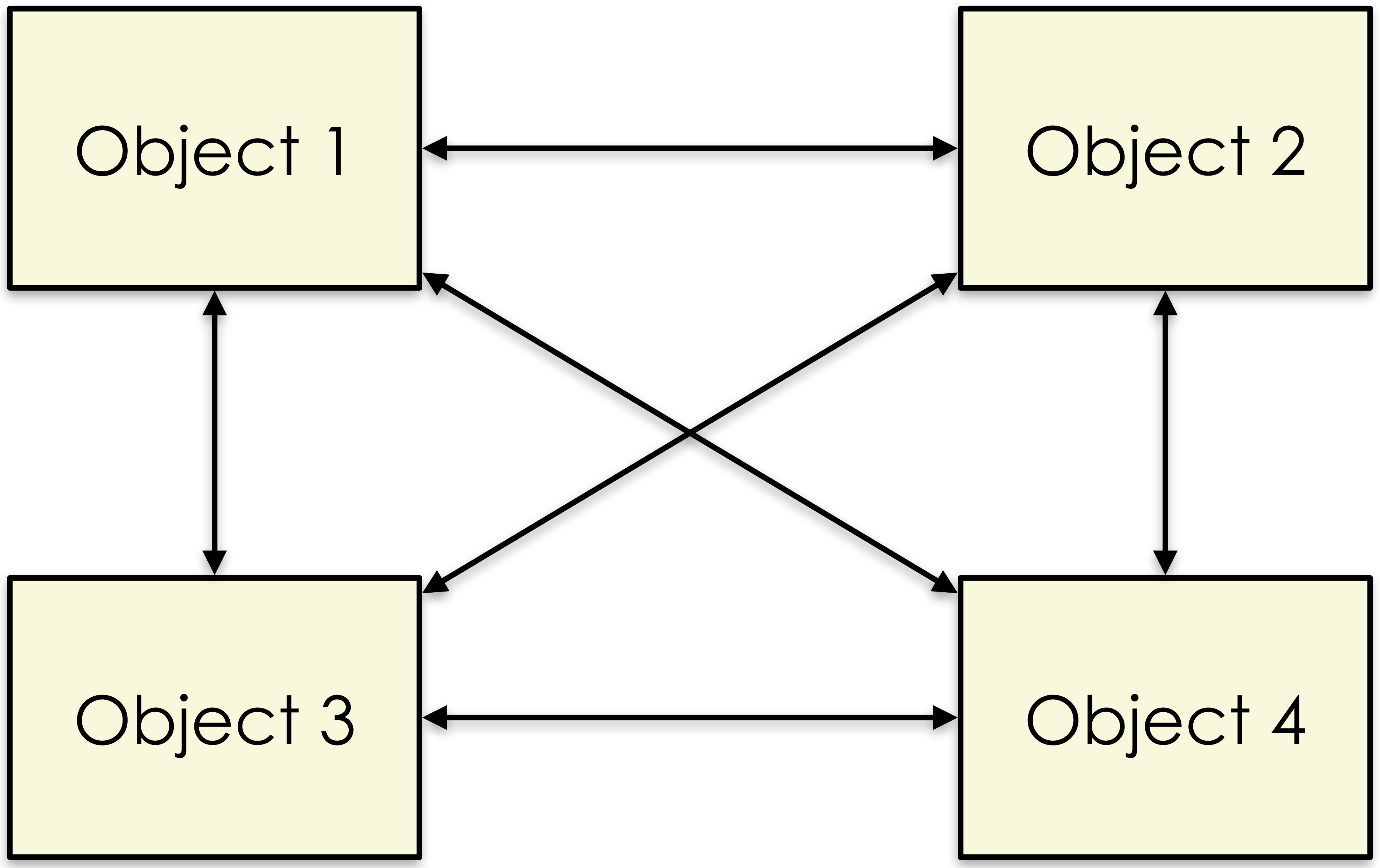
Mediator

A Design Challenge

- Consider a system where multiple objects need to communicate with each other, but we want to avoid tight coupling between them.
- The objects should not have direct references to each other, but still need to exchange information.
- Additionally, we want to be able to change the communication structure of the system without affecting the objects themselves.
- We need a way to encapsulate the communication between objects and make it more manageable, while also ensuring that the system can handle any future changes or requirements.

A Lesser Approach

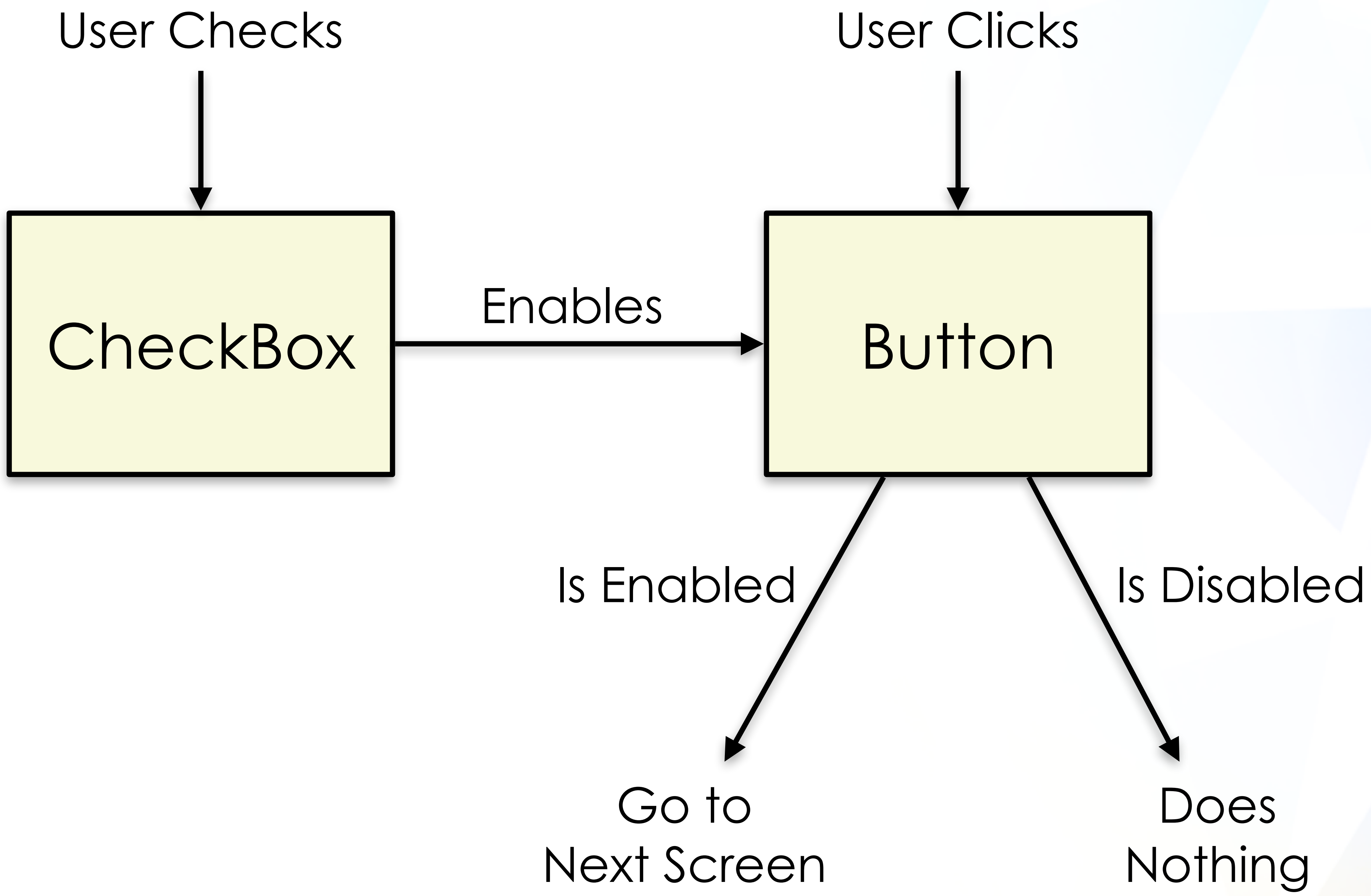
Allow objects to communicate with each other directly



Sample Problem

The objects are tightly coupled to each other when they hold too much of the program's logic

Our Button is **disabled initially**. When the Checkbox is clicked, it enables the Button. Clicking on the enabled-Button moves the user to the next screen.



User Checks

User Clicks

CheckBox

Enables

Button

Is Enabled

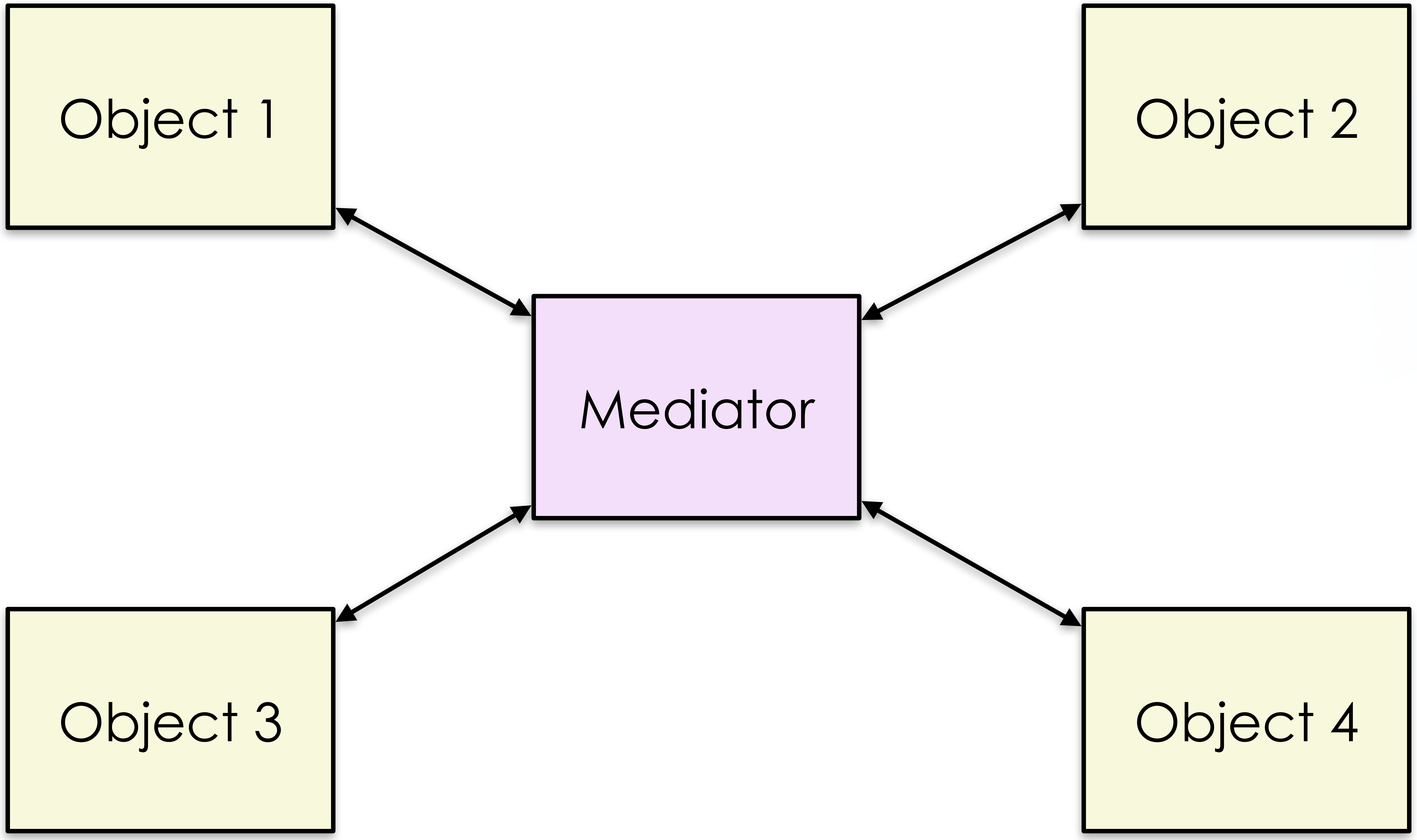
Is Disabled

Go to Next Screen

Does Nothing

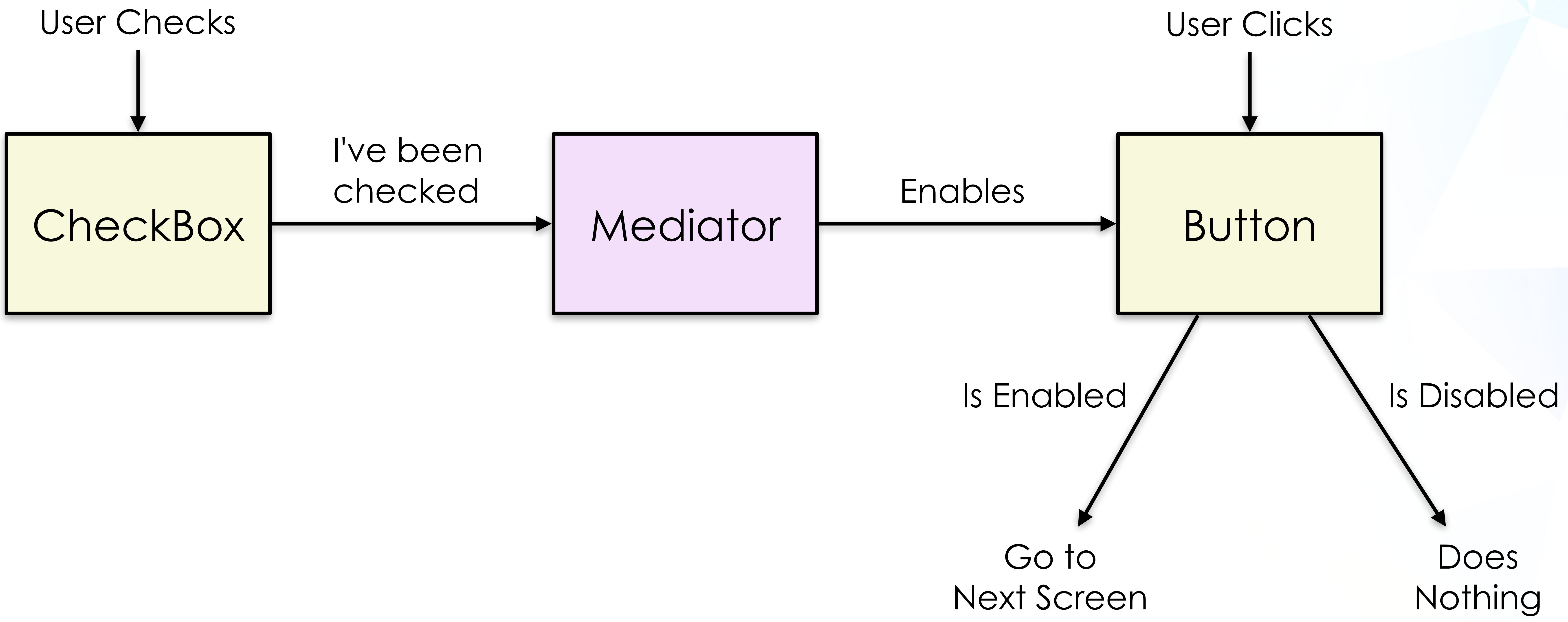
A Better Approach

All objects only communicate with a Mediator



Problem Solved

The Mediator holds the logic to determine what to do next when the Checkbox has been clicked



The Mediator receives a notification from Checkbox and asks the Button to enable itself

Go Implementation

- Review the Go implementation of the Mediator design pattern located in the Mediator folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code in order to determine
  - How the Mediator design pattern enabled loose-coupling between the Button and Checkbox classes
  - Identify the class that acts as the Mediator in the implementation and examine the program's behaviour when the Button is clicked before checking the Checkbox

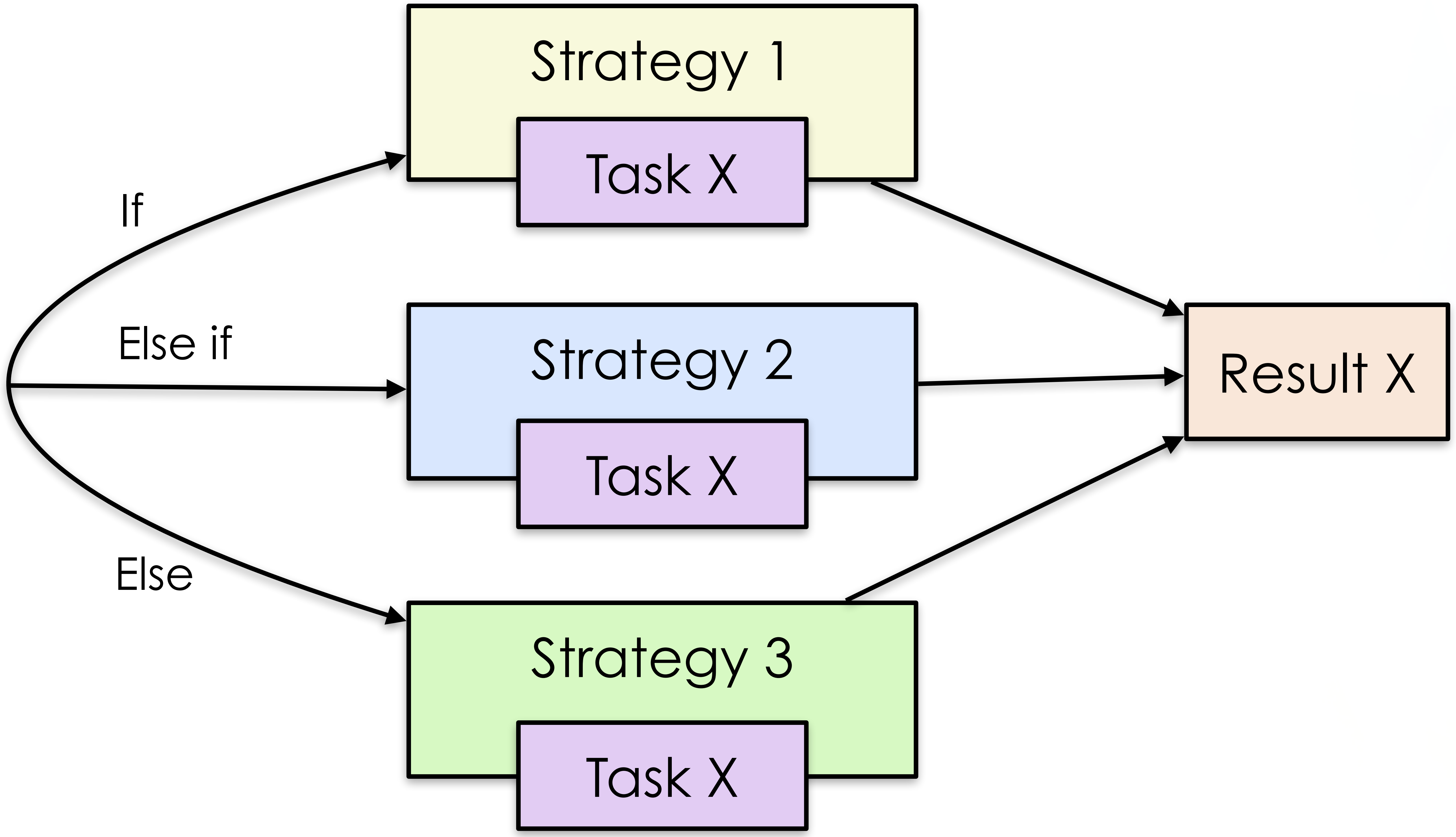
Strategy

A Design Challenge

- Consider a system that requires multiple interchangeable algorithms to accomplish a specific task.
- We want to avoid using a large number of if-else statements or switch-case statements to handle different algorithms.
- Additionally, we want to be able to change the algorithm used by the system without affecting the client code that uses it.
- We need a way to encapsulate different algorithms and make them interchangeable within the system, while also ensuring that the system can handle any future changes or requirements.

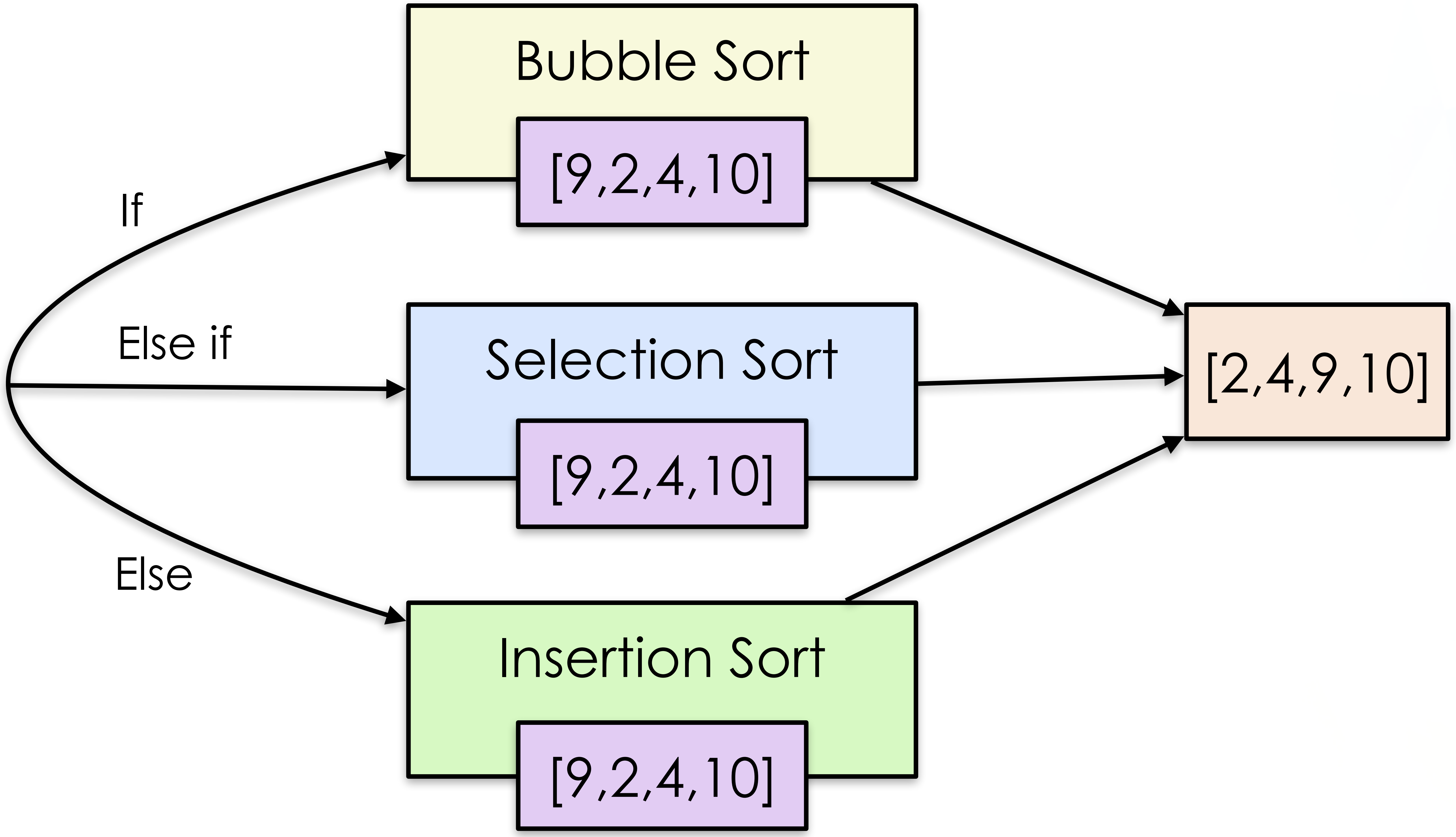
A Lesser Approach

Uses If-Else statements in selecting a strategy to apply to a task



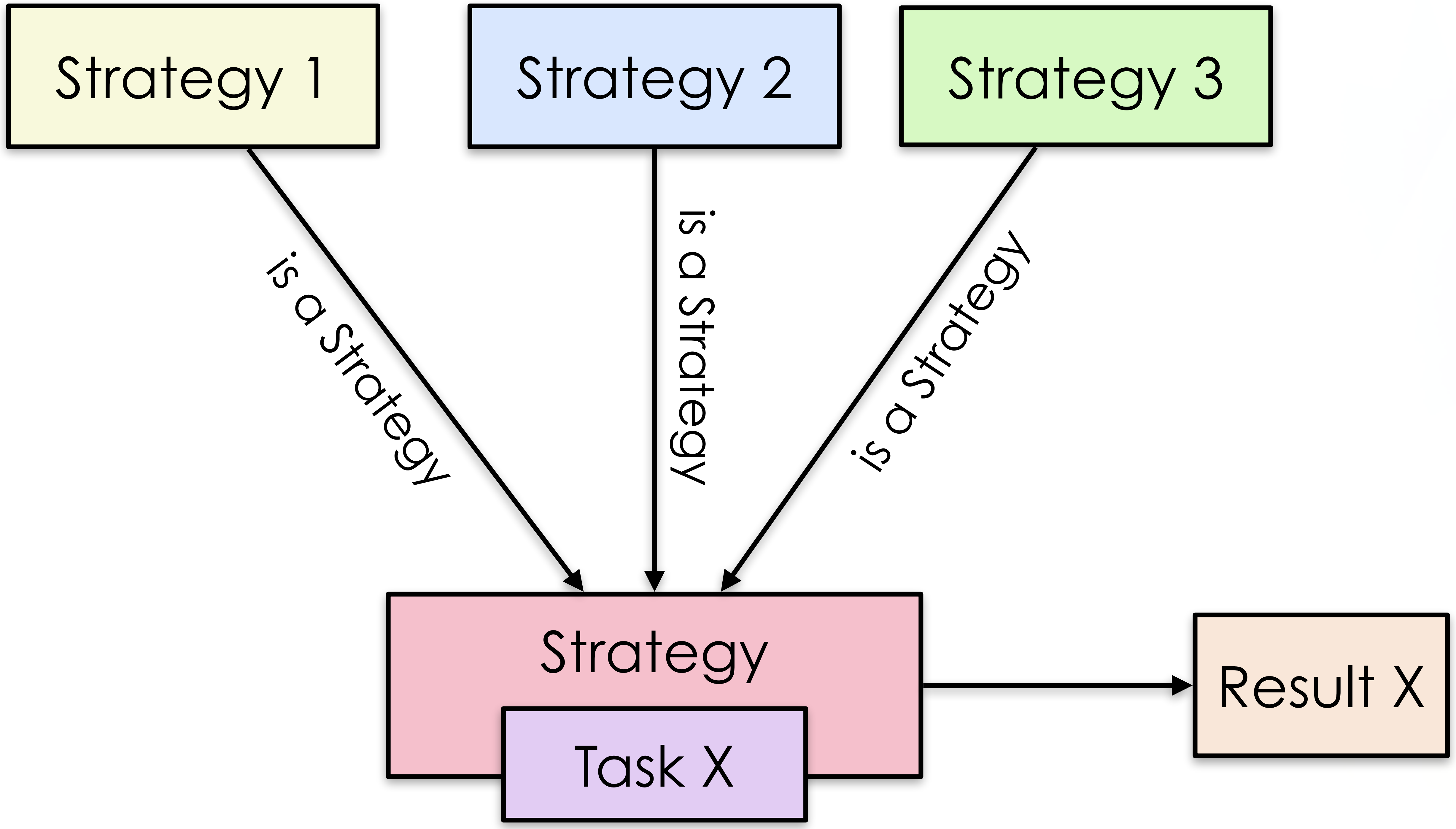
Sample Problem

Uses If-Else statements to select an algorithm to sort an array of integers



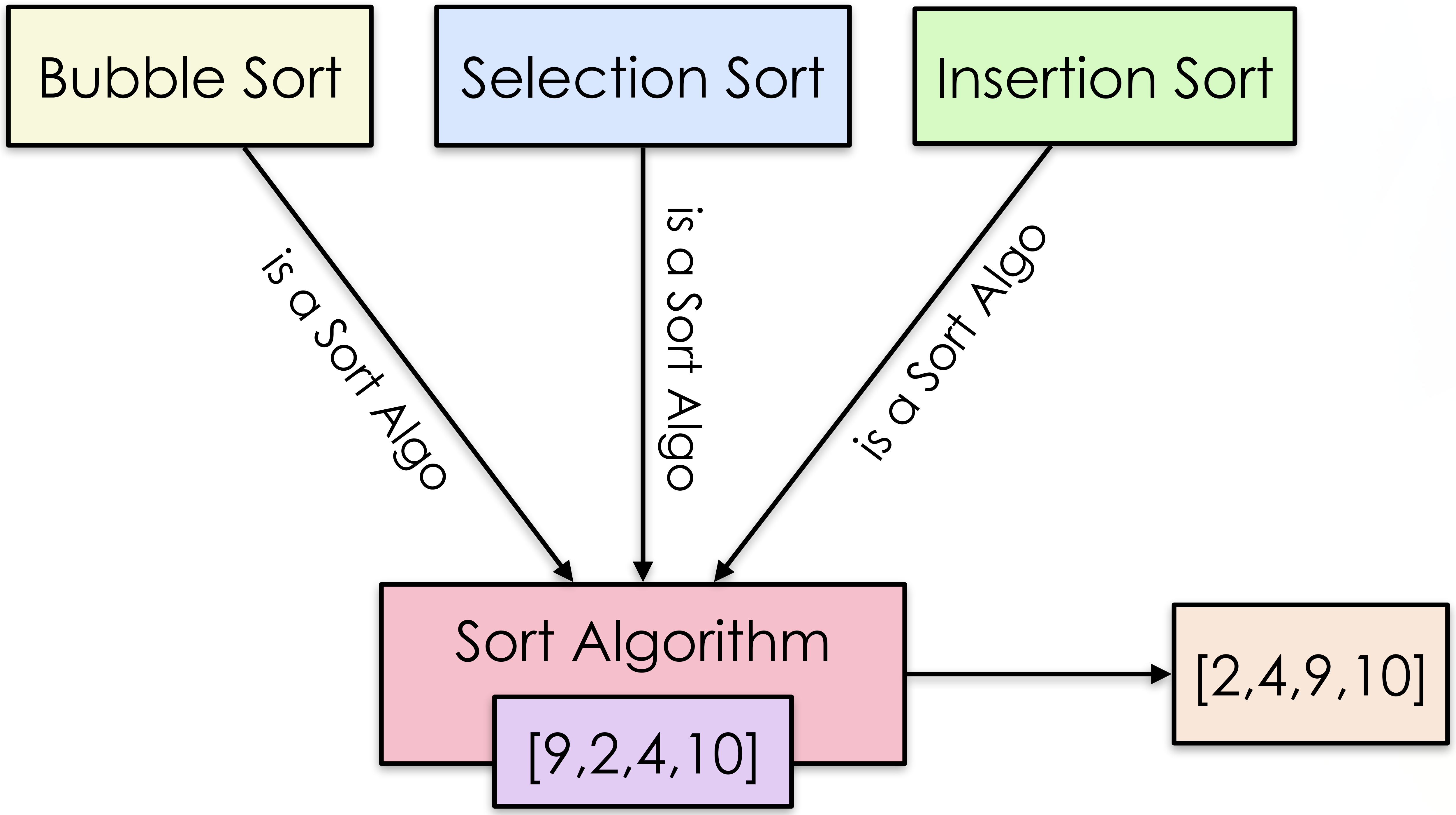
A Better Approach

Design all Strategies to conform to a standardised interface for interchangeability



Problem Solved

Each Sort algorithm implements a Sort() interface



Go Implementation

- Review the Go implementation of the Strategy design pattern located in the Strategy folder.
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code in order to determine
  - Which language construct of Go enables the ease of implementation of the Strategy design pattern
  - How should new sorting algorithms be designed such that they can fit in easily with the existing design

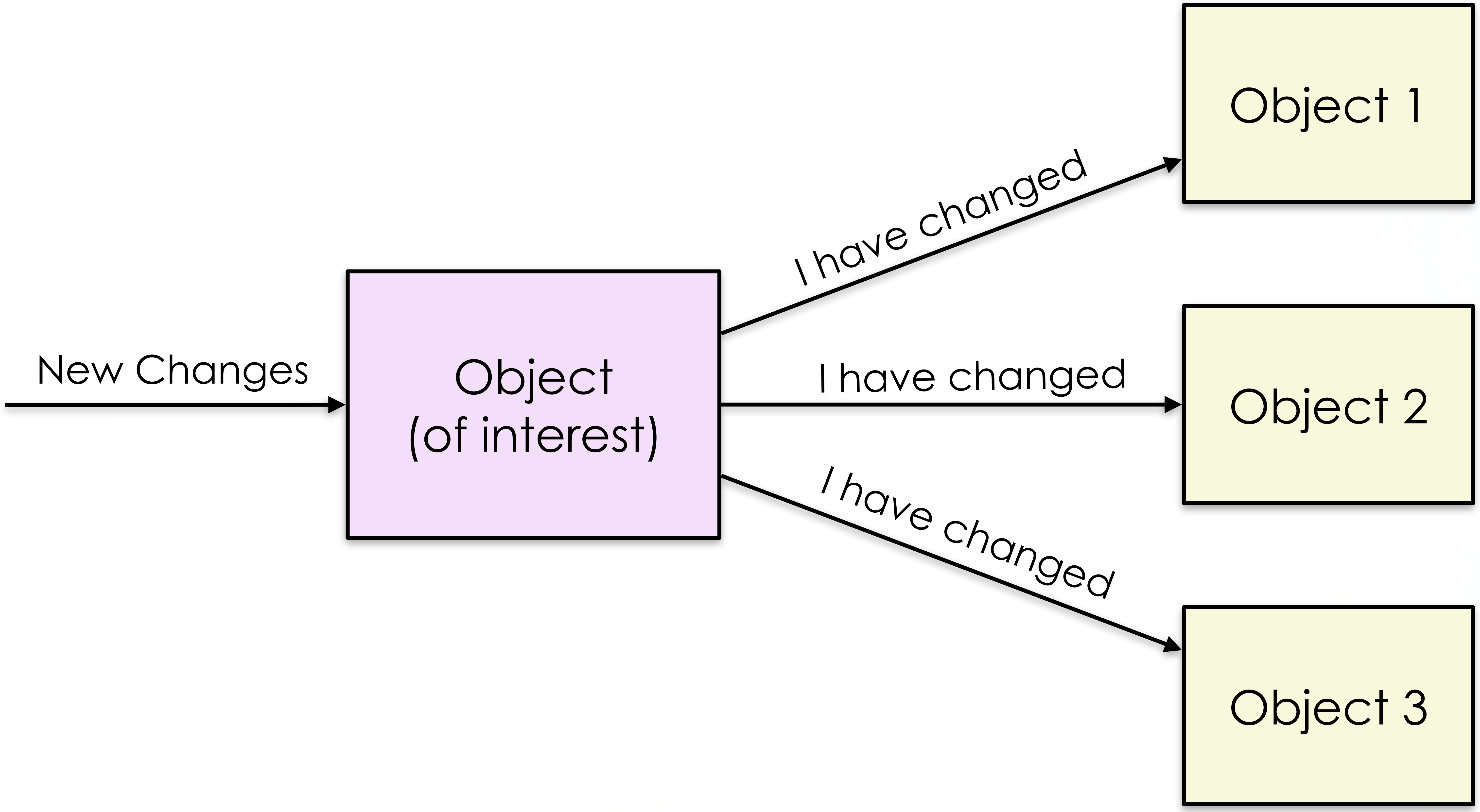
Observer

A Design Challenge

- Consider a system where multiple objects need to be aware of the changes happening in a specific object, and take necessary actions based on those changes.
- However, the specific object should not be tightly coupled with the objects that need to be notified of the changes.
- This will allow for flexibility, maintainability and scalability of the system.
- Additionally, the system needs to be event-driven, where changes to the state of one object can trigger updates or actions in other objects in a coordinated and efficient way.

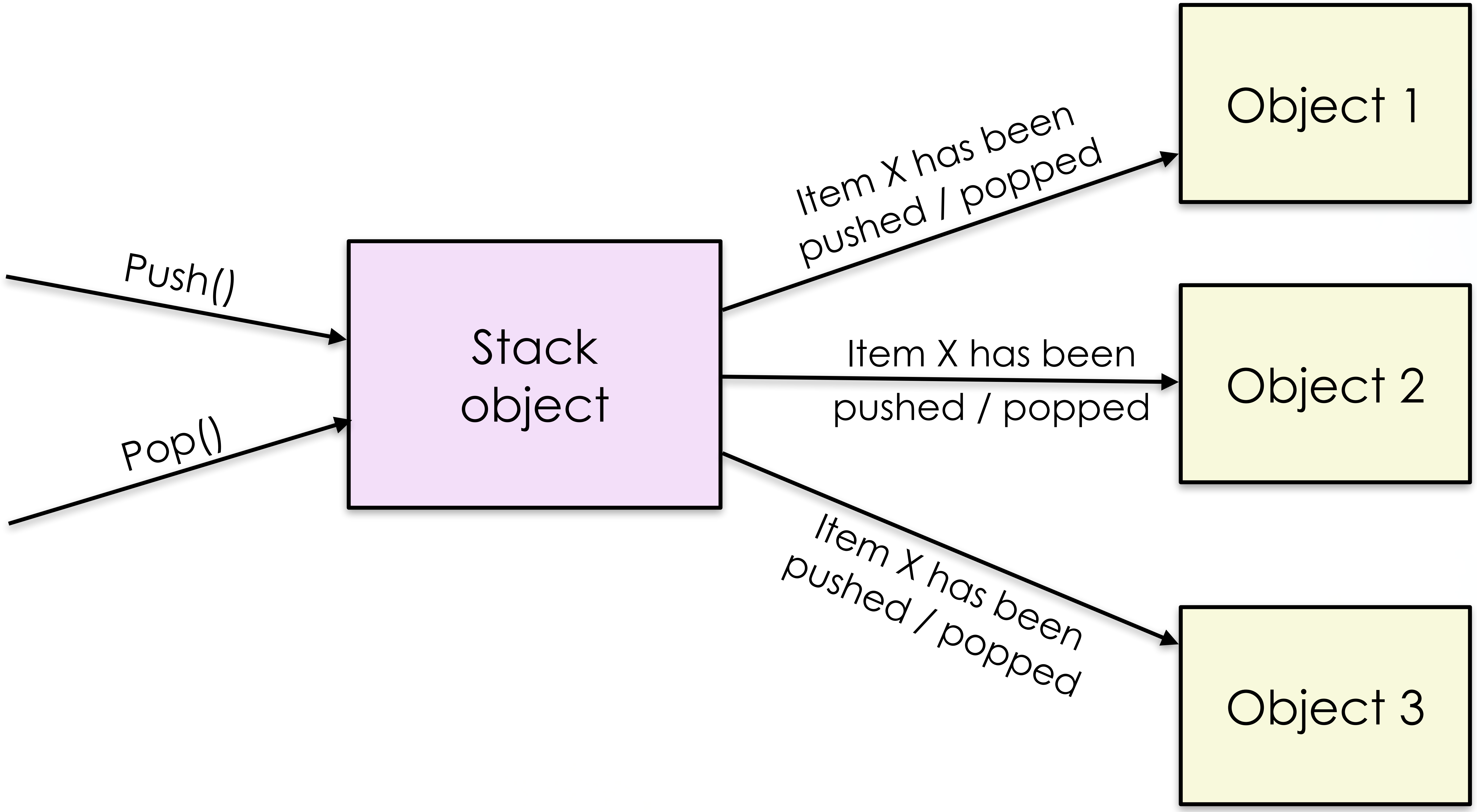
A Lesser Approach

The object (of interest) communicates directly with other objects whenever it has changes



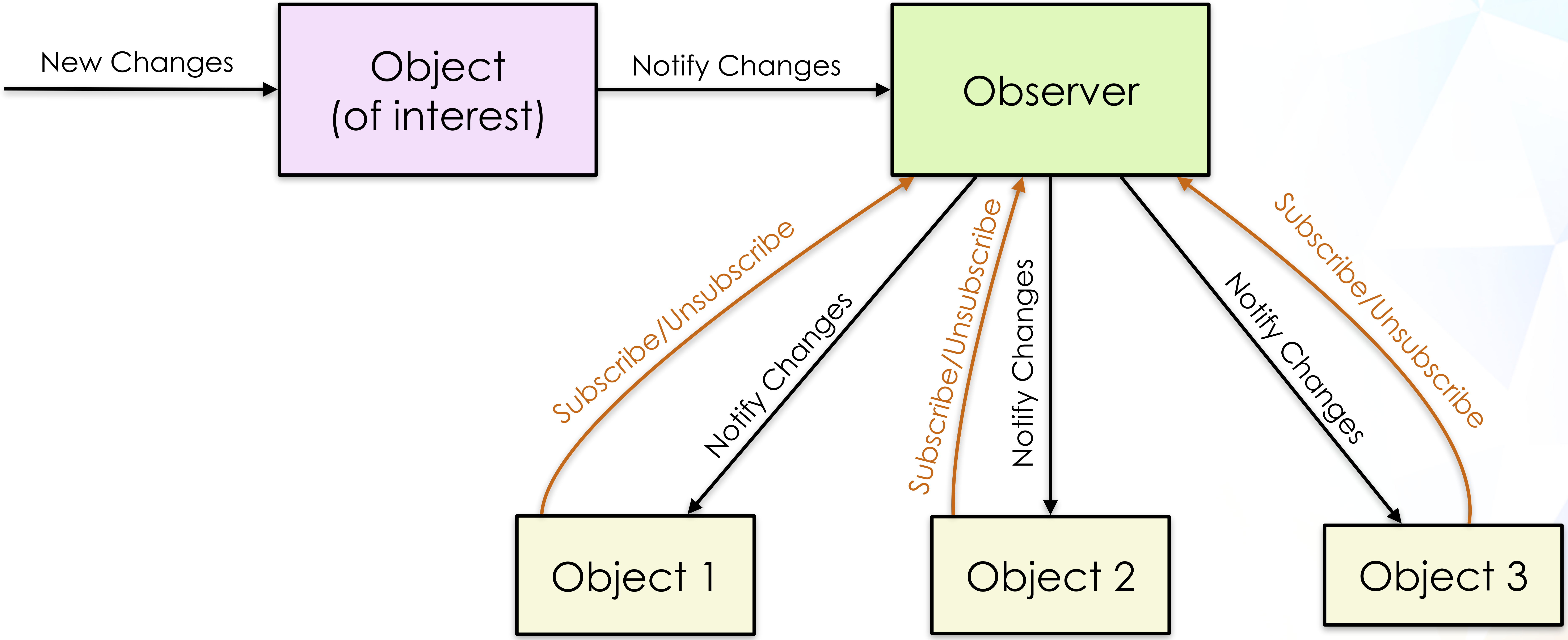
Sample Problem

Other objects are interested to be notified when new items have been added or removed to a Stack object


Push()
Pop()
Stack object
Item X has been pushed / popped
Item X has been pushed / popped
Item X has been pushed / popped
Object 1
Object 2
Object 3

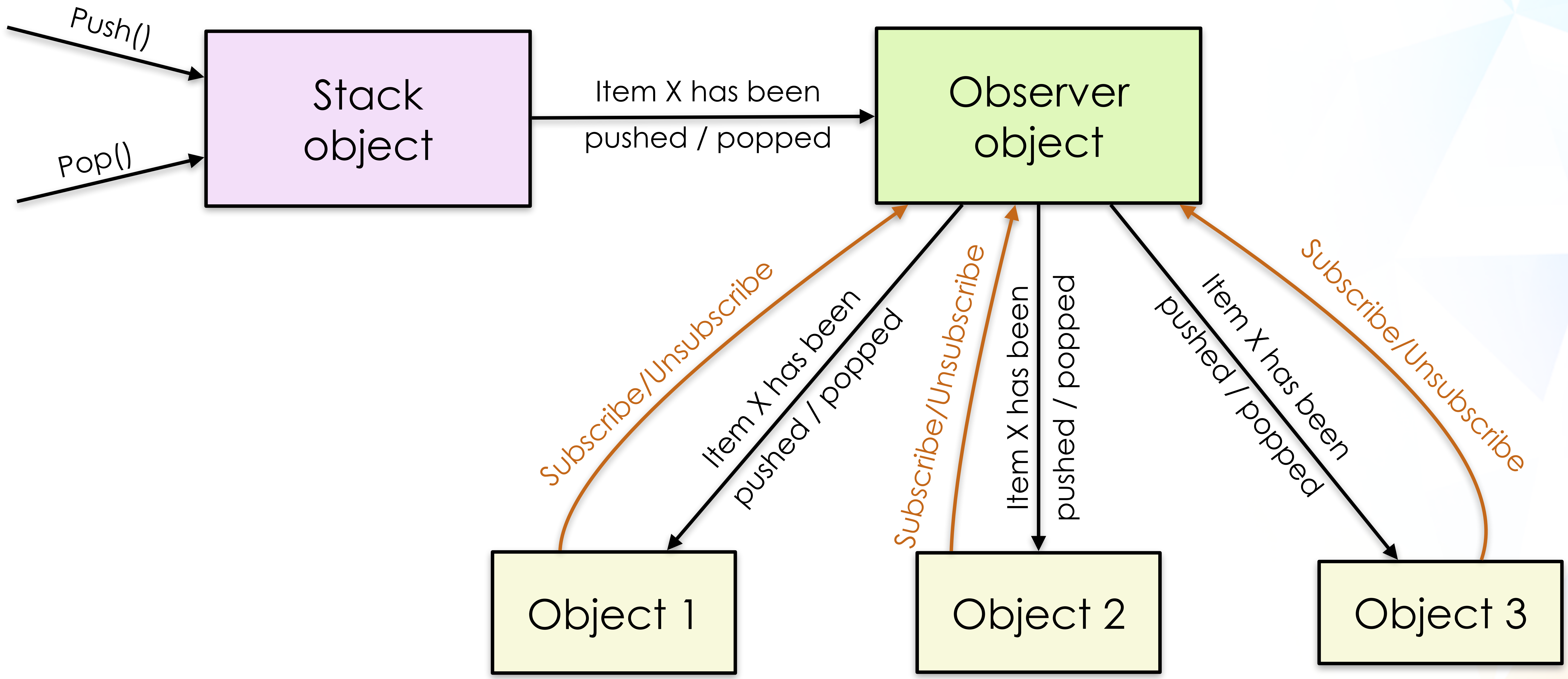
A Better Approach

The object (of interest) only communicates with an **Observer** whenever it has a change, and the Observer will communicate directly with the other objects



Problem Solved

The Stack object communicated only with an Observer object, which interacts with all other objects (listeners)



Go Implementation

- Review the Go implementation for the Observer design pattern (under the Observer folder)
- This includes running and debugging the code to understand its design.
- Additionally, inspect the code to determine
  - The key responsibilities of the Observer object
  - The object(s) that the Stack object communicates with
  - The action(s) that the Listener objects must take to be notified of changes made to a Stack object

Visitor

A Design Challenge

- Consider a system where we need to perform operations on a complex object structure and we need to separate the actions from the object that it operates on.

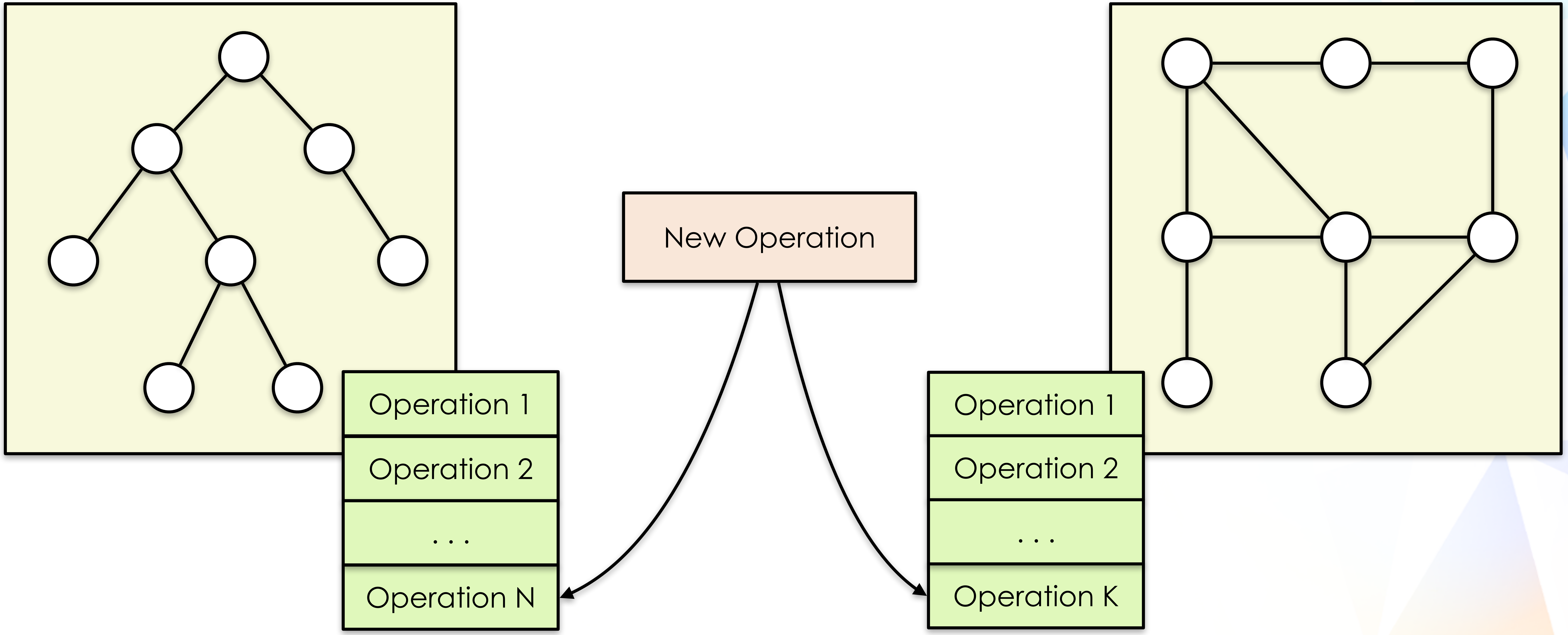
- The object structure can be frequently updated, and adding new operations or changing existing ones should not affect the structure itself.

- We need a way to add new operations to the system in a flexible, maintainable and scalable way, without having to change the object structure.

- Additionally, we want to avoid using a large number of if-else statements or switch-case statements to handle different types of objects.

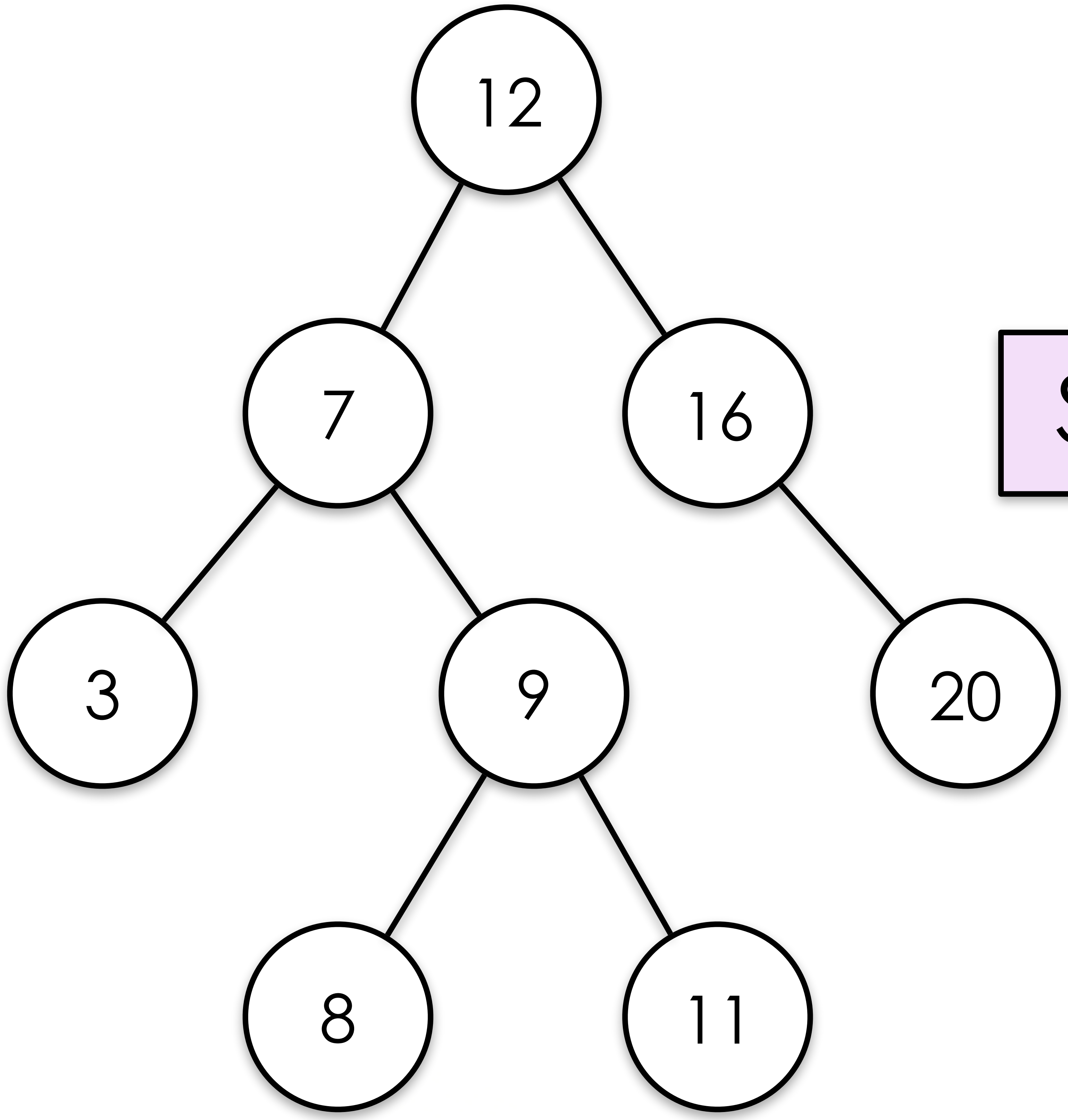
A Lesser Approach

Adapt the complex object structure to allow a new operation to be added to it



Sample Problem

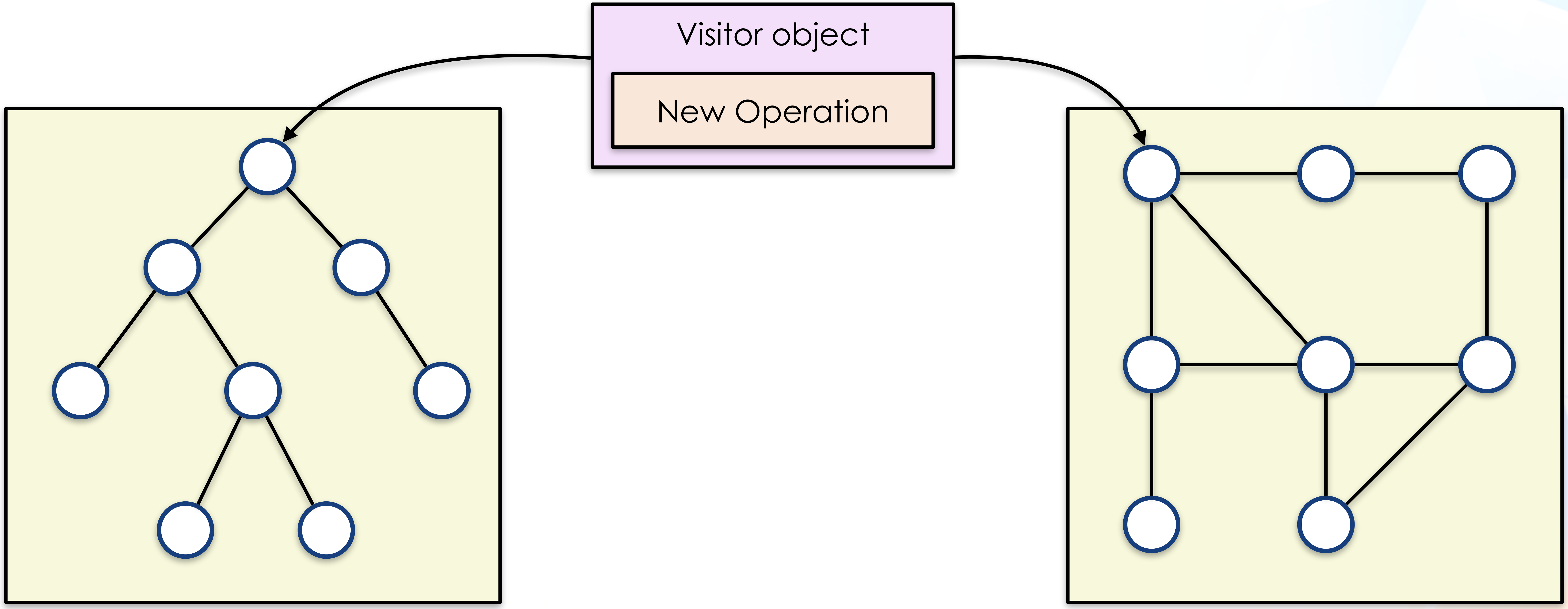
Add a new functionality that sums up the node-values in the following data structure without the need to know how to traverse it



Sum of Node Values?

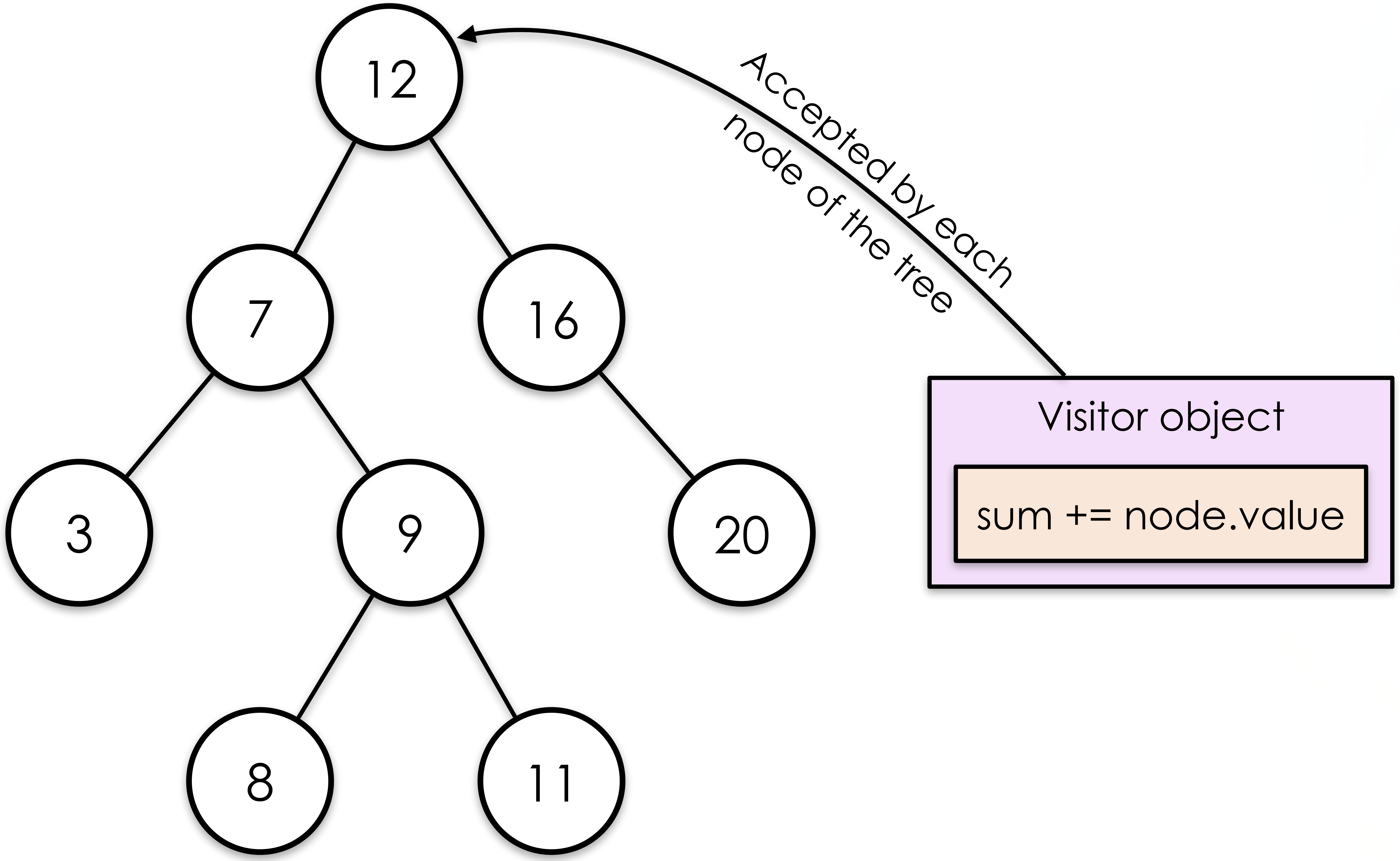
A Better Approach

Design it such that each node of the complex object structure accepts a "Visitor" object that contains the new operation



Problem Solved

Have the data structure accepts a Visitor object that visits every node and performs computation on it



Go Implementation

- Review the Go implementation for the Visitor design pattern (under the Visitor folder)

- This includes running and debugging the code to understand its design.

- Additionally, inspect the code to determine
  - The main role of the Visitor class
  - The object responsible for traversing the nodes of the binary search tree
  - The number of times the Accept method of the Visitor object is called during the execution of the program

THE END