Design Defects and Restructuring

LECTURE 12

SAT, DEC 05, 2020

Behavioral Patterns

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

Visitor

Mediator

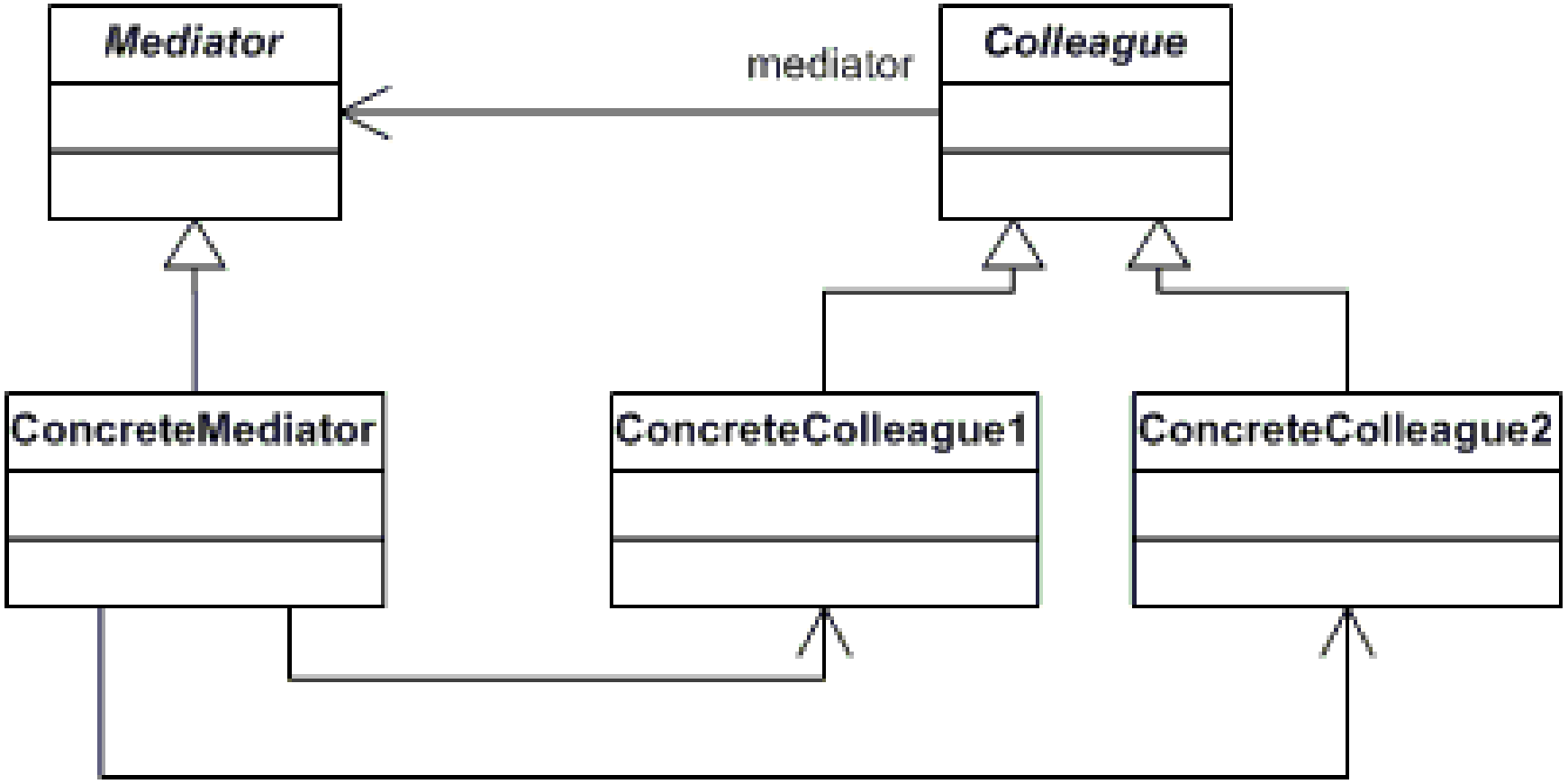
Intent

- Define an object that encapsulates how a set of objects interact
- Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently

Applicability

- A set of objects communicate in well-defined but complex ways
- Reusing an object is difficult because it refers to and communicates with many other objects
- A behavior that is distributed between several classes should be customizable without a lot of subclassing

Mediator


Mediator
mediator
Colleague
ConcreteMediator
ConcreteColleague1
ConcreteColleague2

Memento

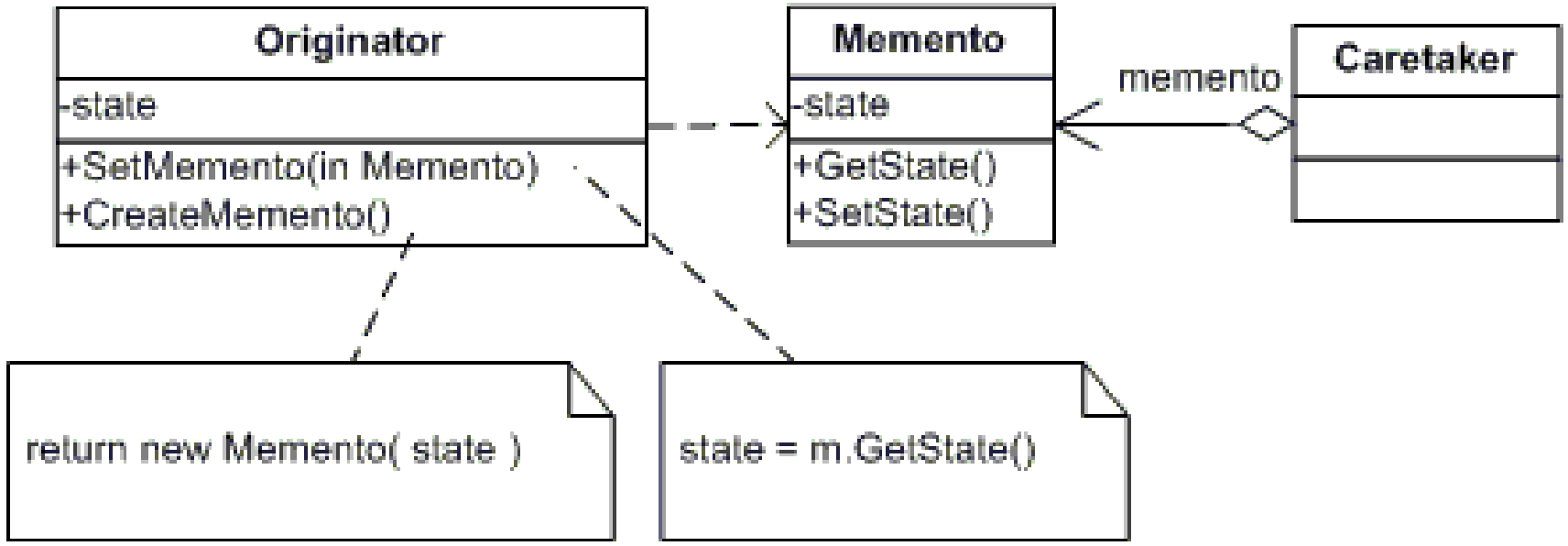
Intent

- Without violating encapsulation, capture and externalize an object’s internal state so that the object can be restored to this state later

Applicability

- A snapshot of (some portion of) an object’s state must be saved so that it can be restored to that state later
- A direct interface to obtaining the state would expose implementation details and break the object's encapsulation

Memento


Originator
-state
+SetMemento(in Memento)
+CreateMemento()
Memento
-state
+GetState()
+SetState()
memento
Caretaker
return new Memento(state)
state = m.GetState()

Observer

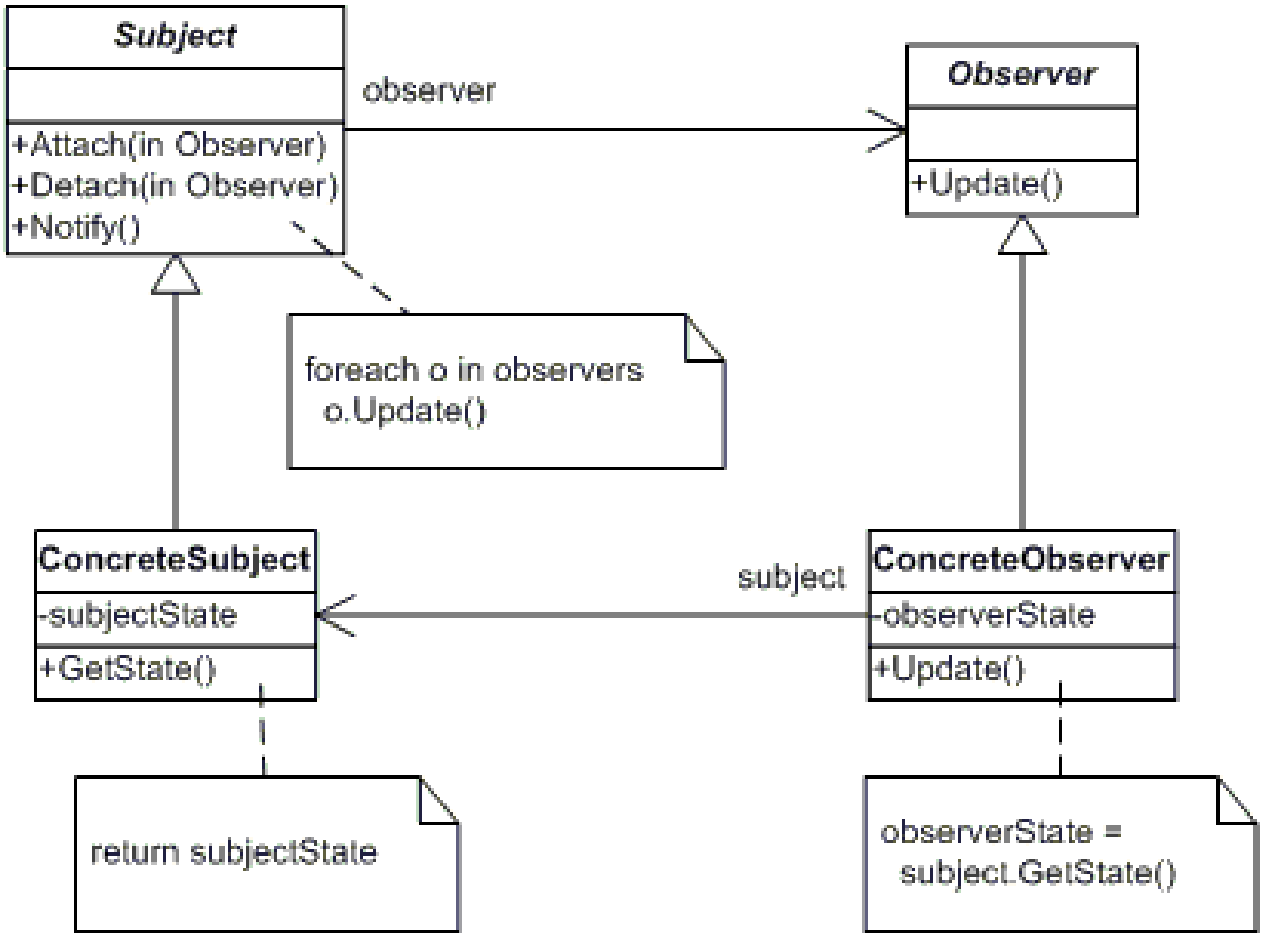
Intent

- Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically

Applicability

- When an abstraction has two aspects, one dependent on the other
- When a change to one object requires changing others, and you don’t know how many objects need to be changed
- When an object should be able to notify other objects without making assumptions about who these objects are

Observer


Subject
+Attach(in Observer)
+Detach(in Observer)
+Notify()
observer
Observer
+Update()
foreach o in observers
o.Update()
ConcreteSubject
-subjectState
+GetState()
subject
ConcreteObserver
-observerState
+Update()
return subjectState
observerState =
subject.GetState()

State

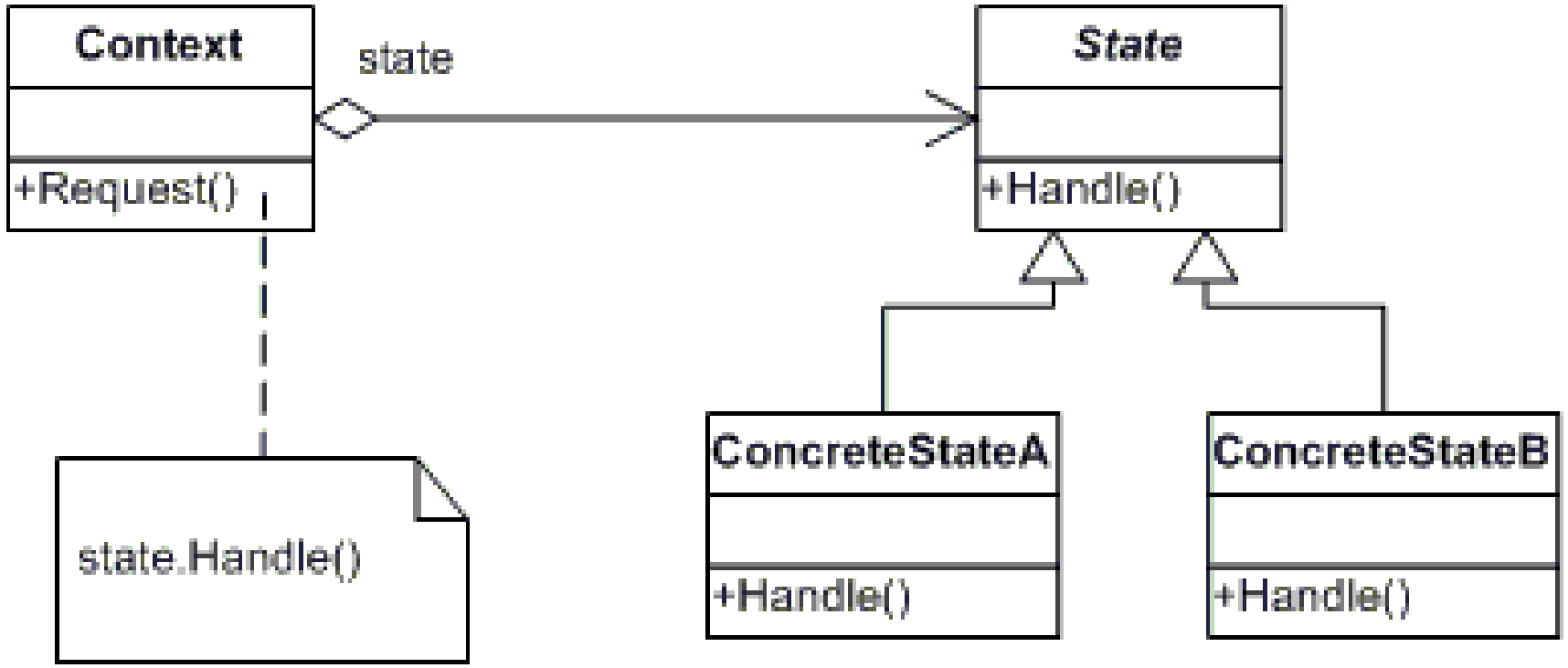
Intent

- Allow an object to alter its behavior when its internal state changes
- The object will appear to change its class

Applicability

- An object's behavior depends on its state, and it must change its behavior at run-time depending on that state
- Operations have large, multipart conditional statements that depend on the object's state

State


Context
state
State
+Request()
+Handle()
state.Handle()
ConcreteStateA
+Handle()
ConcreteStateB
+Handle()

Strategy

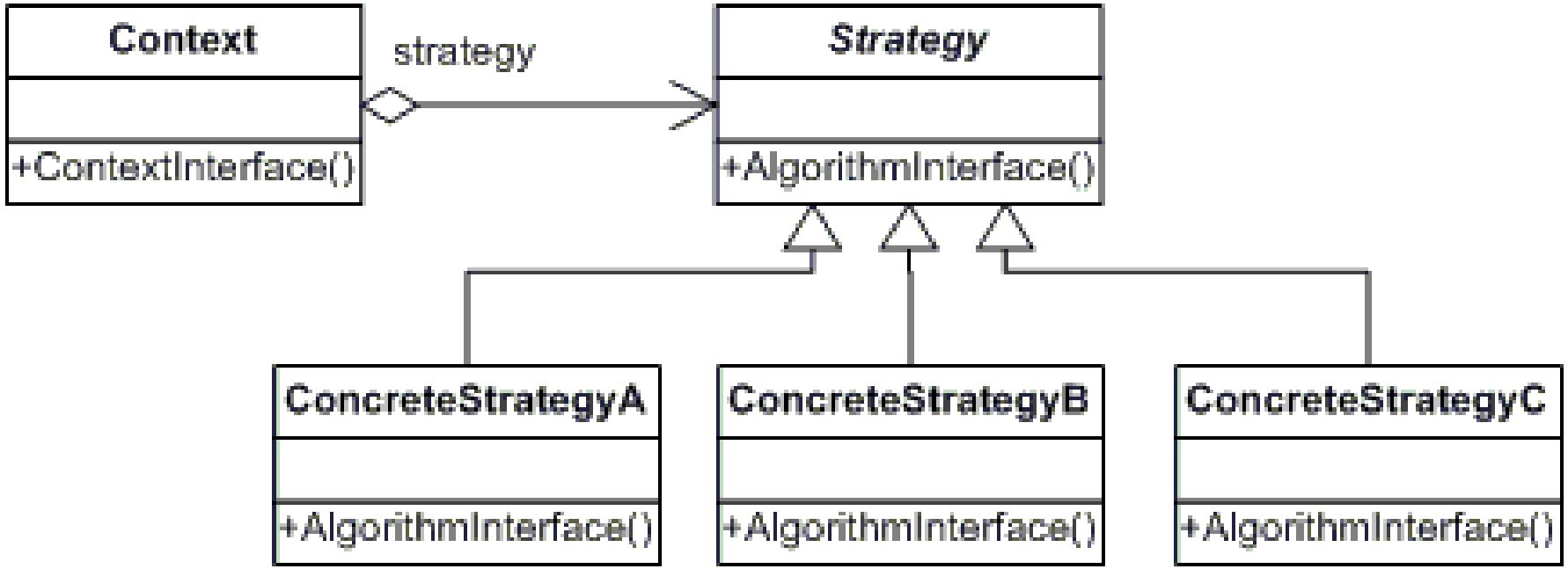
Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable
- Strategy lets the algorithm vary independently from clients that use it

Applicability

- Many related classes differ only in their behavior
  - Strategies provide a way to configure a class with one of many behaviors
- You need different variants of an algorithm
- An algorithm uses data that clients should not know about
  - Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures
- A class defines many behaviors, and these appear as multiple conditional statements in its operations
  - Instead of many conditionals, move related conditional branches into their own Strategy class

Strategy


Context
strategy
Strategy
+ContextInterface()
+AlgorithmInterface()
ConcreteStrategyA
+AlgorithmInterface()
ConcreteStrategyB
+AlgorithmInterface()
ConcreteStrategyC
+AlgorithmInterface()

Template Method

Intent

- Define the skeleton of an algorithm in an operation, deferring some steps to subclasses
- Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure

Applicability

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary
- When common behavior among subclasses should be factored and localized in a common class to avoid code duplication
- To control subclasses extensions
  - You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points

Template Method