# Design Defects and Restructuring

LECTURE 12

SAT, NOV 23, 2019

# Behavioral Patterns

Chain of Responsibility

Command

Interpreter

Iterator

Mediator

Memento

Observer

State

Strategy

Template Method

Visitor

# Strategy

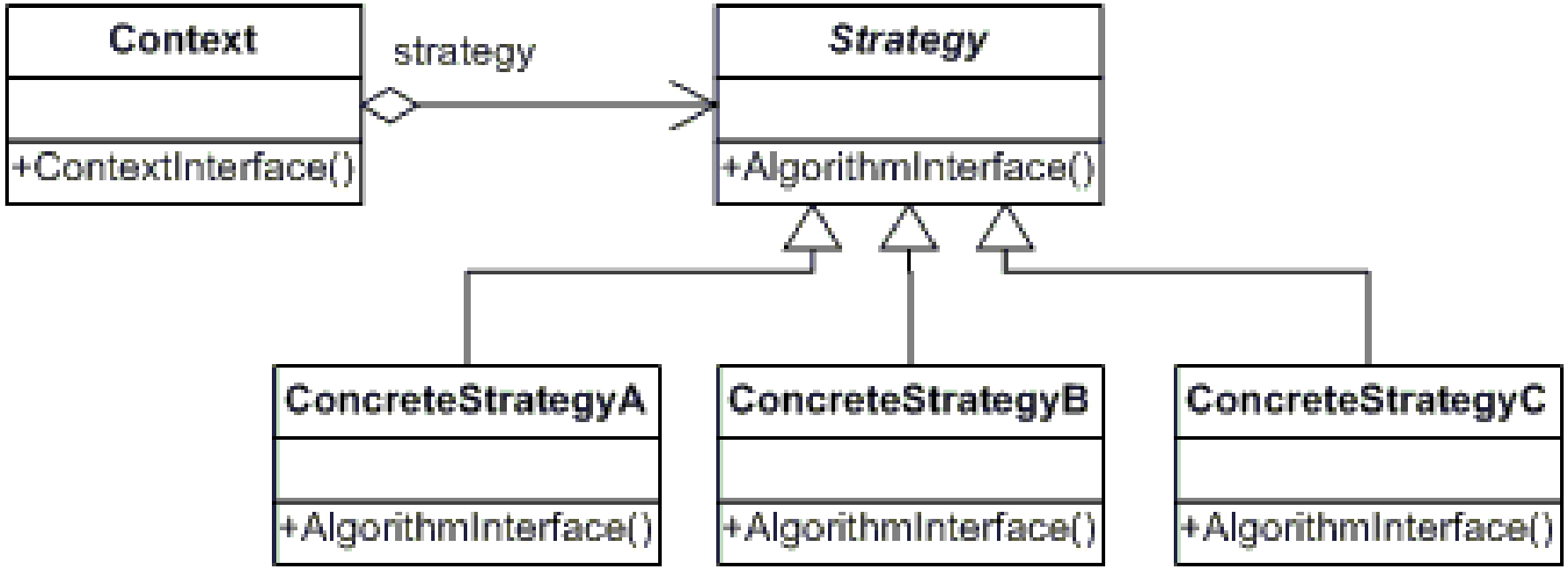
Intent

- 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
ConcreteStrategyB
ConcreteStrategyC
+AlgorithmInterface()
+AlgorithmInterface()
+AlgorithmInterface()

# Template Method

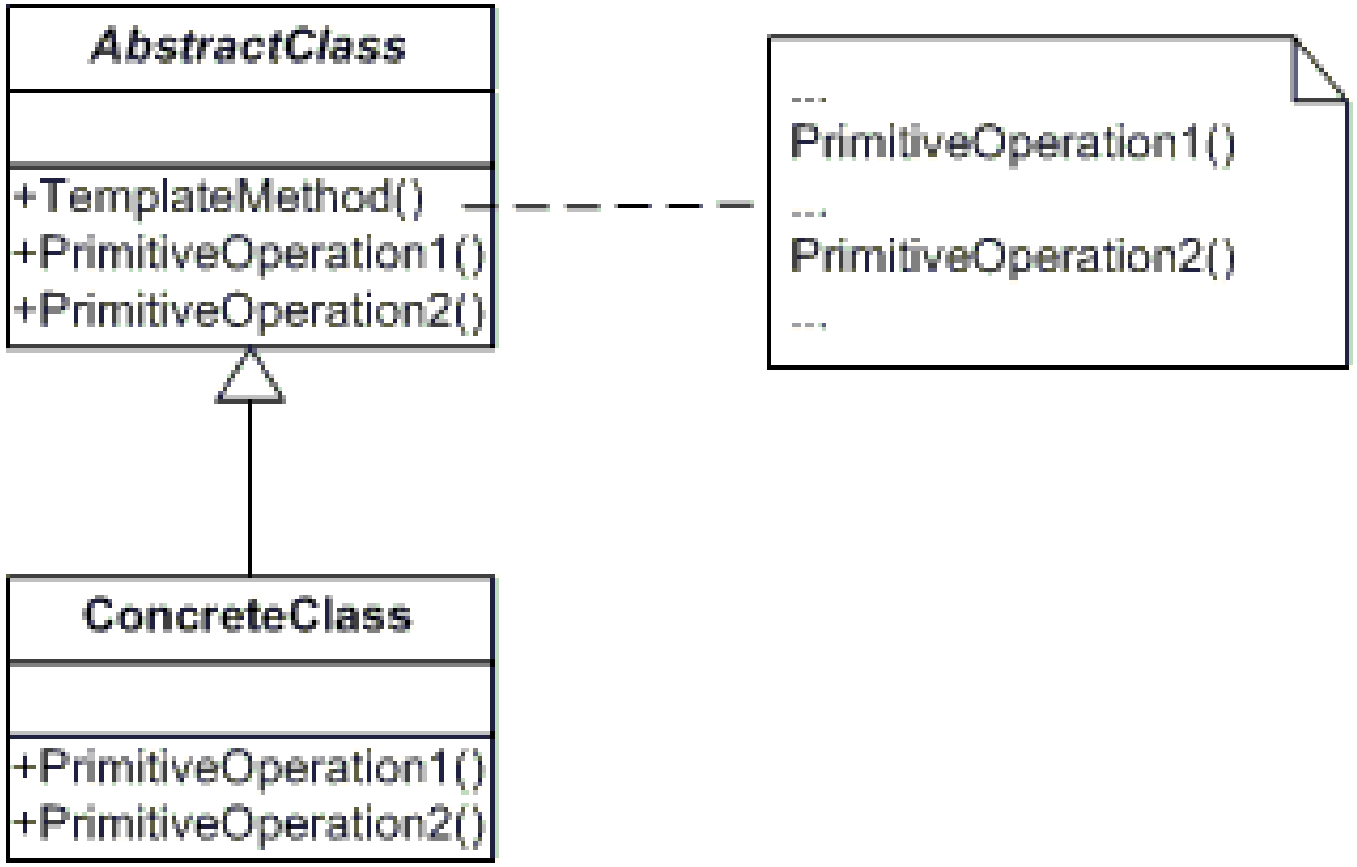
Intent

- 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



# Visitor

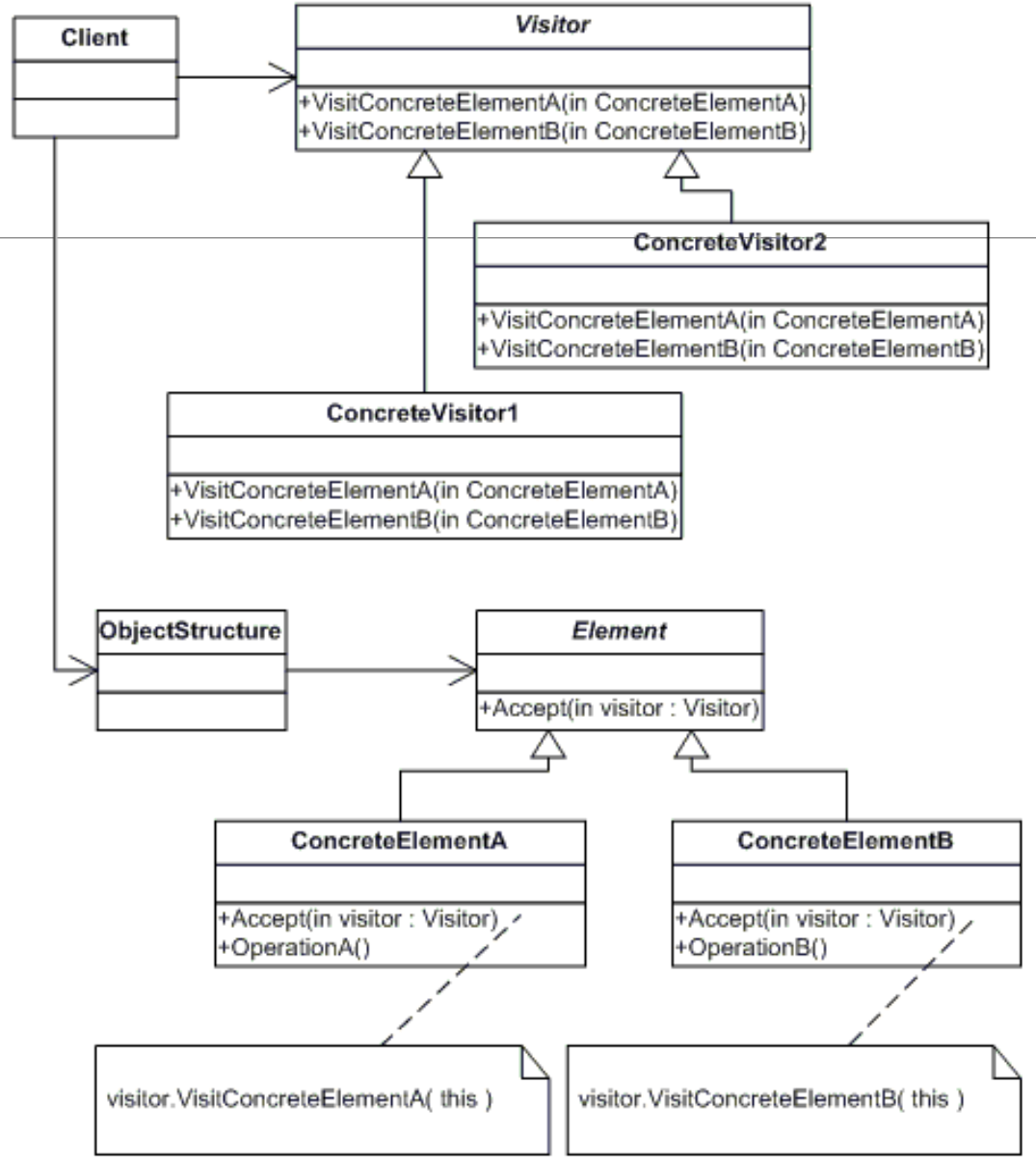
Intent

- Represent an operation to be performed on the elements of an object structure
- It lets you define a new operation without changing the classes of the elements on which it operates

Applicability

- An object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes
- Many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid “polluting” their classes with these operations
  - Visitor lets you keep related operations together by defining them in one class
  - When the object structure is shared by many applications, use Visitor to put operations in just those applications that need them
- The classes defining the object structure rarely change, but you often want to define new operations over the structure
  - Changing the object structure classes requires redefining the interface to all visitors, which is potentially costly
  - If the object structure classes change often, then it’s probably better to define the operations in those classes

# Visitor


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA( this )
visitor.VisitConcreteElementB( this )

# Design Patterns (Revisited)

Construction Patterns

- Abstract Factory
  - Provide for the creation of a family of related or dependent objects
- Builder
  - Move the construction logic for an object outside the class to instantiate, typically to allow piecemeal construction or to simplify the object
- Factory Method
  - Define an interface for creating an object while retaining control of which class to instantiate
- Prototype
  - Provide new objects by copying an example
- Memento
  - Provide for the storage and restoration of an object's state

# Design Patterns (Revisited)

Interface Patterns

- Adapter
  - Provide the interface that a client expects, using the services of a class with a different interface
- Façade
  - Provide an interface that makes a subsystem easy to use
- Composite
  - Allow clients to treat individual object and composition of objects uniformly
- Bridge
  - Decouple a class that relies on abstract operations from the implementation of those abstract operations so that the class and the implementation can vary independently

# Design Patterns (Revisited)

Extension Patterns

- Decorator
  - Let the developer compose an object's behavior dynamically
- Iterator
  - Provide a way to access the elements of a collection sequentially
- Visitor
  - Let the developer define a new operation for a hierarchy without changing the hierarchy classes

# Design Patterns (Revisited)

Responsibility Patterns

- Singleton
  - Ensure that a class has only one instance, and provide a global point of access to it
- Observer
  - Define a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
- Mediator
  - Define an object that encapsulates the way that a set of objects interact. This keeps the object from referring to each other explicitly and let you vary their interaction independently
- Proxy
  - Provide a placeholder for another object to control access to it
- Chain of Responsibility
  - Avoid coupling the sender  of request to its receiver, by giving more than one object chance to handle the request
- Flyweight
  - Use sharing to support large numbers of fine grained objects efficiently

# Design Patterns (Revisited)

## Operational Patterns

- Template Method
  - Implement an algorithm in a method, deferring the definition of some steps of the algorithm so that the other classes can supply them
- State
  - Distribute state specific logic across classes that represent an object's state
- Strategy
  - Encapsulate alternative strategies, or approaches, in separate classes that each implement a common operation
- Command
  - Encapsulate a request as an object, so that you can parameterize clients with different requests; queue, time or log requests; and allow a client to prepare a special context in which to invoke the request
- Interpreter
  - Let developers compose executable objects according to set of composition rules that you define