Software Design Patterns

Overview

- Design Patterns – Basics
- Structural Design Patterns
- Behavioral Design Patterns
- Appendix: More on the Observer Pattern
  More on the Strategy Pattern

Design Patterns

"*Each pattern describes*

a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice".

--- Christopher Alexander, 1977

This was in describing patterns in buildings and towns.

In SE, design patterns are in terms of objects and interfaces, not walls and doors.

The manner in which a collection of interacting objects collaborate to accomplish a specific task or provide some specific functionality.

Architecture vs. Design Patterns

Architecture

- High-level framework for structuring an application
  - “client-server based on remote procedure calls”
  - “abstraction layering”
  - “distributed object-oriented system based on CORBA”
- Defines the system in terms of computational components & their interactions

Design Patterns

- Lower level than architectures (Sometimes, called *micro-architecture*)
- Reusable collaborations that solve subproblems within an application
  - how can I decouple subsystem X from subsystem Y?

Why Design Patterns?

- Design patterns support *object-oriented reuse* at a high level of abstraction
- Design patterns provide a “framework” that guides and constrains object-oriented implementation

4 Essential Elements of Design Patterns

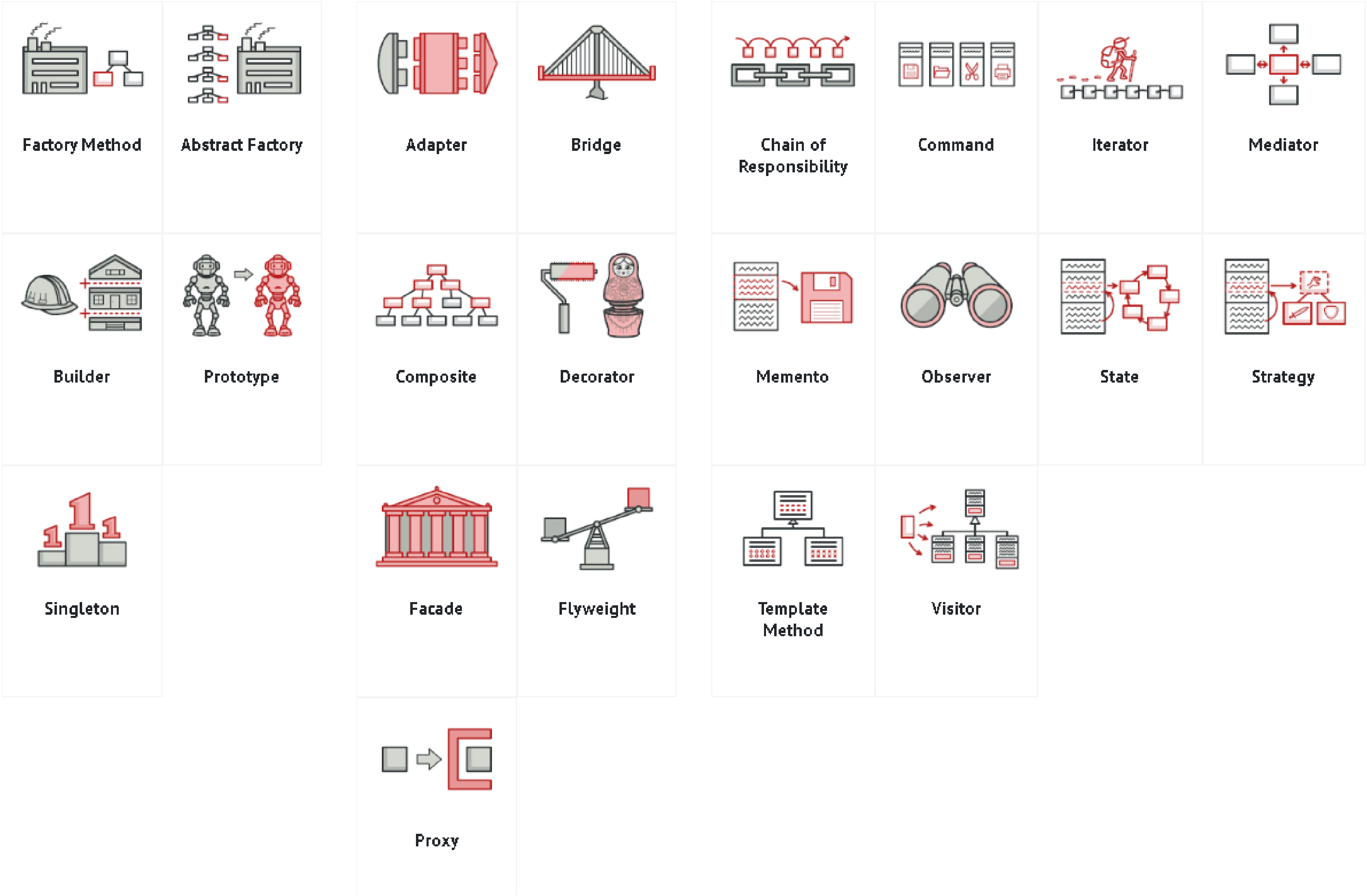
- *Name:* identifies a pattern
- *Problem*: describes when to apply the pattern in terms of the problem and context
- *Solution*: describes elements that make up the design, their relationships, responsibilities, and collaborations
- *Consequences*: results and trade-offs of applying the pattern

Organizing Design Patterns

- By *Purpose* (reflects what a pattern does):
  - Creational Patterns
  - Structural Patterns
  - Behavioral Patterns

- By *Scope*: specifies whether the pattern applies primarily to
  - *classes* or to
  - *objects*.

Design Pattern Overview



Factory Method

Abstract Factory

Adapter

Bridge

Chain of Responsibility

Command

Iterator

Mediator

Builder

Prototype

Composite

Decorator

Memento

Observer

State

Strategy

Singleton

Facade

Flyweight

Template Method

Visitor

Proxy

Design Patterns Space

		Purpose		
		Creational	Structural	Behavioral
Scope	Class	Factory Method	Adapter	Interpreter Template
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational Patterns

- Factory
- Builder
- Singelton

Creational Patterns
Factory

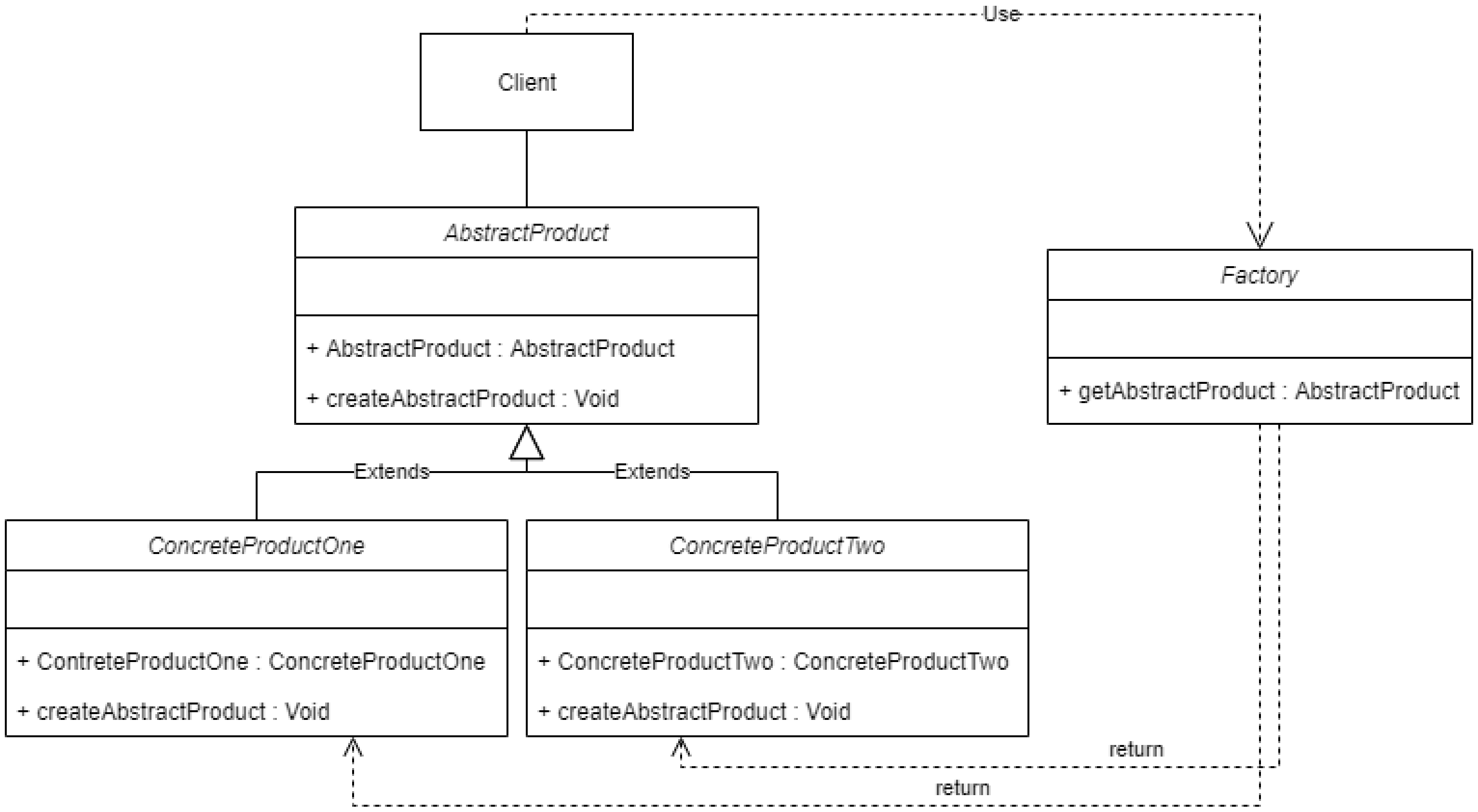
Intent

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Applicability

- A class cannot anticipate the class of objects it must create.
- A class wants its subclasses to specify the objects it creates.
- Classes delegate responsibility to one of several helper subclasses, and you want to localize the knowledge of which helper subclass is the delegate.

Creational Patterns
Factory


Use
Client
AbstractProduct
+ AbstractProduct : AbstractProduct
+ createAbstractProduct : Void
Factory
+ getAbstractProduct : AbstractProduct
Extends
Extends
ConcreteProductOne
+ ContreteProductOne : ConcreteProductOne
+ createAbstractProduct : Void
ConcreteProductTwo
+ ConcreteProductTwo : ConcreteProductTwo
+ createAbstractProduct : Void
return
return

Lab

Lab d-07

Creational Patterns
Builder

Intent

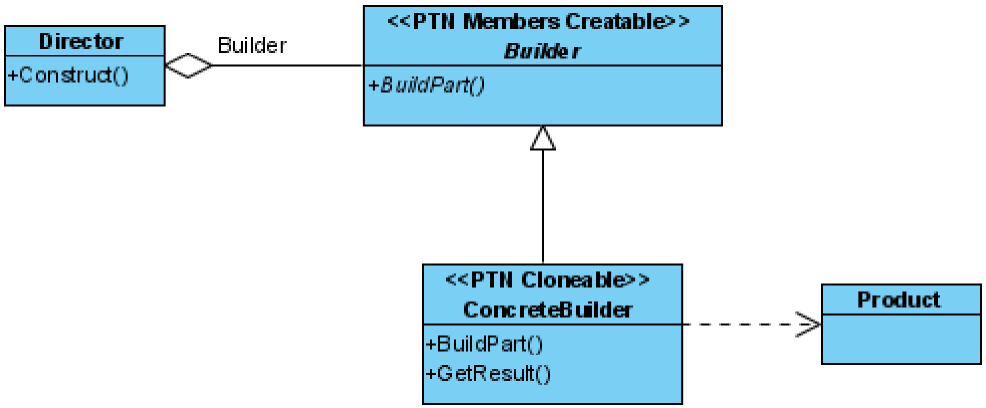
Separate the construction of a complex object from its representation so that the same construction process can create different representations.

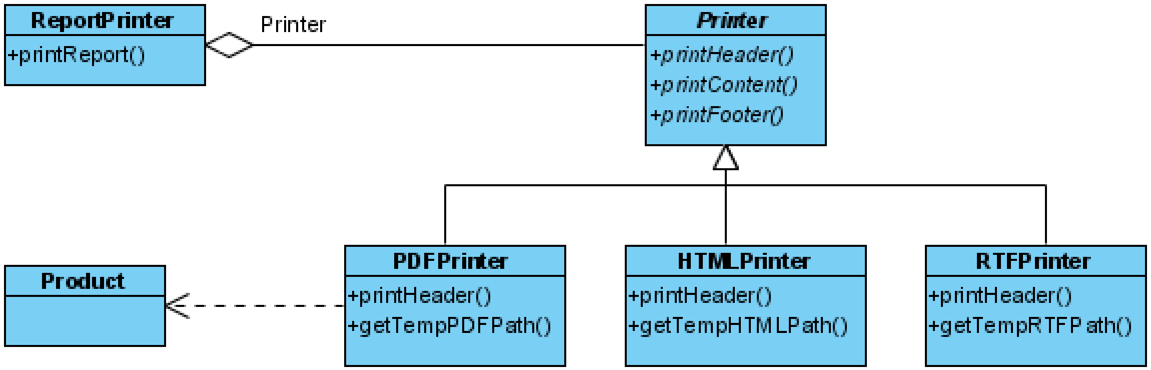
Applicability

- The algorithm for creating a complex object should be independent of the parts that make up the object and how they are assembled.
- The construction process must allow different representations of the object being built.
- You want to isolate complex construction logic from the object's final representation to make the code easier to maintain and extend.

Creational Patterns

Builder





Lab

Lab d-08

Creational Patterns
Singelton

Intent

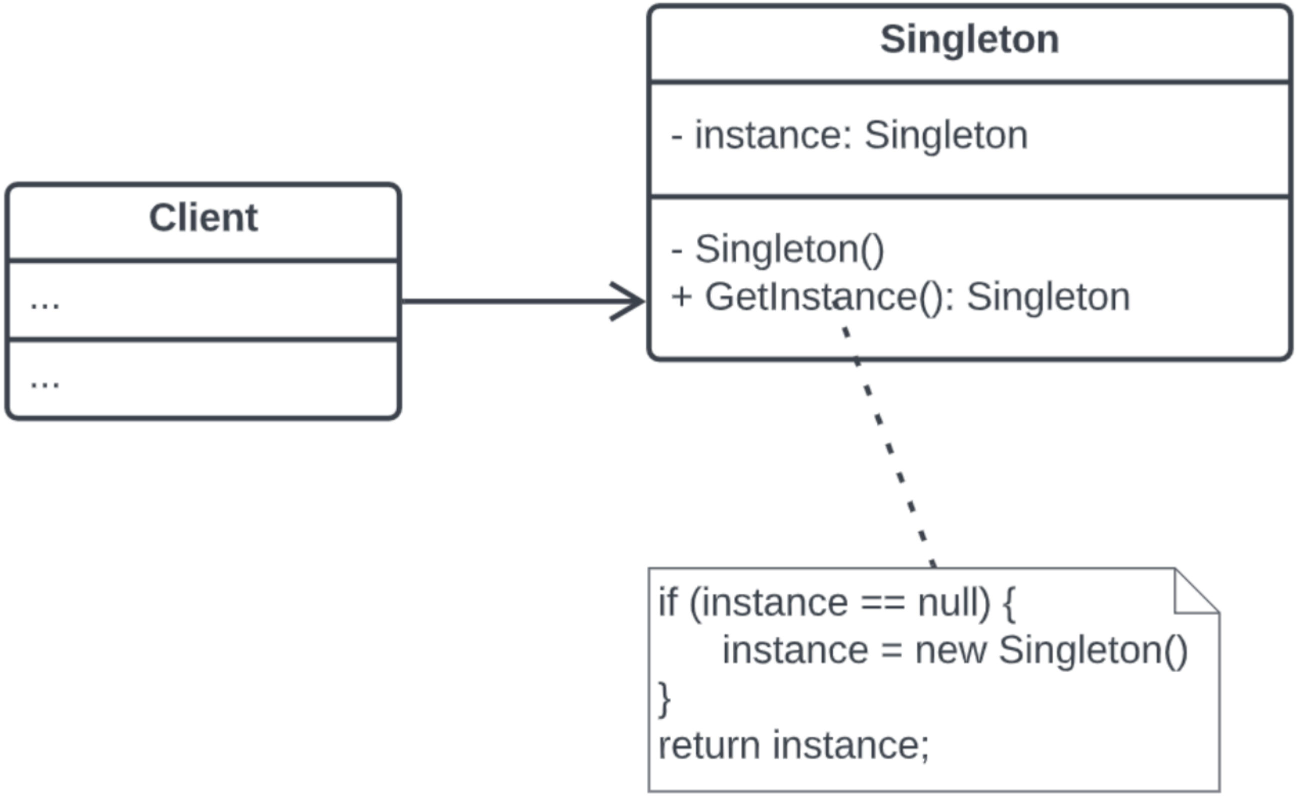
Ensure a class has only one instance and provide a global point of access to it.

Applicability

- There must be exactly one instance of a class, and it must be accessible to clients from a well-known access point.
- The single instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code.
- You want to control access to shared resources, such as configuration settings, logging, or connection pools.

Creational Patterns

Singelton



Client

...

...

Singleton

- instance: Singleton

- Singleton()

+ GetInstance(): Singleton

```
if (instance == null) {
      instance = new Singleton()
}
return instance;
```

Structural Patterns

- Composite
- Adapter
- Facade
- Proxy

Structural Patterns
Composite

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

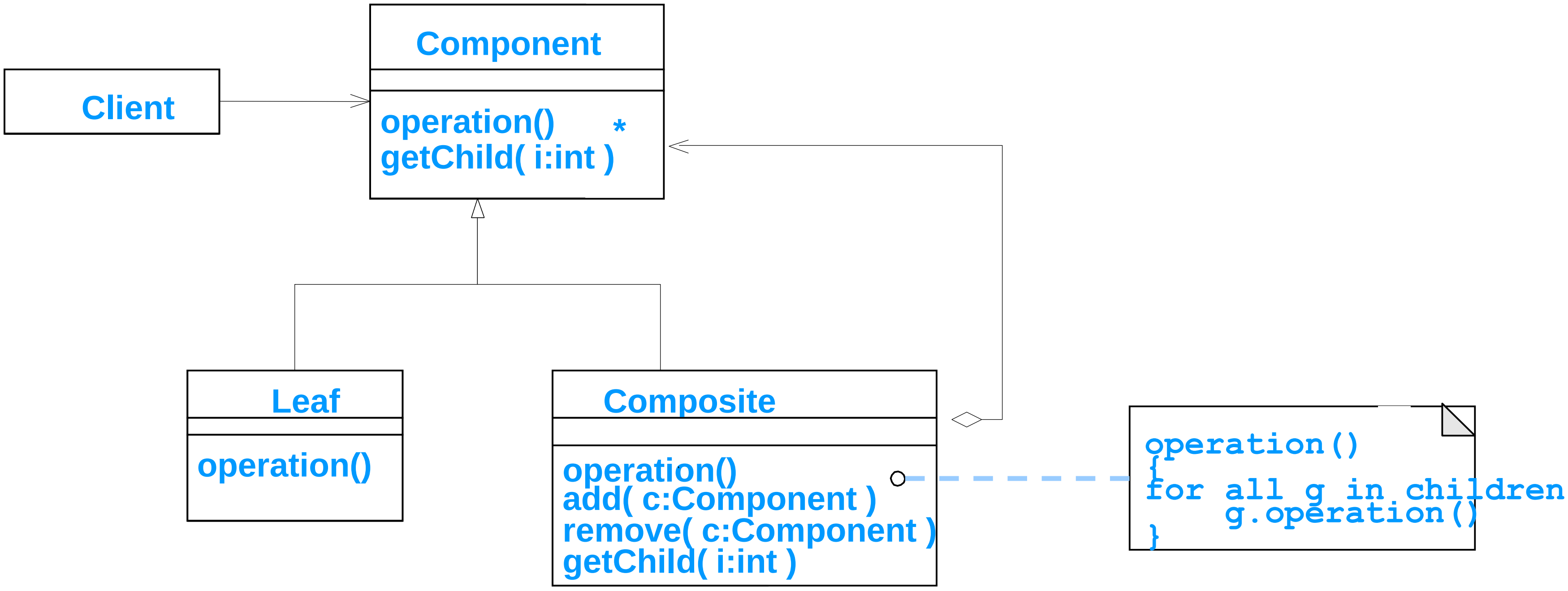
Composite: Applicability

- Represents part-whole hierarchies of objects.
- Clients ignore the difference between compositions of objects and individual objects.
- Clients treat all objects in the composite structure uniformly.

Structural Patterns

Composite

Class Diagram



Client

Component

operation() *
getChild(i:int)

Leaf

operation()

Composite

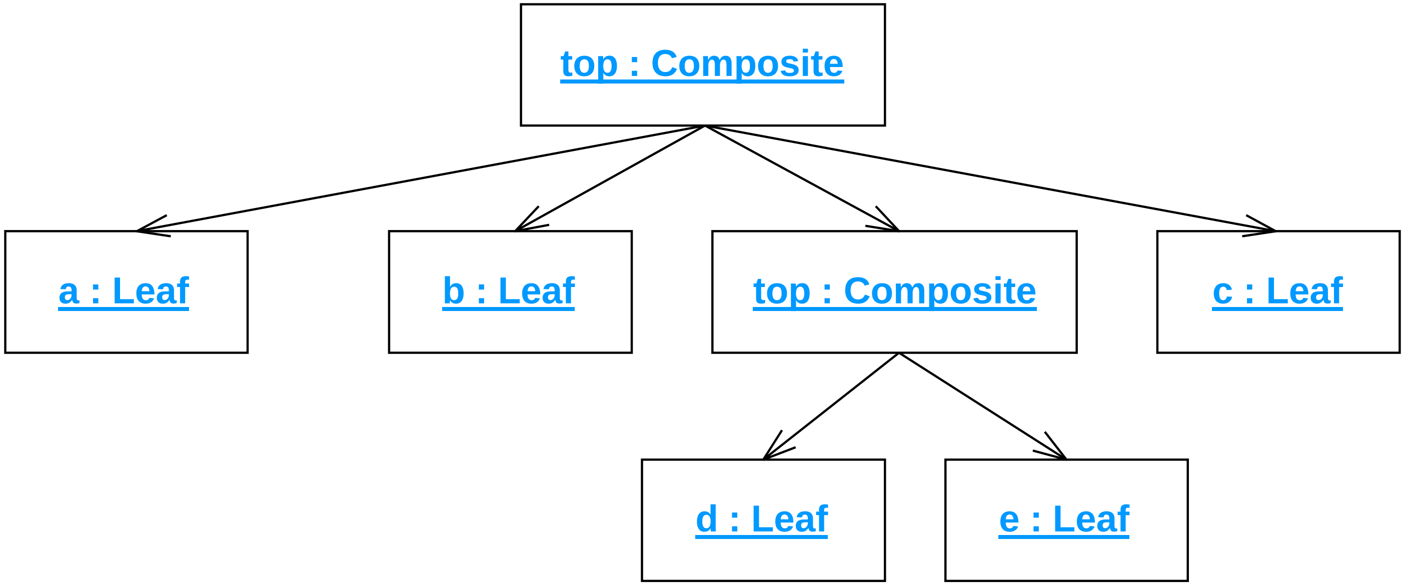
operation()
add(c:Component)
remove(c:Component)
getChild(i:int)

```
operation()
{
for all g in children
    g.operation()
}
```

Structural Patterns

Composite

Object Diagram



Structural Patterns

Composite

Participants

Component

- Declares the interface for objects in the composition.
- Implements default behavior for the interface common to all classes, as appropriate.
- Declares an interface for accessing and managing its child components.
- Optionally defines an interface for accessing a components parent.

Leaf

- Represents leaf objects in the composition.
- Defines behavior for primitive objects in the composition.

Composite

- Defines behavior for components having children.
- Stores child components.
- Implements child-related operations.

Client

- Manipulates objects in the composition through the Component interface.

Structural Patterns

Composite

Collaborations

- Clients use the Component class interface to interact with objects in the composite structure.
- If the recipient is a Leaf, then the request is handled directly.
- If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Lab

Lab d-01

Structural Patterns

Adapter

Intent

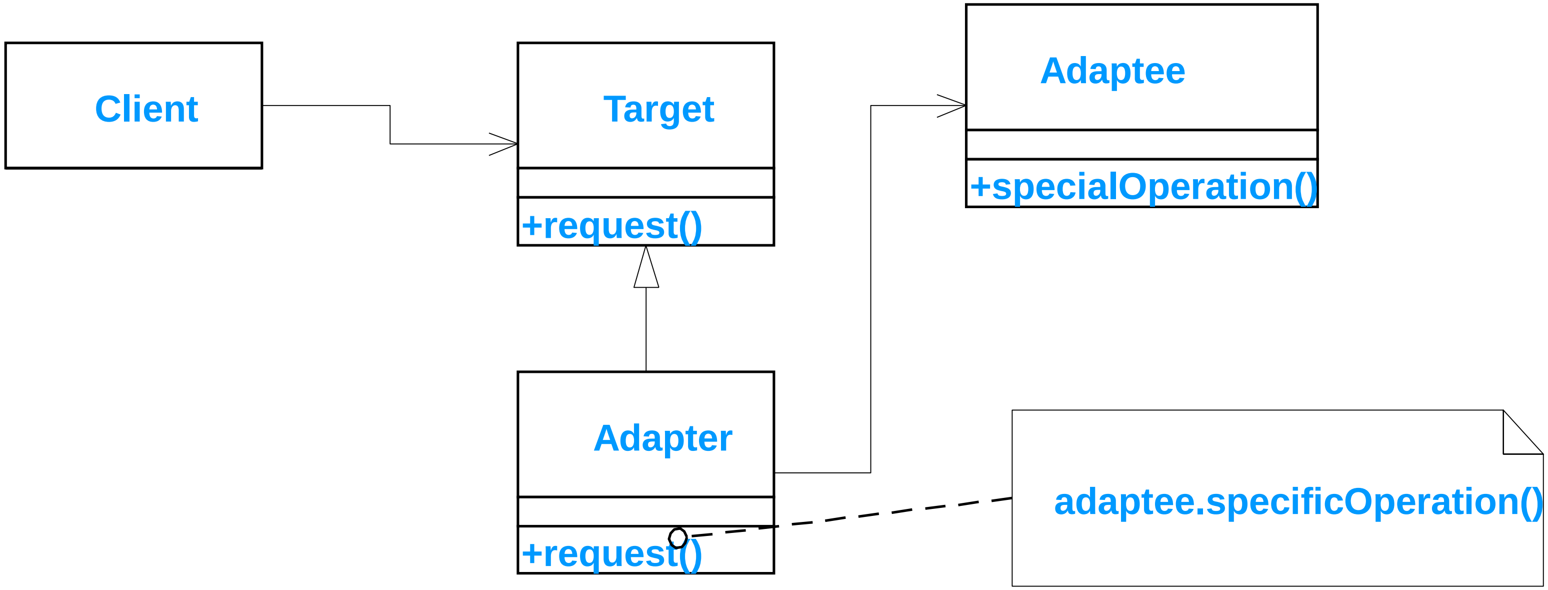
Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

Applicability

- Reuse of an existing class is desired, but the interface does not match the need.
- Design of a reusable class that cooperates with unrelated or unforeseen classes, but classes don't have compatible interfaces.

Structural Patterns
Adapter

Class Diagram



Structural Patterns

Adapter

Participants

- Target — defines the domain-specific interface that the client uses.
- Client — collaborates with objects conforming to the Target interface.
- Adaptee — defines an existing interface that needs adapting.
- Adapter — adapts the interface of Adaptee to the Target interface.

Collaborations

- Clients call operations on an Adapter instance. In turn, the Adapter calls Adaptee operations that carry out the request.

Lab

Lab d-02

Structural Patterns
Facade

Intent

Provide a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use.

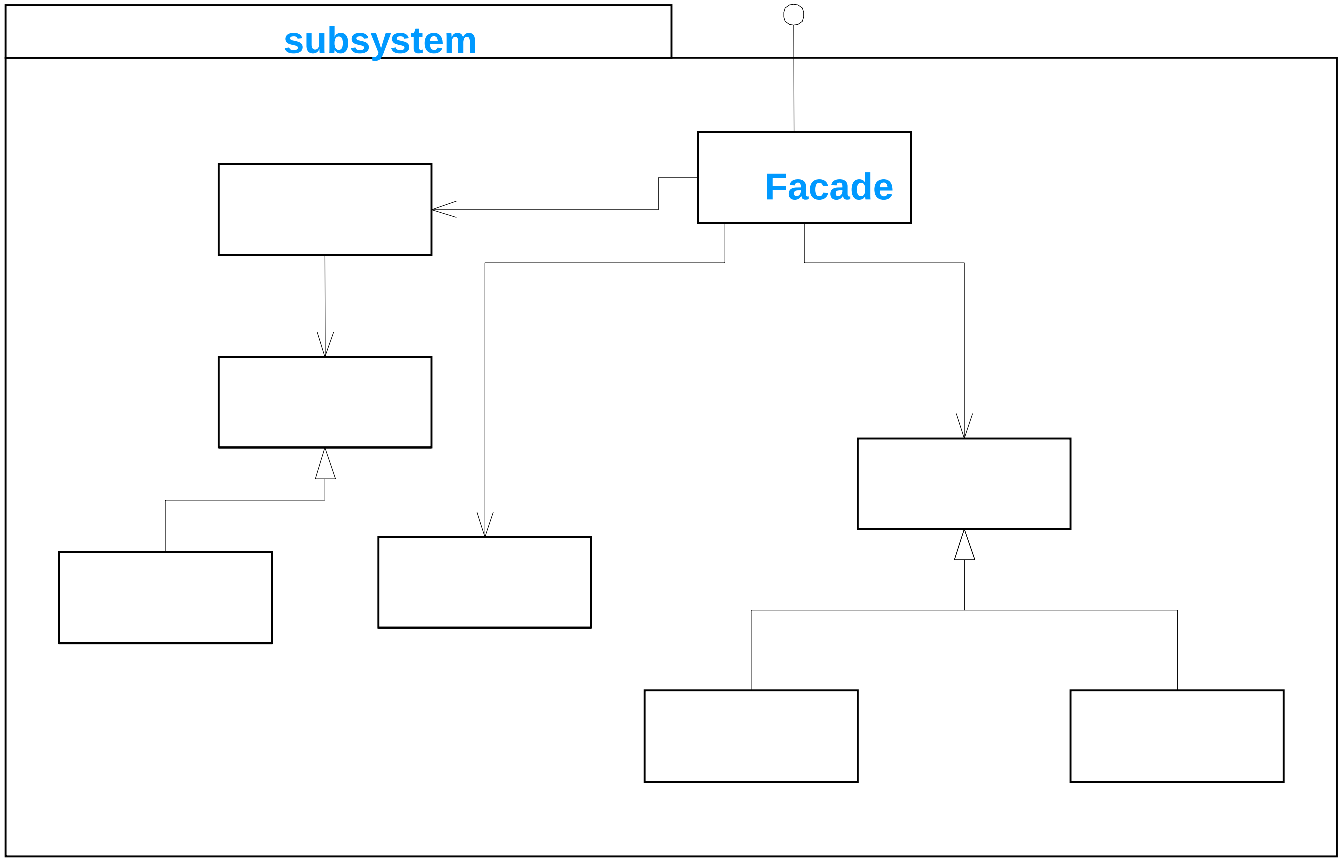
Applicability

- Provides a simple interface to a complex subsystem.
- Decouples the details of a subsystem from clients and other subsystems.
- Provides a layered approach to subsystems.

Structural Patterns

Facade

Class Diagram



Structural Patterns
Facade

Participants

- Façade
  - Knows which classes are responsible for each request.
  - Delegates client requests to appropriate objects.
- Subsystem classes
  - Implement subsystem functionality.
  - Handle work assigned by the Façade object.
  - Have no knowledge of the façade.

Collaborations

- Clients communicate with the subsystem sending requests to the Façade.
  - Reduces the number of classes the client deals with.
  - Simplifies the subsystem.
- Clients do not have to access subsystem objects directly.

Lab

Lab d-03

Structural Patterns

Proxy

Intent

Provide a surrogate or placeholder for another object to control access to it.

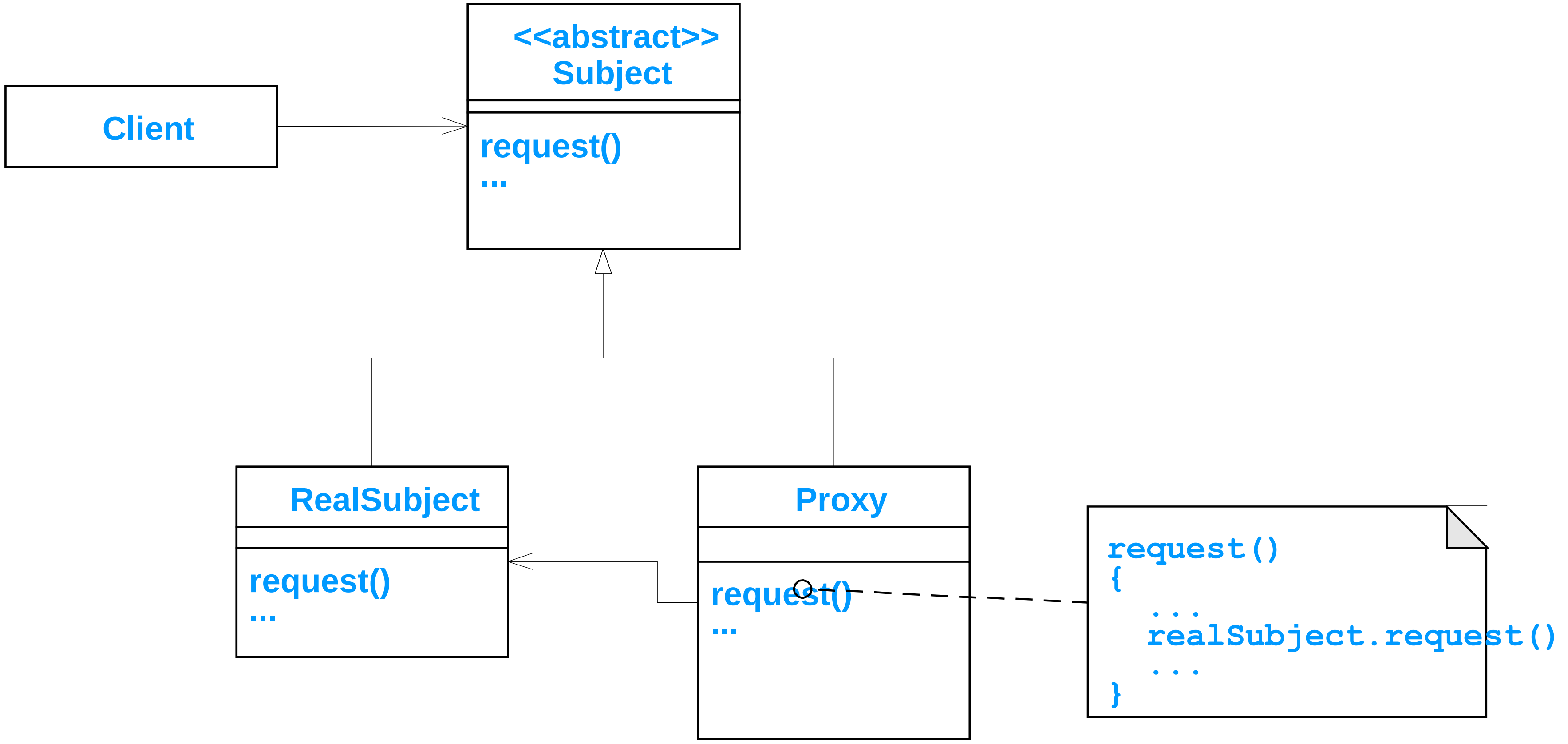
Applicability

- Remote proxy — provides a local representative for an object in a different address space.
- Virtual proxy — creates expensive objects on demand.
- Protection proxy — controls access to the original object.
- Smart reference — replacement for a bare pointer
  - Reference counting
  - Loading persistent object on access
  - Transactional locking

Structural Patterns

Proxy

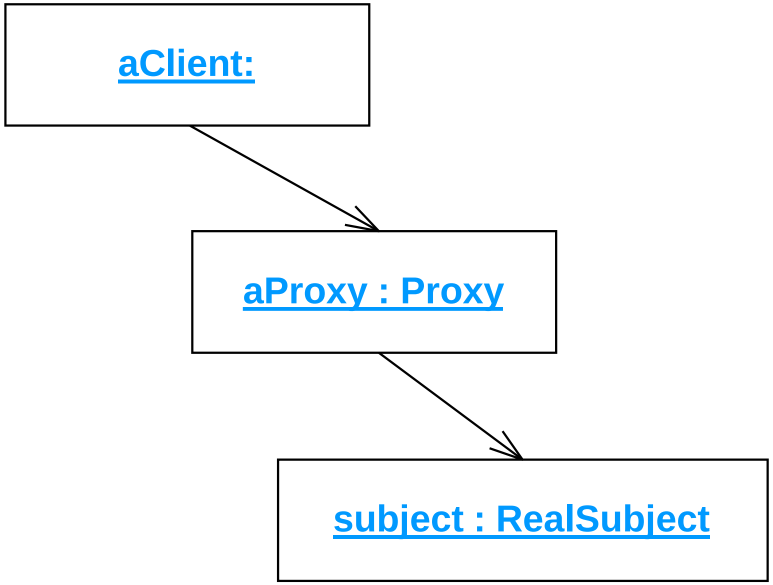
Class Diagram



Structural Patterns

Proxy

Object Diagram



Structural Patterns

Proxy

Participants

- Subject: Defines the common interface for RealSubject and Proxy.
- Proxy:
  - Maintains reference to real subject
  - Can be substituted for a real subject
  - Controls access to real subject
  - May be responsible for creating and deleting the real subject
  - Special responsibilities
    - Marshaling for remote communication
    - Caching data
    - Access validation
- RealSubject: Defines the real object that the proxy represents.
- Client: Accesses the RealSubject through the intervention of the Proxy.

Collaborations

- Proxy forwards requests to RealSubject when appropriate, depending on the kind of proxy.

Lab

Lab d-04

Behavioral Patterns

- Observer
- Strategy
- Command
- State
- Visitor

Behavioral Patterns
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.

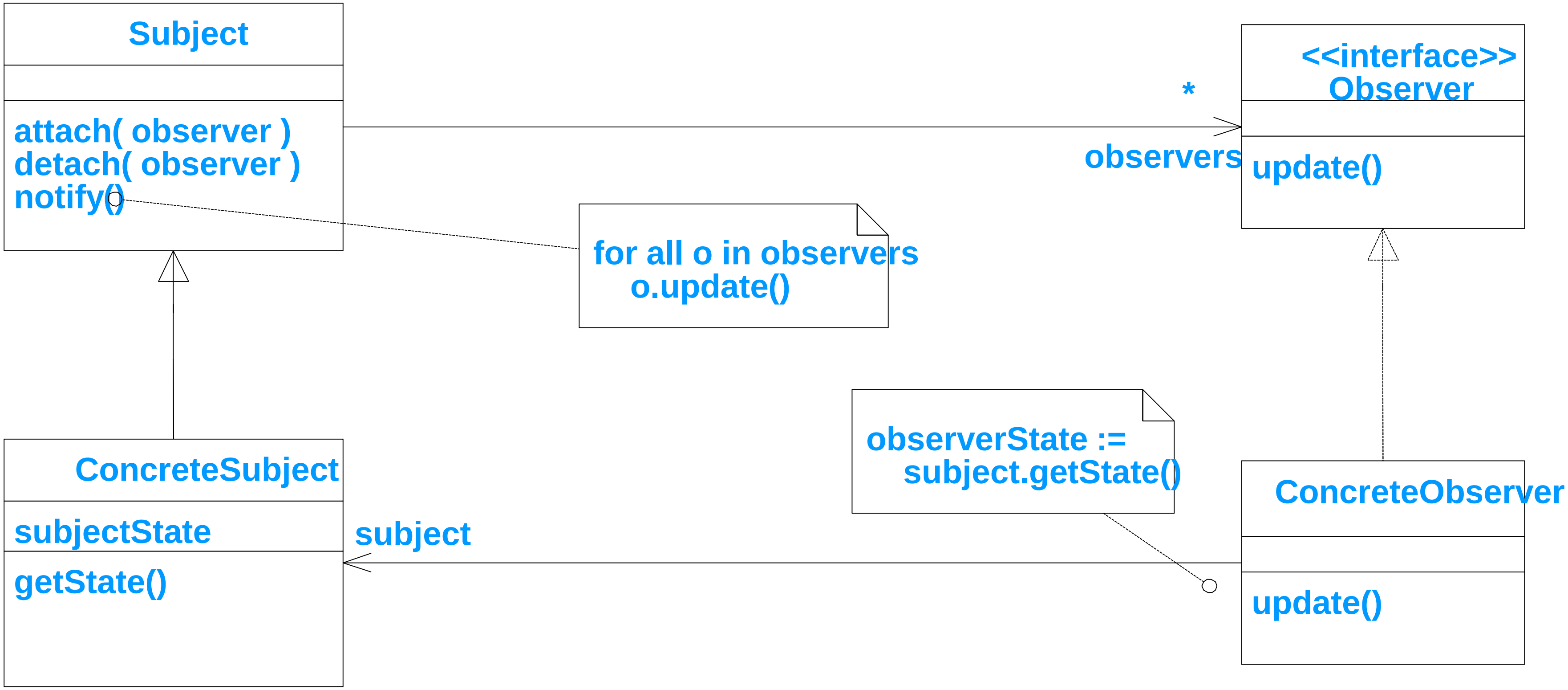
Applicability

- An abstraction has two aspects, one dependent on the other.
- When changing one object requires changing others, and you don’t know how many objects need changed.
- When an object needs to notify others without knowledge about who they are.

Behavioral Patterns

Observer

Class Diagram



Behavioral Patterns

Observer

Participants

- Subject
  - Knows its observers, but not their “real” identity.
  - Provides an interface for attaching/detaching observers.
- Observer
  - Defines an updating interface for objects that should be identified of changes.
- ConcreteSubject
  - Stores state of interest to ConcreteObserver objects.
  - Sends update notice to observers upon state change.
- ConcreteObserver
  - Maintains reference to ConcreteSubject (sometimes).
  - Maintains state that must be consistent with ConcreteSubject.
  - Implements the Observer interface.

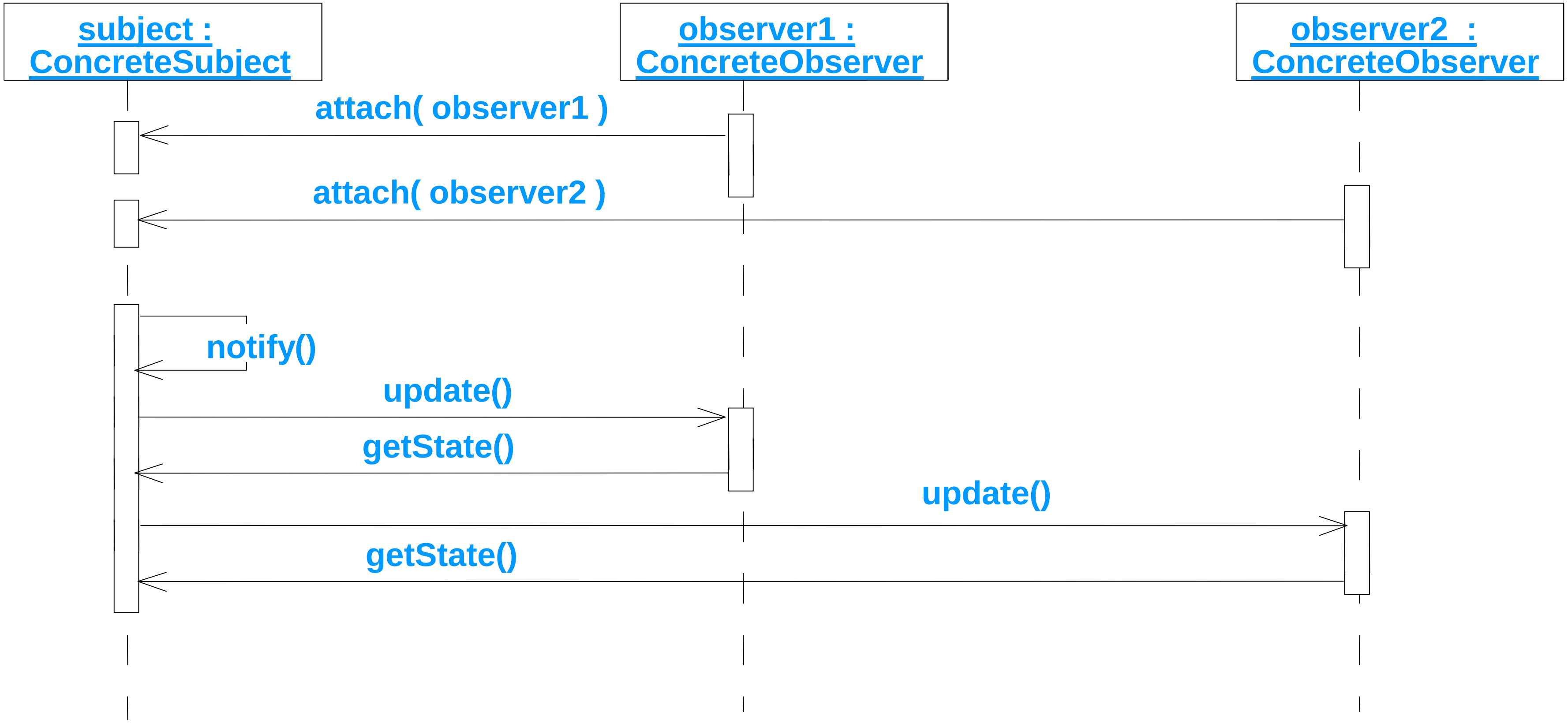
Collaborations

- ConcreteSubject notifies observers when changes occur.
- ConcreteObserver may query subject regarding state change.

Behavioral Patterns

Observer

Sequence Diagram



Lab

Lab d-05

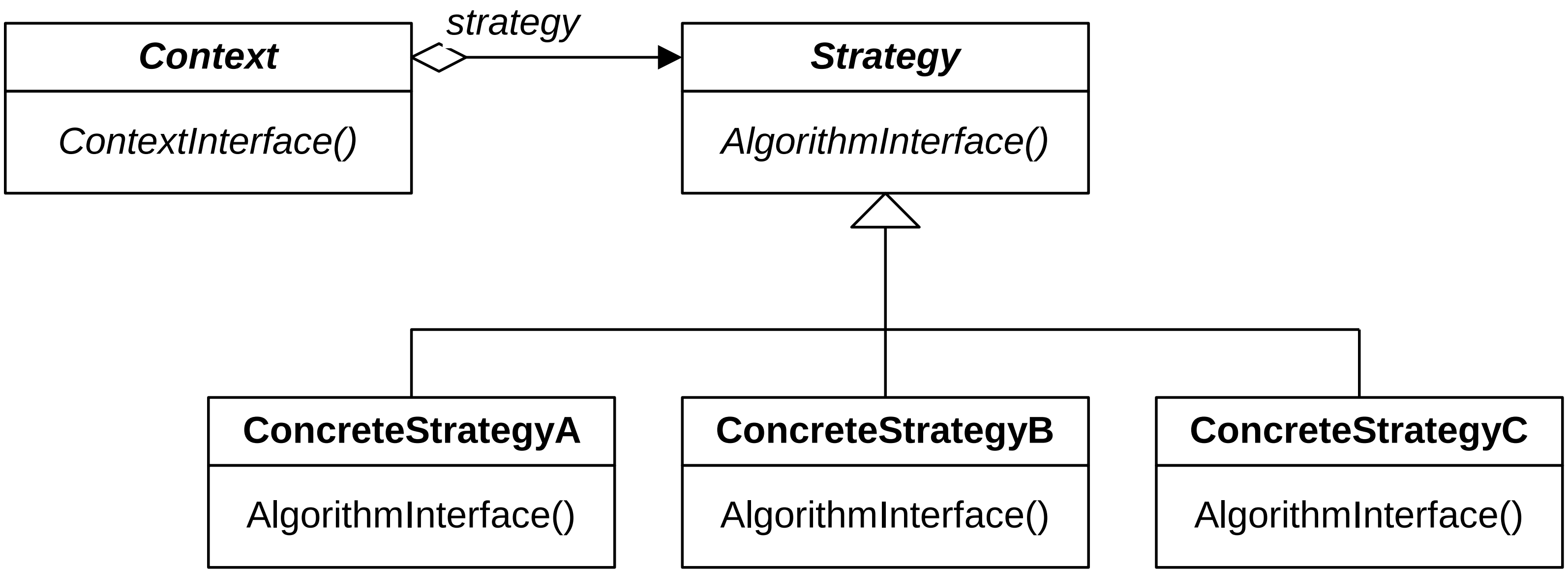
Behavioral Patterns
Strategy Pattern

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

- **Motivation:** when there are many algorithms for solving a problem, hard-wiring all algorithms in client's code may have several problems:
  - Clients get fat and harder to maintain
  - Different algorithms may be appropriate at different time
  - It is difficult to add new algorithms

Behavioral Patterns

Strategy Pattern



Behavioral Patterns

Participants of Strategy

- **Strategy:** declares an interface common to all supported algorithm. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- **ConcreteStrategy:** implements the algorithm using the Strategy interface
- **Context:** maintains a reference to a Strategy object and defines an interface that let Strategy access its data

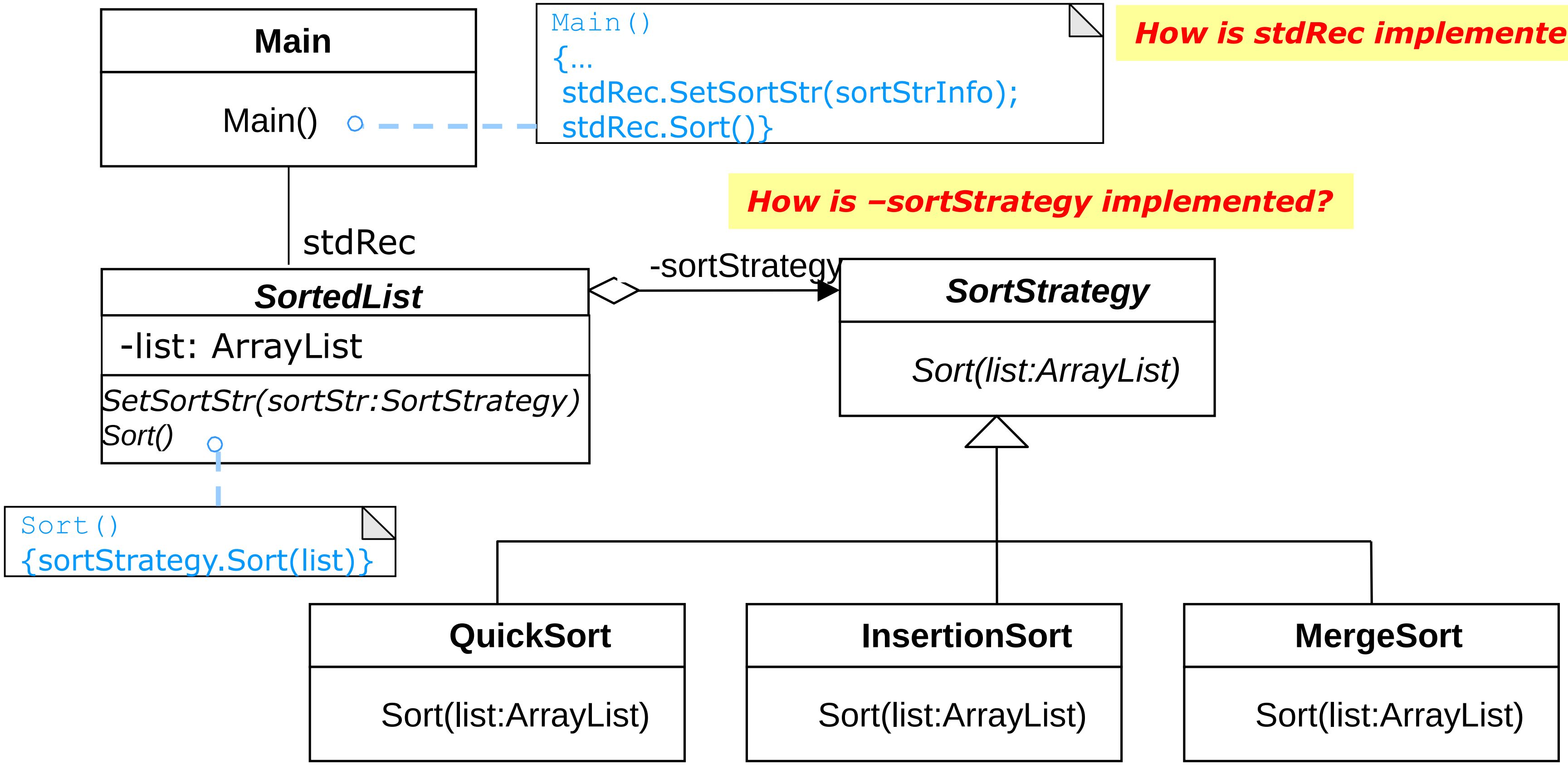
Behavioral Patterns

Sorting Example

- **Requirement:** we want to sort a list of integers using different sorting algorithms, e.g. quick sort, selection sort, insertion sort, etc.
- E.g., {3, 5, 6, 2, 44, 67, 1, 344, ... }
- {1, 2, 3, 5, 6, 44, 67, 344, ... }

- One way to solve this problem is to write a function for each sorting algorithm, e.g.
  - quicksort(int[] in, int[] res)
  - insertionsort(int[] in, int[] res)
  - mergesort(int[] in, int[] res)
- A better way is to use the Strategy pattern

Behavioral Patterns
Strategy Pattern



How is stdRec implemented?

How is –sortStrategy implemented?

Behavioral Patterns
Command

Intent

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

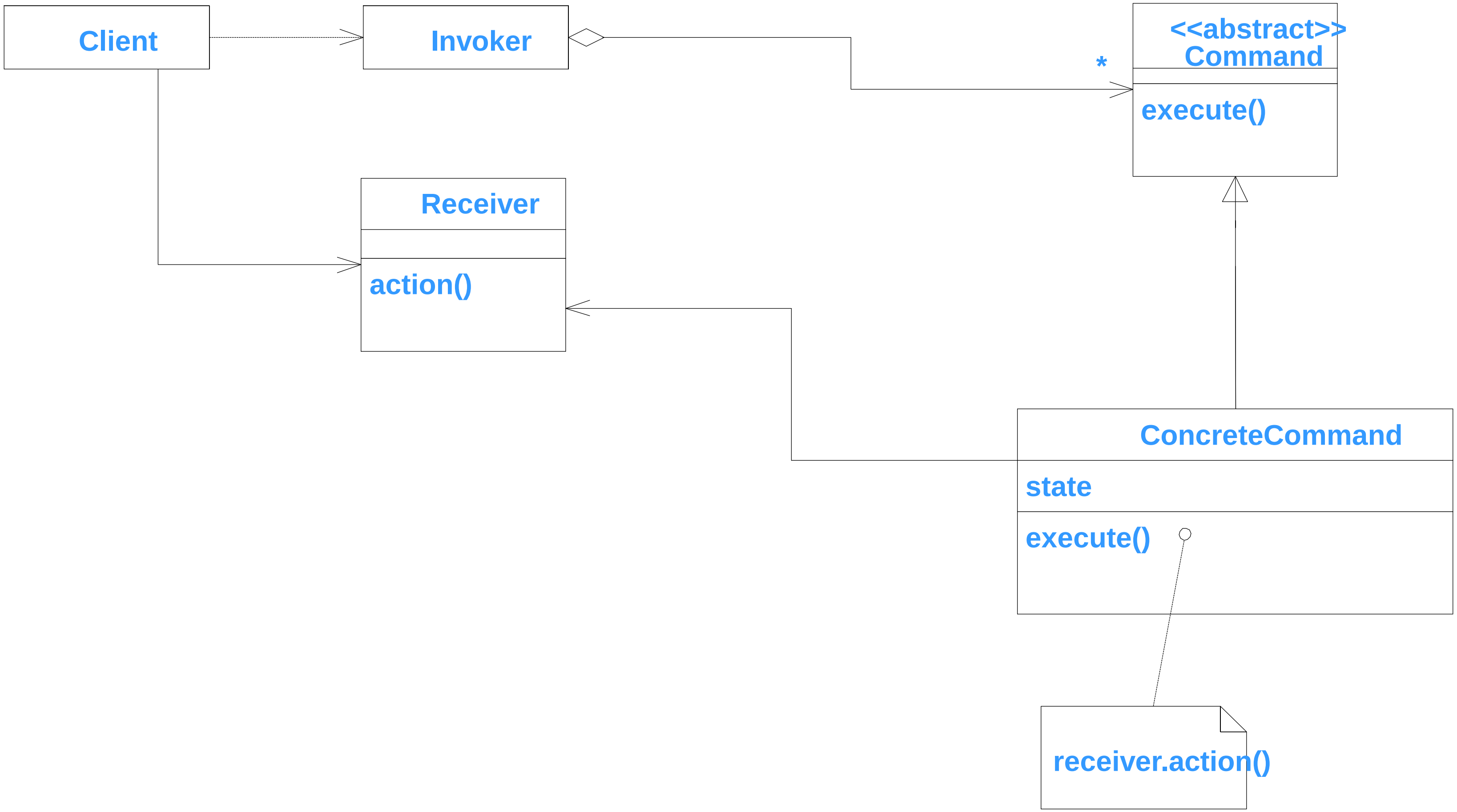
Applicability

- Parameterize objects by an action
- In place of “callbacks”
- Specify, queue, and execute requests at different times
- Supports undo when Command maintains state information necessary for reversing command.
- Added support for logging Command behavior.
- Support high-level operations built on primitive operations (transactions).

Behavioral Patterns

Command

Class Diagram



Behavioral Patterns
Command

Participants

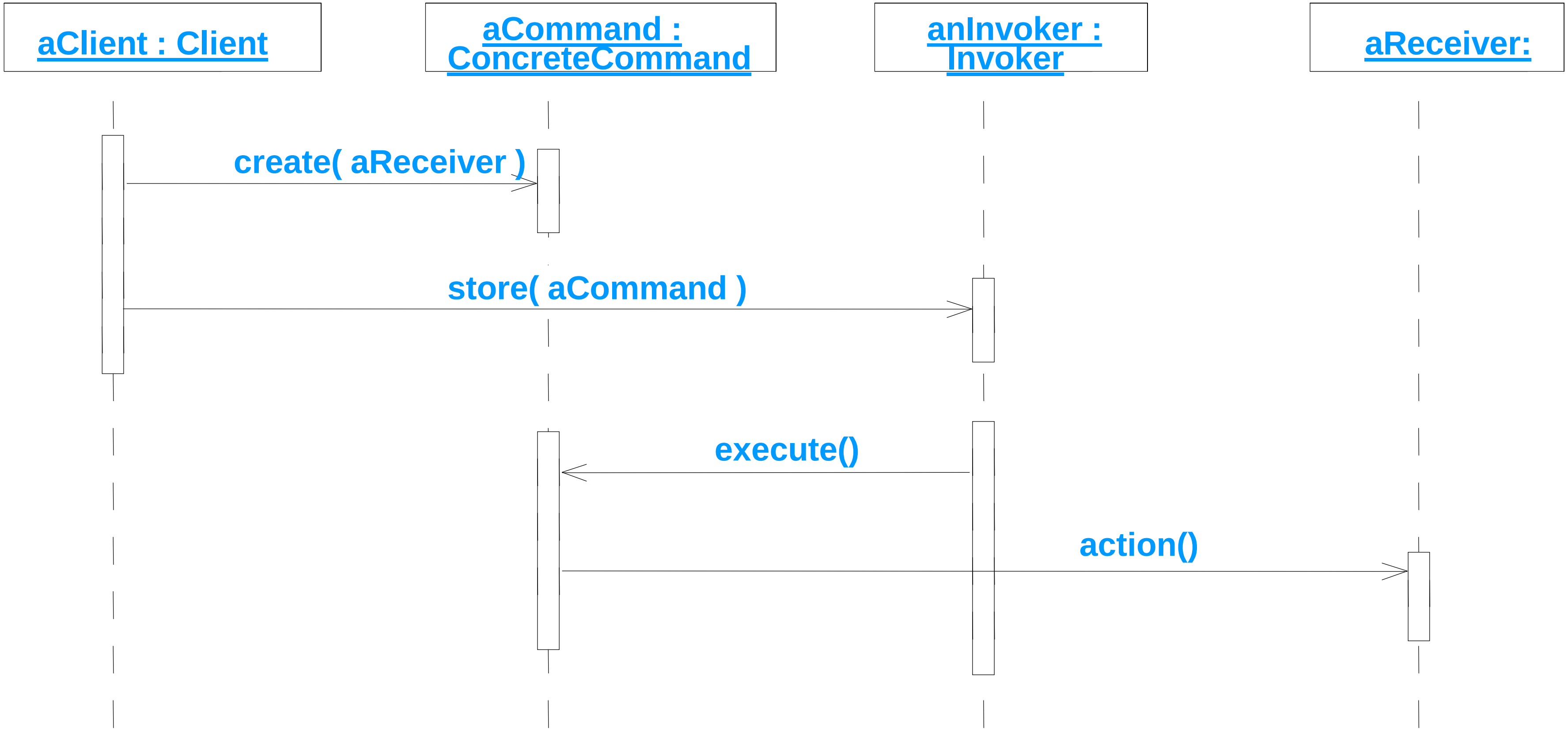
- Command: Declares an interface for executing an operation.
- ConcreteCommand
  - Defines a binding between a Receiver object and an action.
  - Implements execute() by invoking a corresponding operation on Receiver.
- Client (Application): Creates a Command object and sets its Receiver.
- Invoker: Asks the Command to carry out a request.
- Receiver: Knows how to perform the operation associated with a request. Can be any class.

Collaborations

- Creates a ConcreteCommand object and sets its Receiver.
- An Invoker stores the ConcreteCommand.
- Invoker calls *execute()* on command.
- ConcreteCommand invokes operation on its receiver.

Behavioral Patterns
Command

Sequence Diagram



Lab

Lab d-05

Behavioral Patterns
State

Intent

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

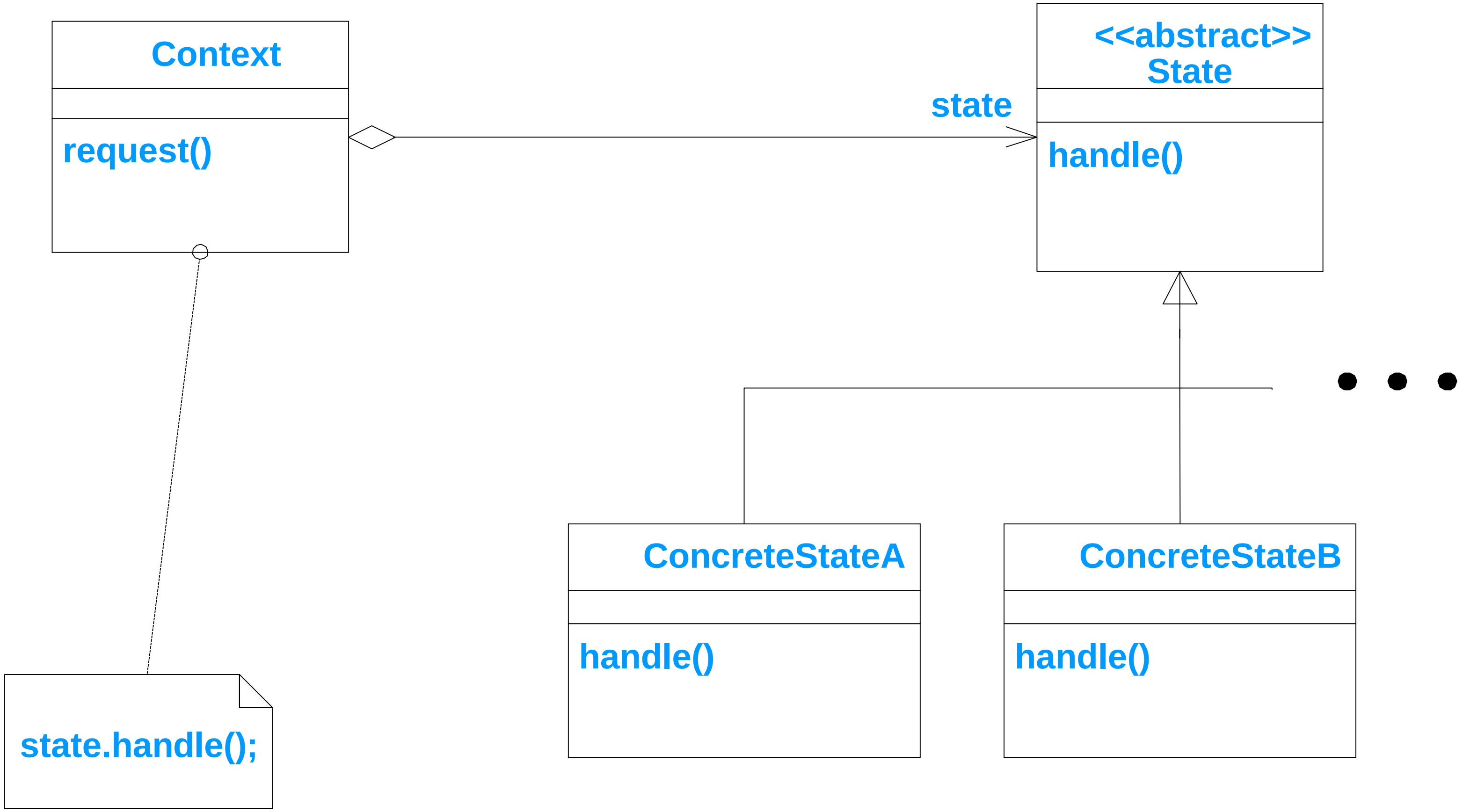
Applicability

- An object’s behavior depends on its state, and it must change its behavior at run-time depending on its state.
- Operations have large, multipart conditional statements that depend on the object’s state.
    - Usually represented by constants.
    - Some times, the same conditional structure is repeated.

Behavioral Patterns

State

Class Diagram



Behavioral Patterns
State

Participants

- Context
  - Defines interface of interest to clients.
  - Maintains an association with a subclass of State, that defines the current state.
- State
  - Defines an interface for encapsulating the behavior with respect to state.
- ConcreteStatex
  - Each subclass implements a behavior associated with a particular state of the Context.

Collaborations

- Context delegates state-specific behavior to the current concrete State object.
- The state object may need access to Context information; so the context is usually passed as a parameter.
- Clients do not deal with State object directly.
- Either Context or a concrete State subclass can decide which state succeeds another.

Behavioral Patterns

Visitor

Intent

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

Applicability

- An object structure contains many disparate classes, and operations need to be performed based on concrete classes.
- Many distinct operations need to be performed on an object structure.
- An object structure rarely changes, but new operations need to be defined over the structure.

Behavioral Patterns
Visitor

Participants

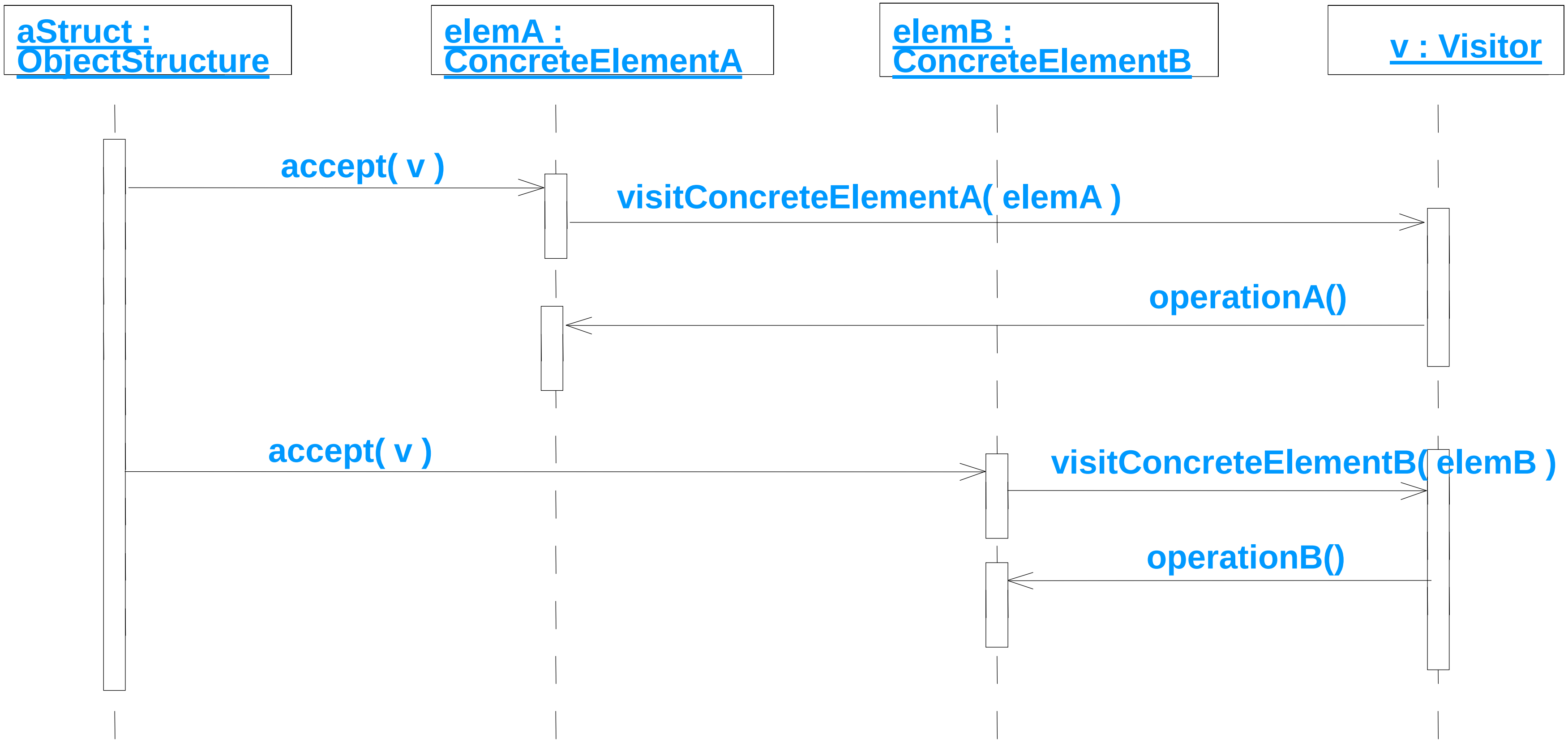
- Visitor — declares a visit operation for each class within the object structure aggregation.
- ConcreteVisitor — implements each operation declared by Visitor. Provides algorithm context.
- Element — defines an accept operation taking a Visitor as an argument.
- ConcreteElementX — implements an accept operation taking a Visitor as an argument.
- ObjectStructure
  - Enumerates its elements; potentially disparate classes.
  - May provide a high level interface for visitor to visit its elements.
  - Potentially a composite or just a general collection.

Collaborations

- A client creates an instance of a concrete Visitor subclass.
- Client requests the ObjectStructure to allow the visitor to visit each.
- When visited, Element invokes the appropriate operation on Visitor; overloading to know the element type.

Behavioral Patterns
Visitor

Sequence Diagram


aStruct : ObjectStructure
elemA : ConcreteElementA
elemB : ConcreteElementB
v : Visitor
accept(v)
visitConcreteElementA(elemA)
operationA()
accept(v)
visitConcreteElementB(elemB)
operationB()