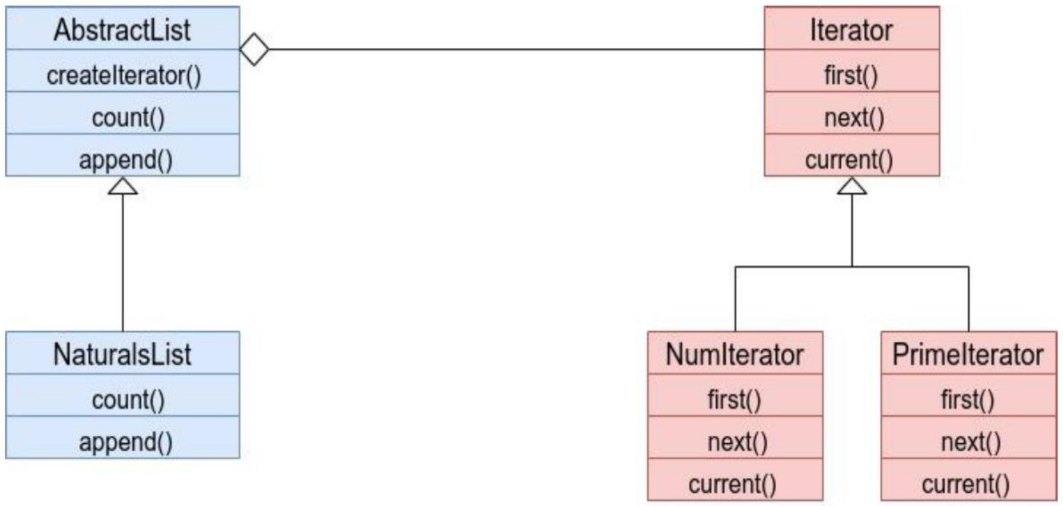
Behavioral Design patterns

Behavioral Design Patterns

- Iterator
- Strategy
- State
- Chain of Responsibility
- Command
- Observer
- Visitor
- Mediator
- Memento
- Template
- Interpreter

Iterator

Iterator



Intent:

- Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Motivation:

- Accessing the elements of an aggregate object(list) without knowing its internal structure.
- Traversal - take the responsibility for access or traversal out of the list object and put into an Iterator Object.

Applicability:

- To access an aggregate object's contents.
- To provide a uniform interface for traversing different aggregate Structures

Iterator – Similar patterns

- The abstract syntax tree of Interpreter is a Composite (therefore Iterator and Visitor are also applicable).
- Iterator can traverse a Composite. Visitor can apply an operation over a Composite.
- Polymorphic Iterators rely on Factory Methods to instantiate the appropriate Iterator subclass.
- Memento is often used in conjunction with Iterator. An Iterator can use a Memento to capture the state of an iteration. The Iterator stores the Memento internally.

Strategy

Strategy

Problem

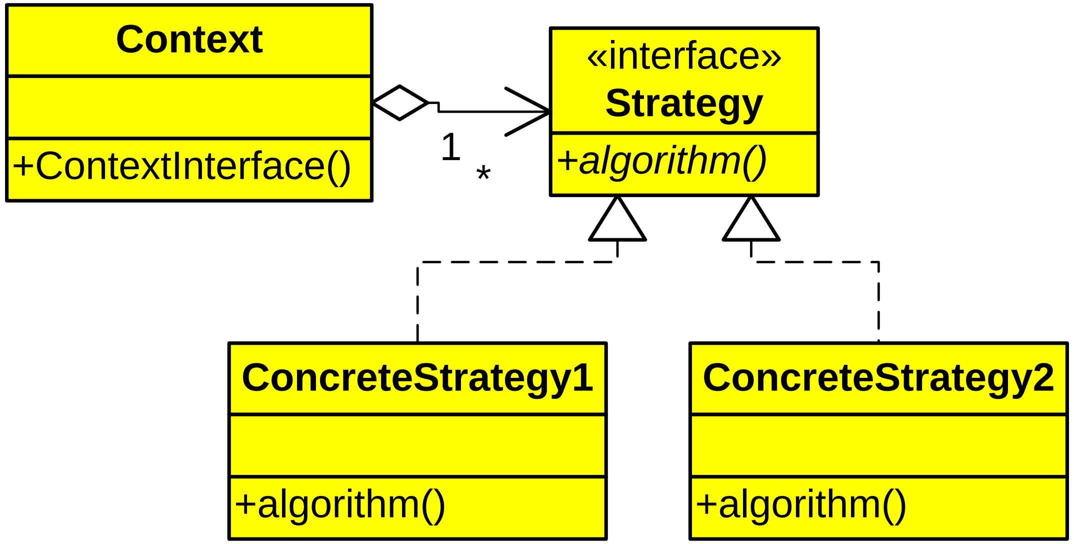
- Sometimes a system must support more than one algorithm for computing a certain result. However:
  - Clients likely do not want to be dependent on which algorithm is used
  - Which algorithm is the most effective may change at run-time
  - We want to integrate new algorithms over time, but without modifying existing code, if possible

- If clients have potentially generic algorithms embedded in them, it is difficult to:
  - reuse/exchange these algorithms, decouple different layers of functionality, and vary choice of policy at run-time
  - These embedded policy mechanisms routinely manifest themselves as multiple, monolithic, conditional expressions

Strategy

Solution Structure

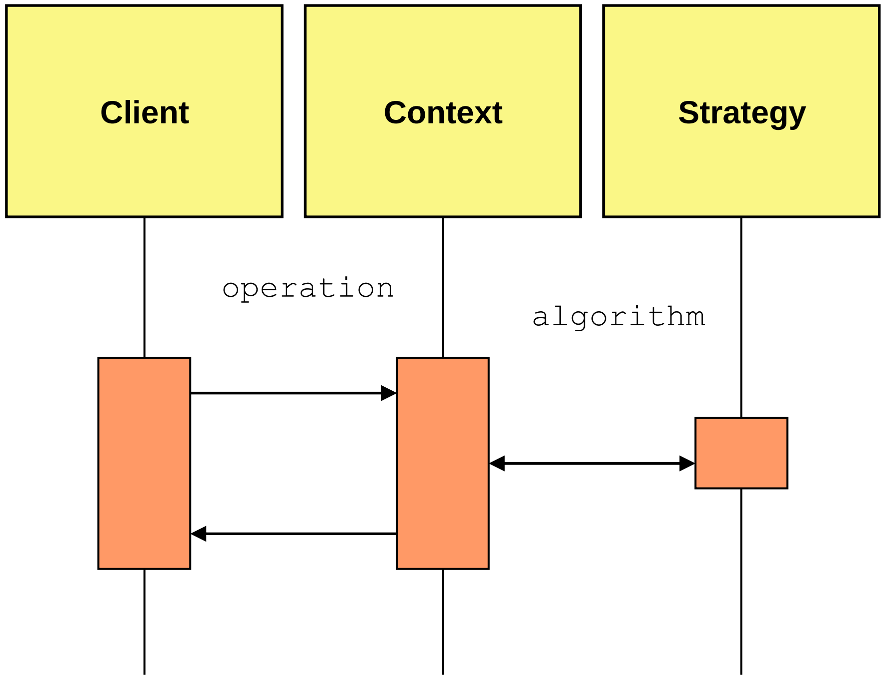
- *Encapsulate the variation* i.e. encapsulate each algorithm as a separate object
- A *Strategy* declares an interface common to all supported algorithms
- *Concrete Strategy* implements the strategy interface for a specific algorithm
- A *Context* offers a service to clients and
  - Is configured with a Concrete Strategy
  - Maintains a reference to a Strategy
  - May define an interface that lets Concrete Strategy access its data



Strategy

Solution Dynamics

- Delegate the execution of the varying algorithm to a concrete strategy
- A client invokes an operation on the context
- The context delegates the execution of a specific algorithm to a concrete strategy



Strategy

```
public class MyCollection {
   public void doSomething(Sorter sorter){
        sorter.sort(list);
   }
   public static void main(String args[]){
        MyCollection collection = new MyCollection();
        collection.doSomething(new QuickSorter());
        collection.doSomething(new MergeSorter());
   }
}
```

Context

Strategy

```
public interface Sorter {
   public List sort(List list);
}
```

```
public class QuickSorter implements Sorter{
   public List sort(List list){
               // sort in ascending order
               return list;
   }
}
```

Concrete Strategy

Strategy

Benefits

- Choice of different algorithms
- Extensibility with new algorithms
- Eliminates conditional statements

Liabilities

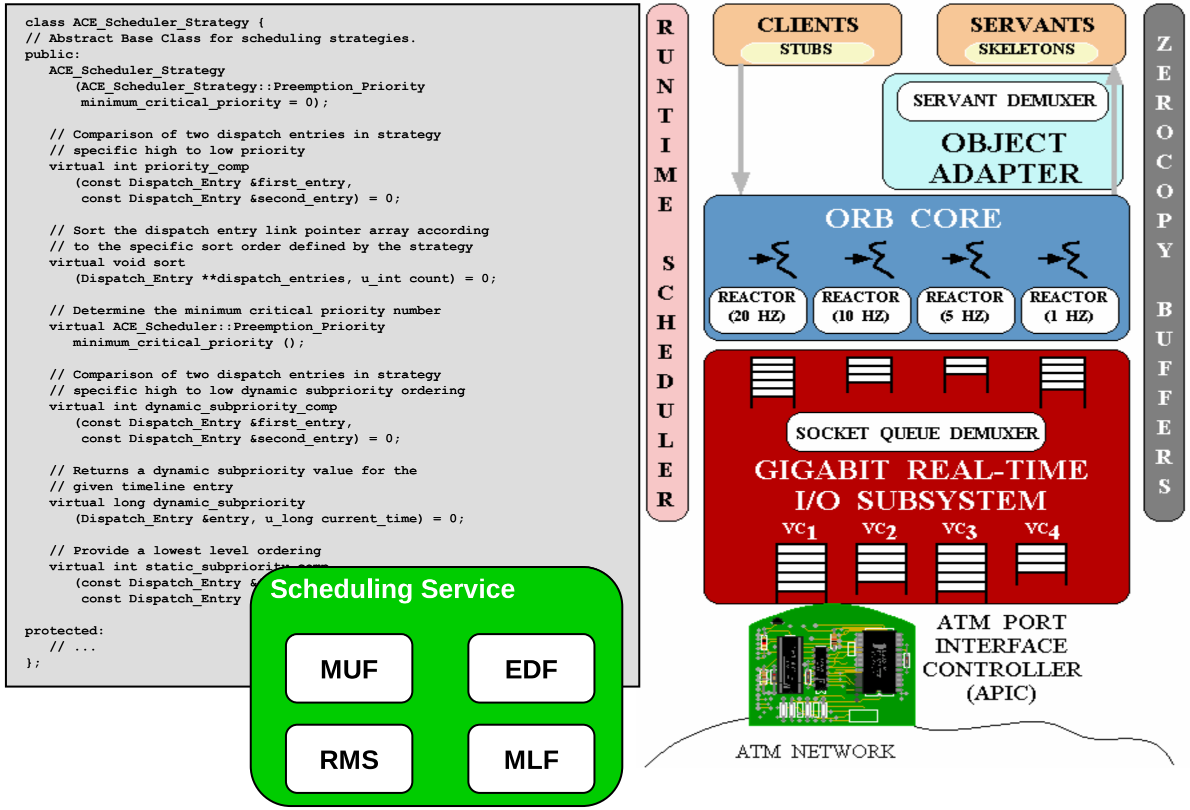
- Clients must be aware of strategies to configure the context
- Communication overhead between context and strategies
- Can result in increased number of objects
  - Solution – implement strategies as stateless objects that contexts can share. Strategy objects can be used as flyweights (Flyweight pattern)

Strategy

<u>Example - CORBA:</u>

- The CORBA ORB TAO provides a Scheduling Service to support real-time applications with deterministic Quality of Service requirements. The service is configurable with various scheduling policies:
  - rate monotonic scheduling (RMS),
  - earliest deadline first (EDF),
  - minimum latency first (MLF),
  - minimal laxity first (MLF),
  - maximum urgency first (MUF).

Another example –
Web Servers like JAWS for supporting various connection and caching policies



Strategy - Variants

- Context passes required data to Strategy
  - This results in less coupling between Context and Strategy
  - But Context might end up passing data that Strategy does not need
- Context passes itself to Strategy as an argument OR strategy can store a reference to its context so nothing needs to be passed
  - Results in increased coupling between Context and Strategy because now Strategy is dependent on Context.
- Strategy objects can be optional
  - If Context has a Strategy object, it uses it
  - Else, it executes default behavior
  - This allows a client to not deal with Strategy objects at all <u>unless</u> they don't like default behavior.

Strategy

<u>Applicability</u>

Use the Strategy pattern when

- Many related classes differ only in their behavior
- You need different variants of an algorithm
- A class defines many behaviors, and these appear as multiple conditional statements in its operations

Strategy – Design principles usage

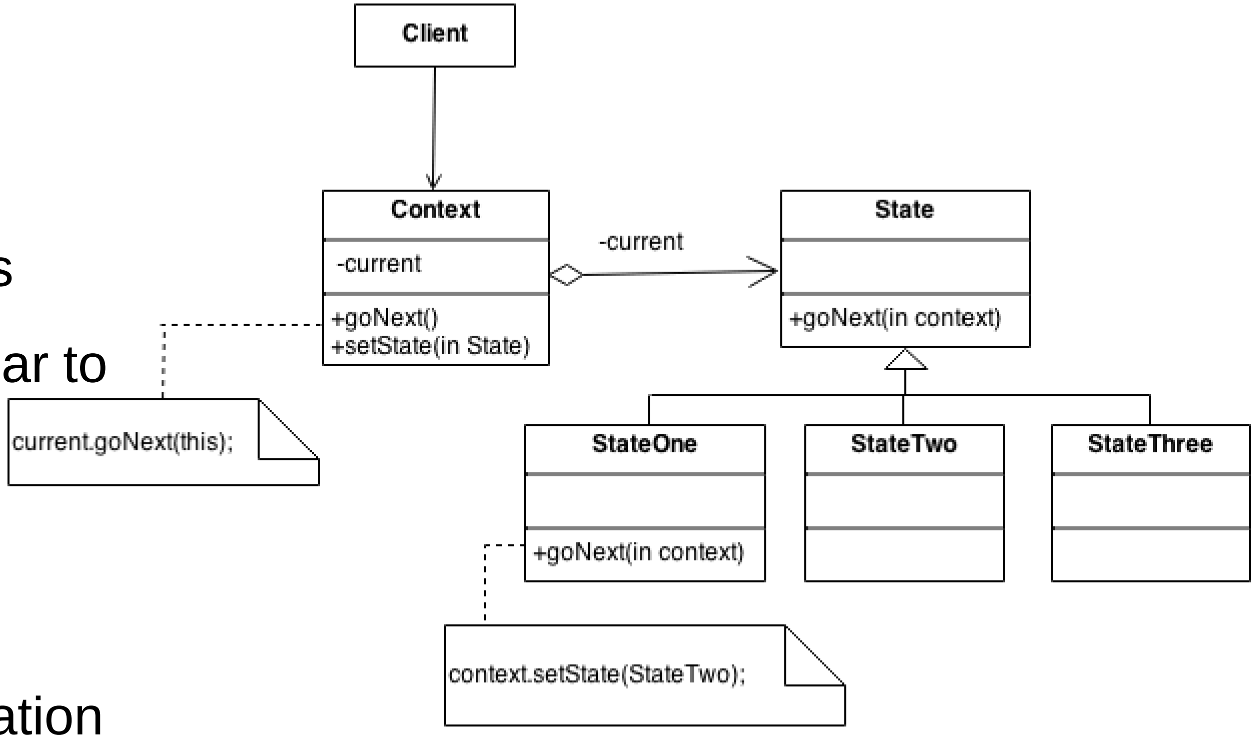
- Code to an interface
- Prefer delegation over inheritance
- LSP - inheritance used between the interface Strategy and the Concrete Strategies because they all behave similarly. This means reference to interface Strategy can be substituted with references to Concrete Strategies
- Dependency Inversion Principle – There is no dependency of Context on Concrete Strategies; instead both Context and Concrete Strategies depend on Strategy interface
- Encapsulate What Varies – Algorithms that were varying have been hidden behind the Strategy interface
- Open Closed Principle – To support new behavior, Context is not modified directly; instead, new behavior comes from new Concrete Strategies

State

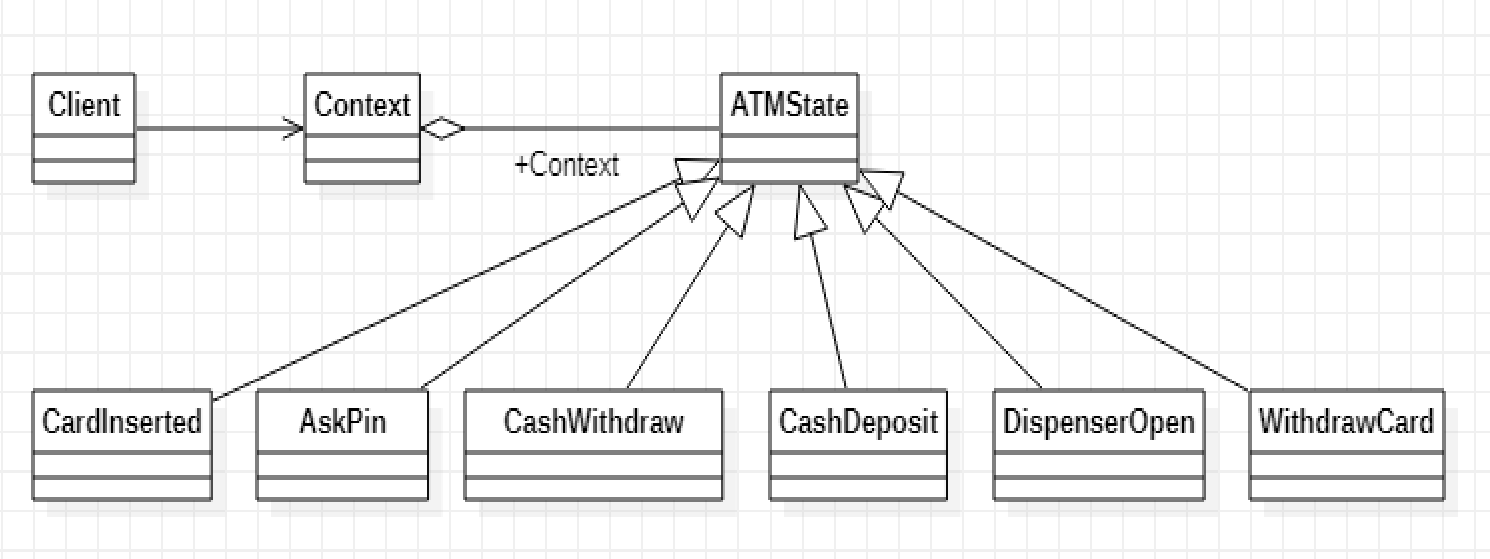
State pattern -

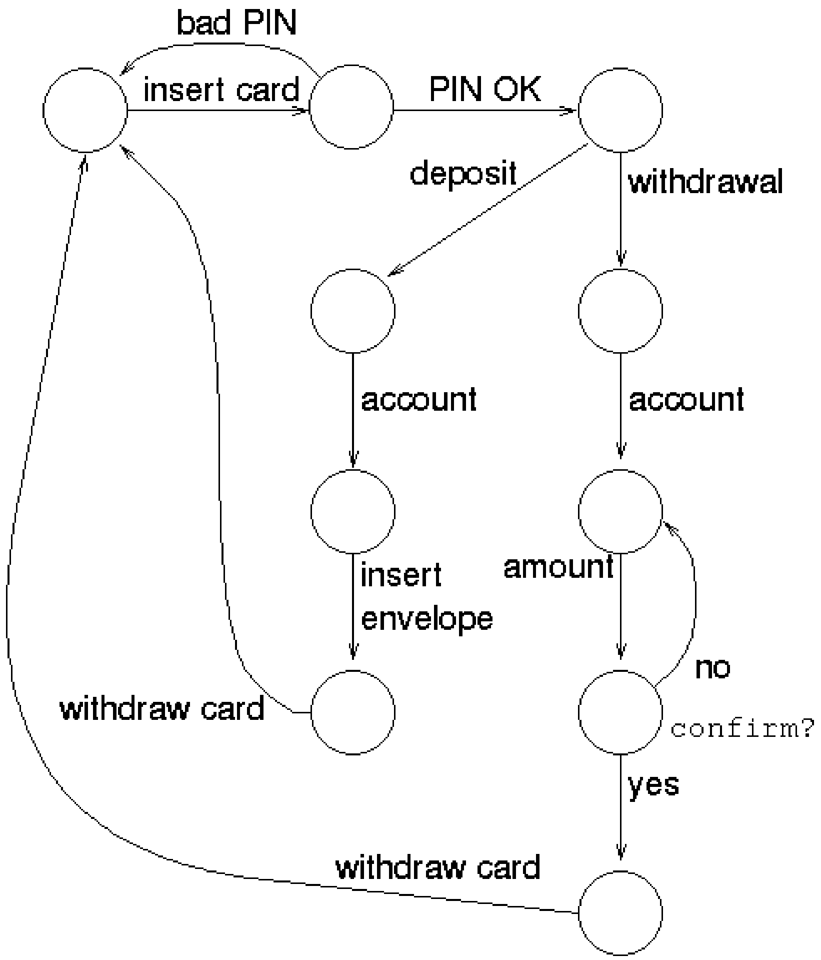
Intent

- Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
- An object-oriented state machine
- wrapper + polymorphic wrappee + collaboration



State pattern - example





State – Similar patterns

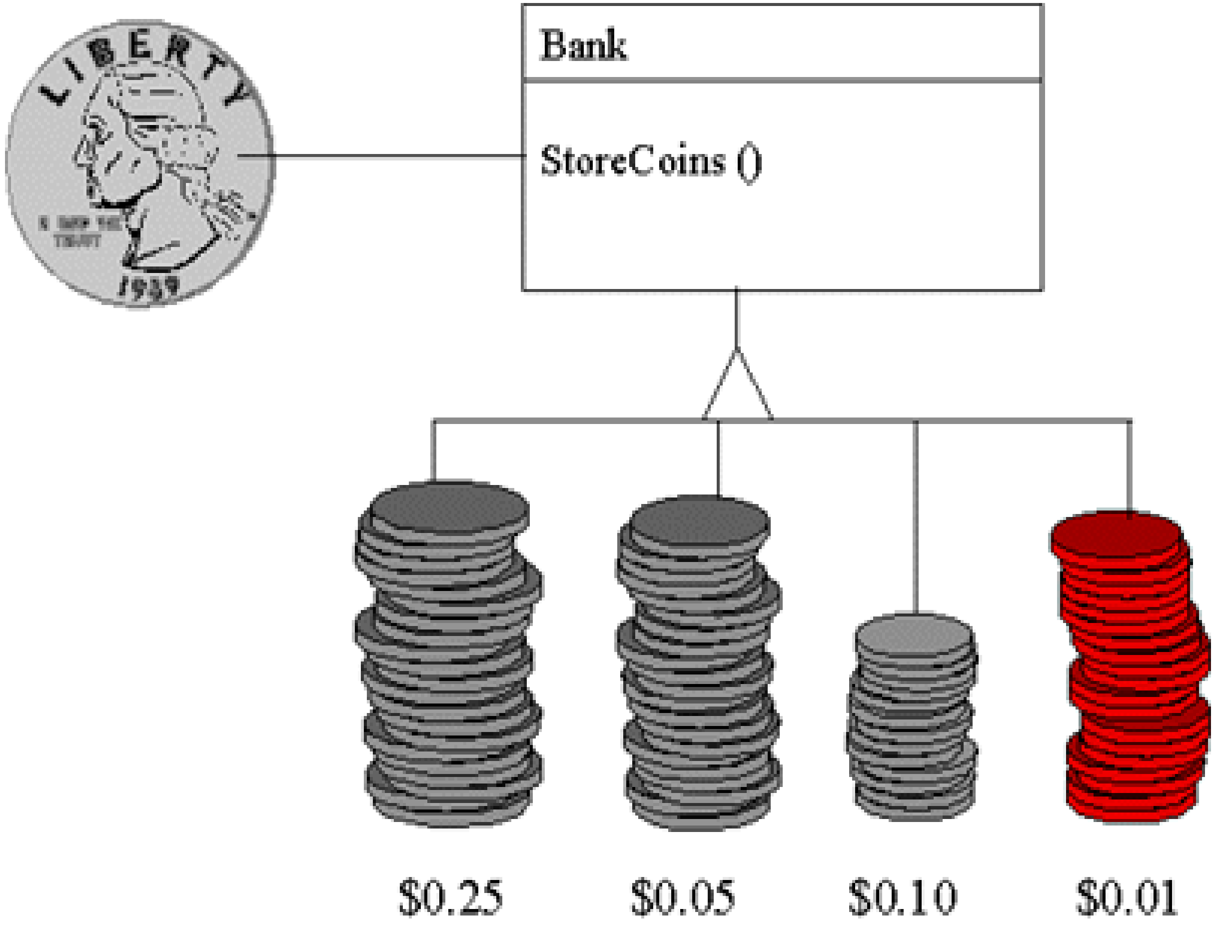
- State objects are often Singletons.
- Flyweight explains when and how State objects can be shared.
- Interpreter can use State to define parsing contexts.
- The structure of State and Bridge are identical (except that Bridge admits hierarchies of envelope classes, whereas State allows only one). The two patterns use the same structure to solve different problems: State allows an object's behavior to change along with its state, while Bridge's intent is to decouple an abstraction from its implementation so that the two can vary independently.
- The implementation of the State pattern builds on the Strategy pattern. With Strategy, the choice of algorithm is fairly stable. With State, a change in the state of the "context" object causes it to select from its "palette" of Strategy objects.

Chain of Responsibility

Chain of Responsibility - Motivation

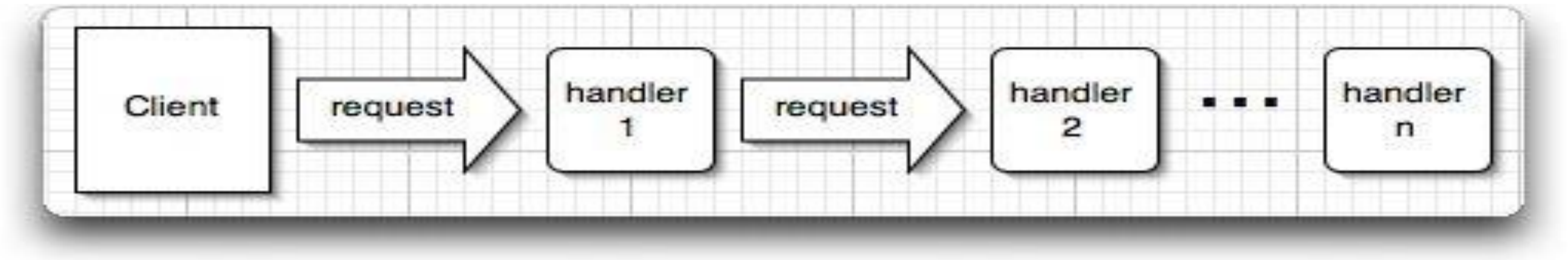
<u>Mechanical coin sorting in a bank or a vending machine</u>

- Uses a single slot for all coins
- As each coin is dropped, a CoR determines which tube accommodates each coin
- If a tube cannot accommodate the coin, the coin is passed on until a tube can accept the coin



Chain of Responsibility - Motivation

- Promote loose coupling between the sender of a request and its receiver by giving more than one object an opportunity to handle the request
- The receiving objects are chained and pass the request along the chain until one of the objects handles it
- Objects that form the chain can be decide dynamically at runtime by the client depending on the current state of the application



Chain of Responsibility pattern (Forces)

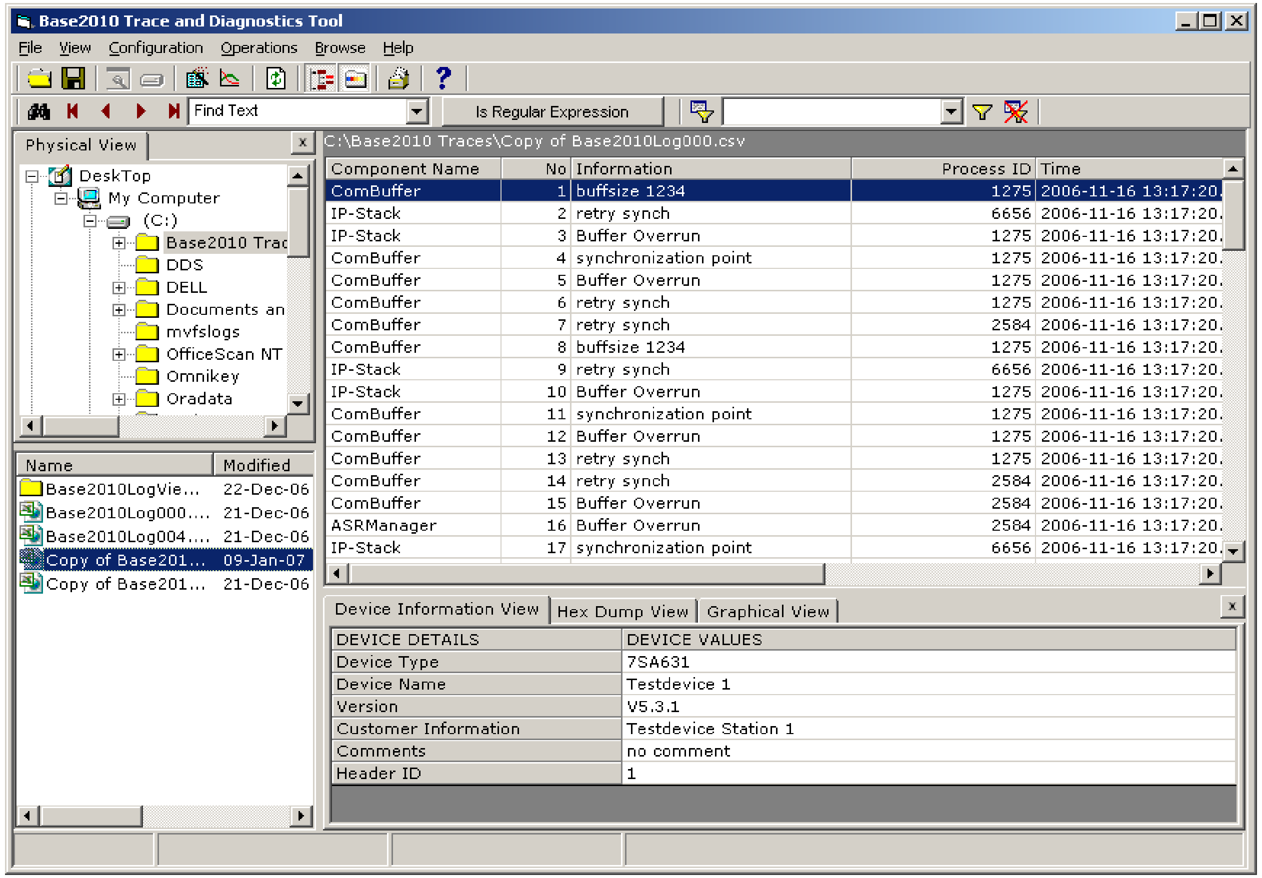
Forces

- You want an object to be able to send a command to another object without specifying the receiver
- More than one object may be able to receive and handle a command
- You need a way to prioritize among the receivers without the sending object knowing about them

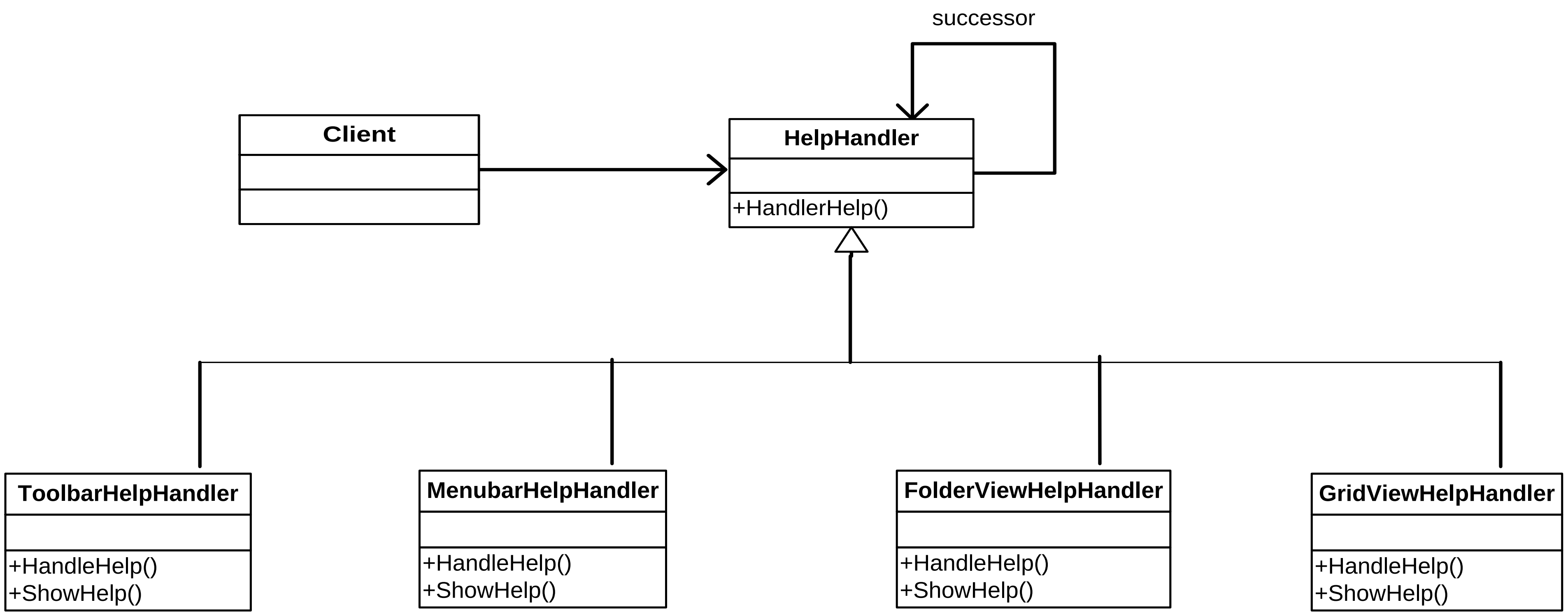
Chain of Responsibility - Problem

Consider a context-sensitive help facility for a graphical user interface

- A context specific help w.r.to display elements are required
- If specific information on a widget is not present then help system should display more general help



Chain of Responsibility - Solution



Chain of Responsibility - Benefits

Benefits

- Decouples the sender of the request and its receivers
- Simplifies your object because it doesn’t have to know the chain’s structure and keep direct reference to its members
- Allows you to add or remove responsibilities dynamically by changing the members or the order of the chain

Uses

- Commonly used in Windows systems to handle events like mouse clicks and keyboard events

Chain of Responsibility - Drawbacks

- Execution of the request isn’t guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage)
- Can be hard to observe the runtime characteristics and debug

Similar patterns

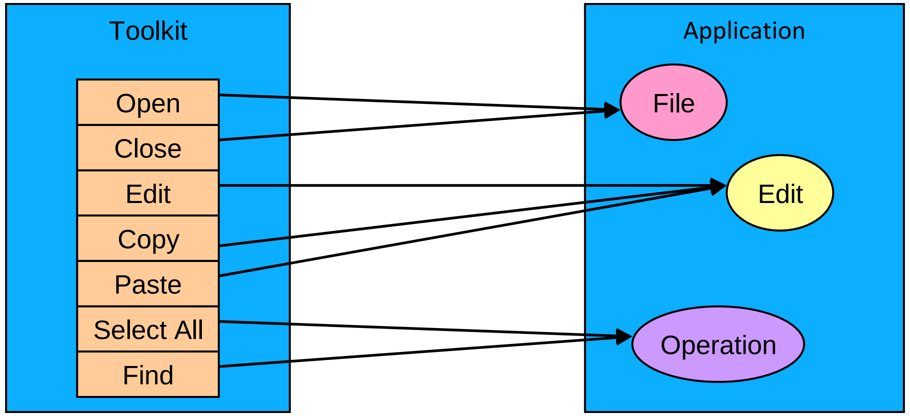
- Chain of Responsibility, Command, Mediator, and Observer, address how you can decouple senders and receivers, but with different trade-offs
- Chain of Responsibility can use Command to represent requests as objects
- Chain of Responsibility is often applied in conjunction with Composite. There, a component’s parent can act as its successor

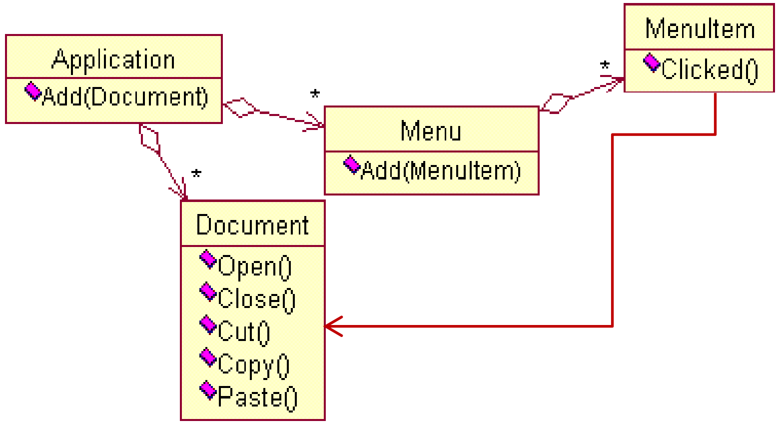
Command

Command (Behavioral)

Motivation

- Consider a User Interface toolkit that includes objects like buttons and menus that carry out a request in response to user input
  - The knowledge of what should be done in response to a request lies with the application using the toolkit
  - If this knowledge were to be embedded within the buttons or menus, it affects their reusability
- Further, UI look-and-feel typically changes at a different rate than interfaces of business components or services yet modifications to either entity should not affect the other
- Thus, loose coupling between UI toolkits (e.g. menus and menu items) and components providing services (e.g. document) is required

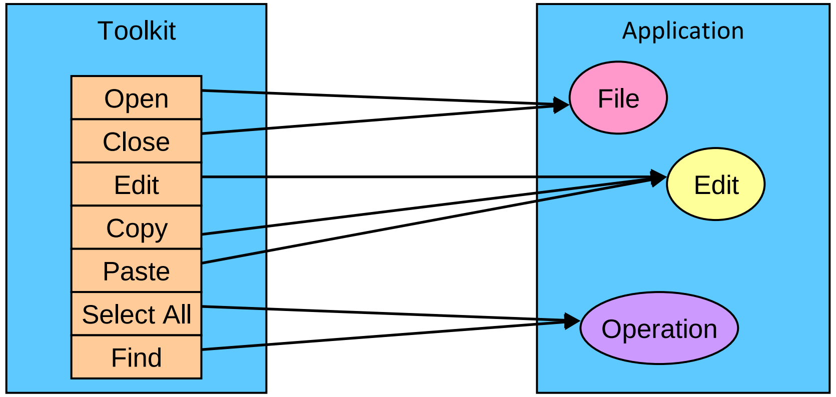


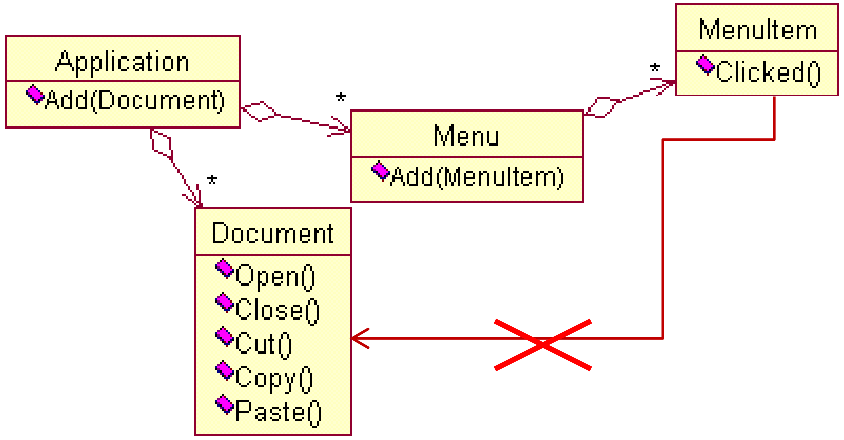


Command

<u>Problem</u>

- In user interface-based applications, the UI client accesses services of a component. However:
  - Object that invokes the operation should be decoupled from the object having the knowledge to perform it
  - It should be possible to specify, queue, and execute client requests at different times
  - Logging of requests should be supported
  - Reversing the effects of a request execution ("undo") should be supported

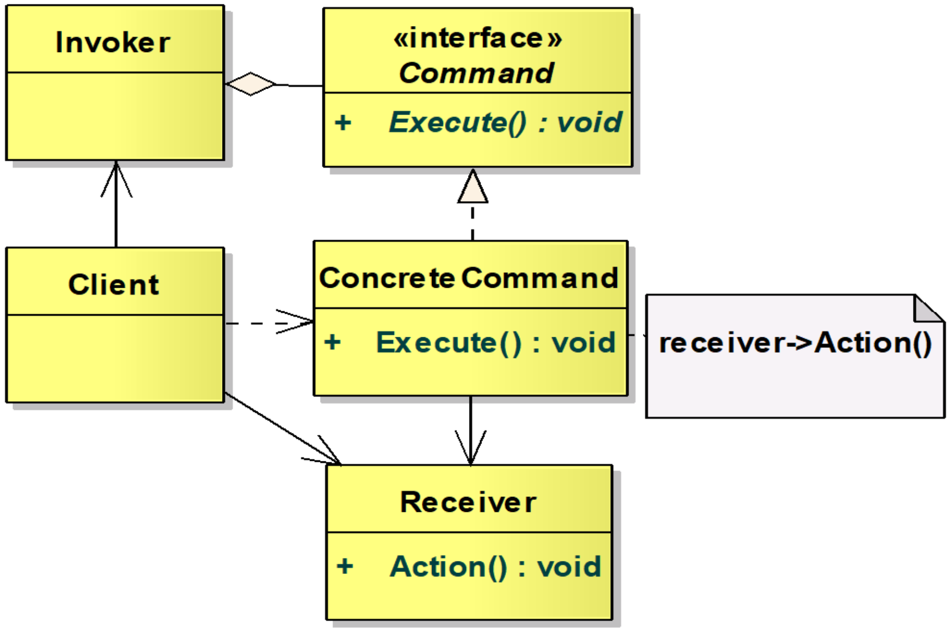


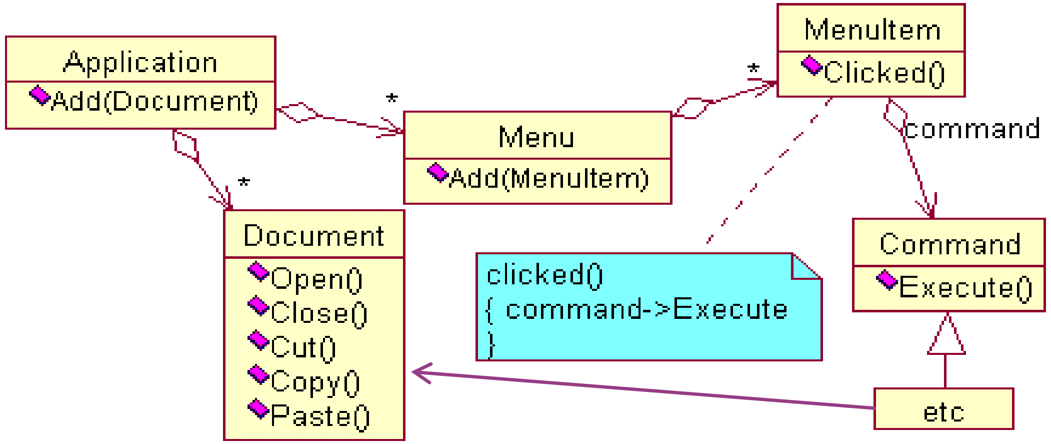


Command

<u>Solution Structure</u>

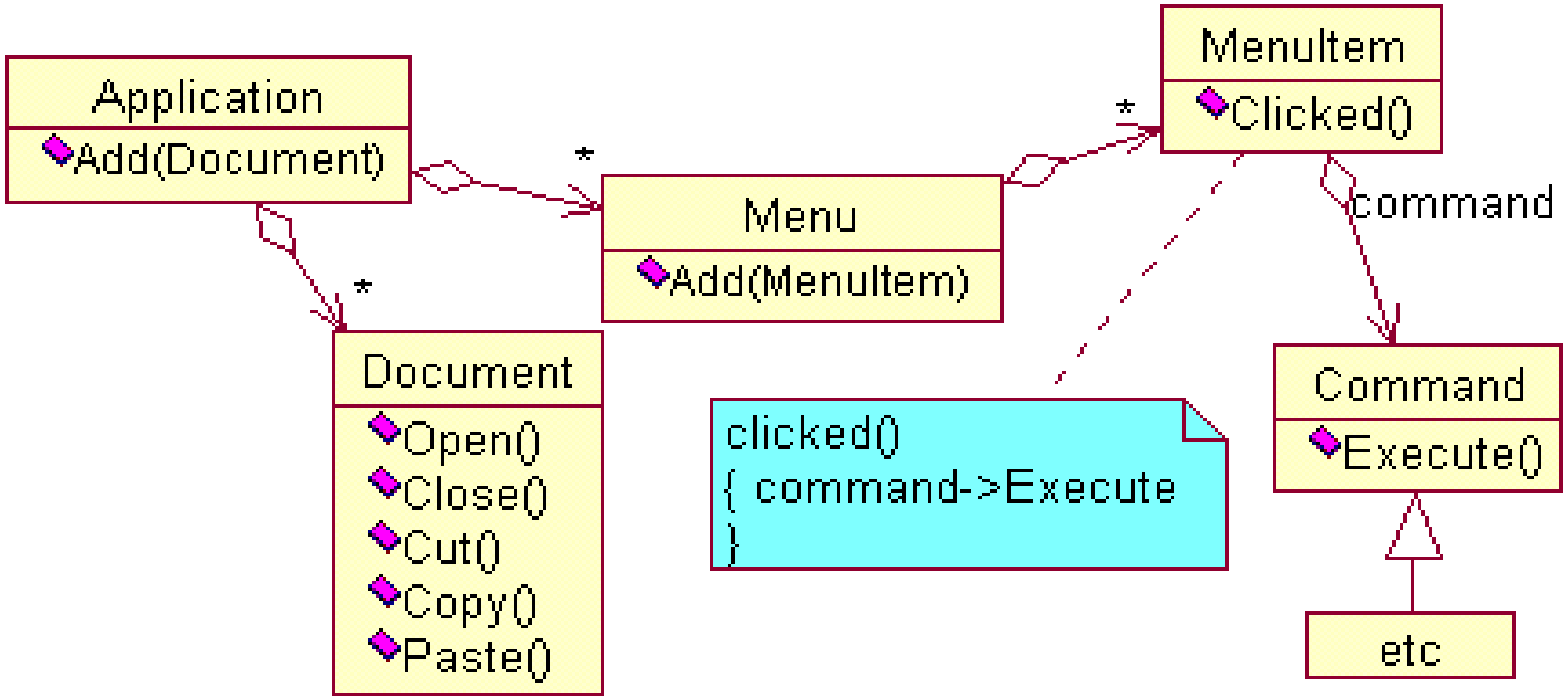
- Encapsulate requests into objects.
  - An interface *command* defines an explicit interface for executing arbitrary requests
  - *Concrete commands* implement the command interface to execute a specific request on the services or components of the application (*receiver*)
  - *Client* creates a concrete command, sets its *receiver,* and stores it in *invoker*
  - *Invoker* calls Execute on the *command*
  - A *receiver* provides the services to execute requests.





Command

Example: User Interface Toolkit

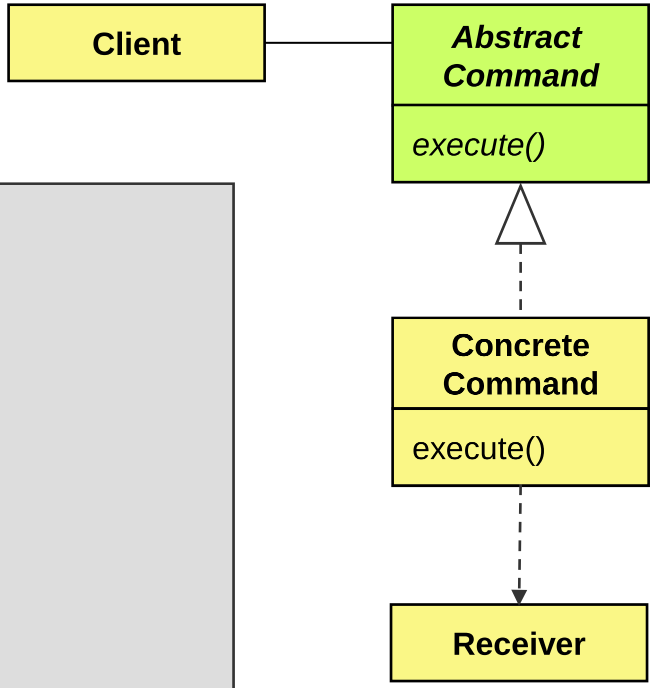


Command

The abstract command interface

```
interface Command { void Execute(); }

class Print : Command {
        public void Execute() {
                        Console.WriteLine("Print"); }
}
...
class Macro : Command {
        private Command[] seq;
        public Macro() {
                seq = new Command[]{new Open(),
                                new Print(), new Close()}; }
        public void Execute() {
                foreach (Command c in seq) c.Execute(); }
}
```




```
class CommandDemo
{
        static void Main(string[] args)
        {
                Command c = new Macro();
                c.Execute();
                Console.ReadLine();
        }
}
```

Observer

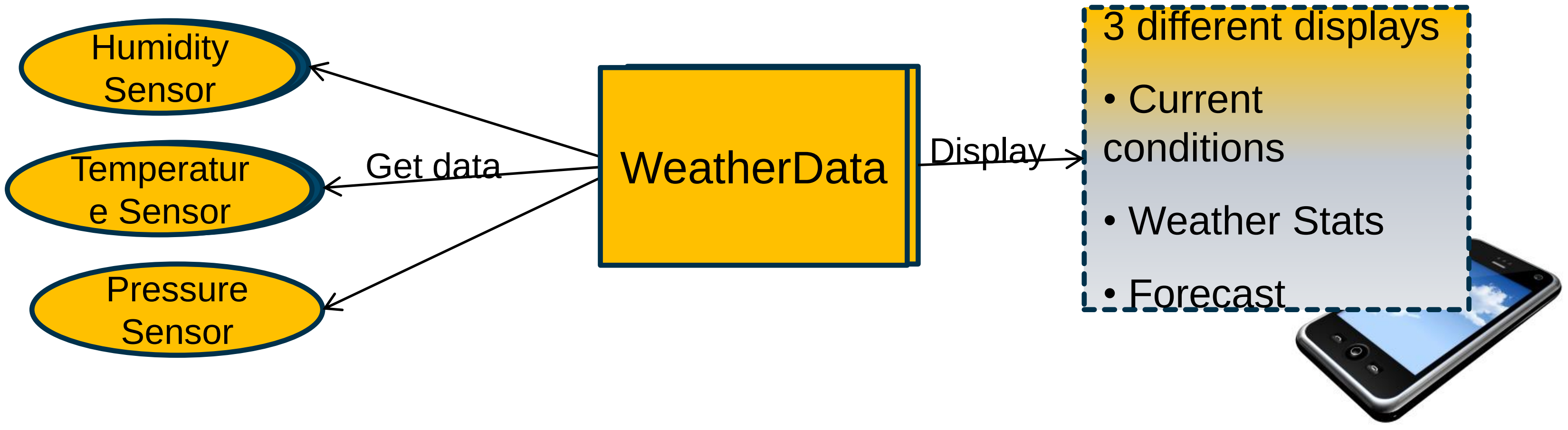
Observer (Motivation)

```
public class WeatherData
{
  // instance variable declarations
  public void measurementsChanged()
 {
  float temp = getTemperature();
  float humidity = getHumidity();
  float pressure = getPressure();

currentConditionsDisplay.update
(temp, humidity, pressure);

statisticsDisplay.notify(temp,
humidity, pressure);

forecastDisplay.inform(temp,
humidity, pressure);
 }
}
```



What is the problem with this code?

- What if there is a new display element tomorrow?
- Will the WeatherData need information about what method to call in the new display element?
- Can we add/remove display elements at *run-time*?

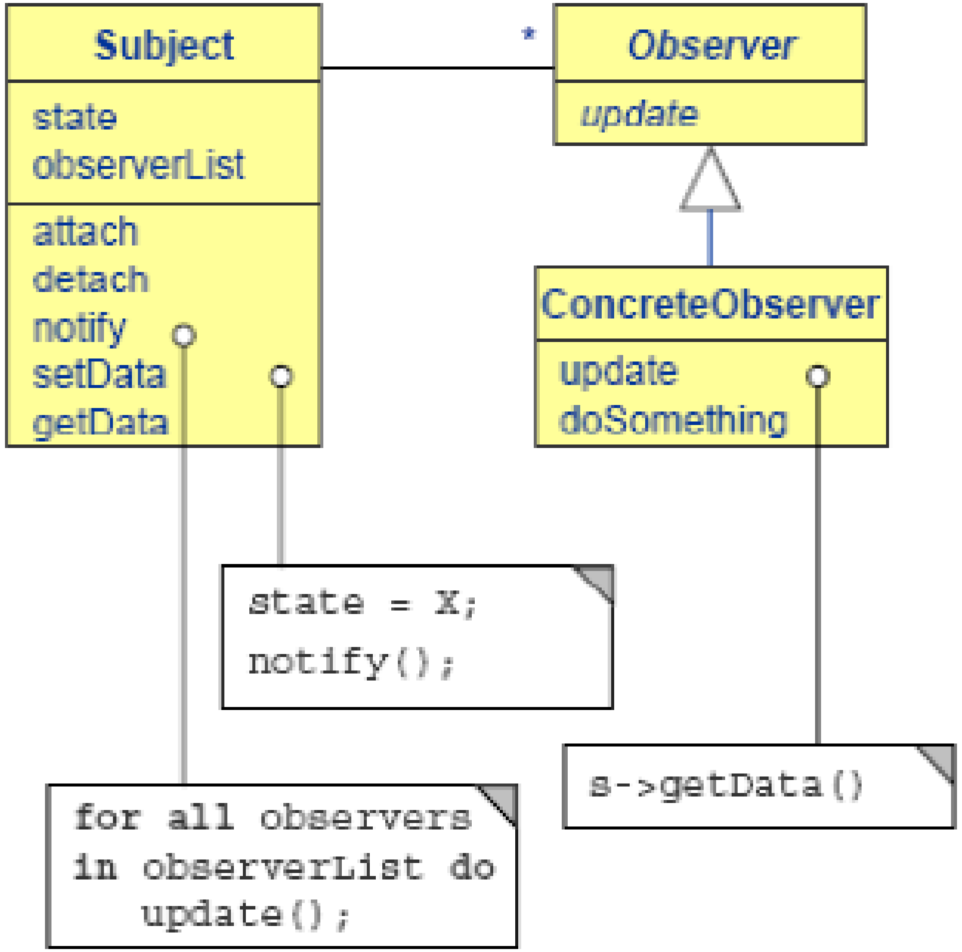
Observer - Problem

Problem

- When the internal state of an object changes, other objects that are dependent on it need to be informed. How can this be done such that
  - Information provider is only loosely coupled with information consumers
  - Information consumers that depend upon the information provider should not be known beforehand

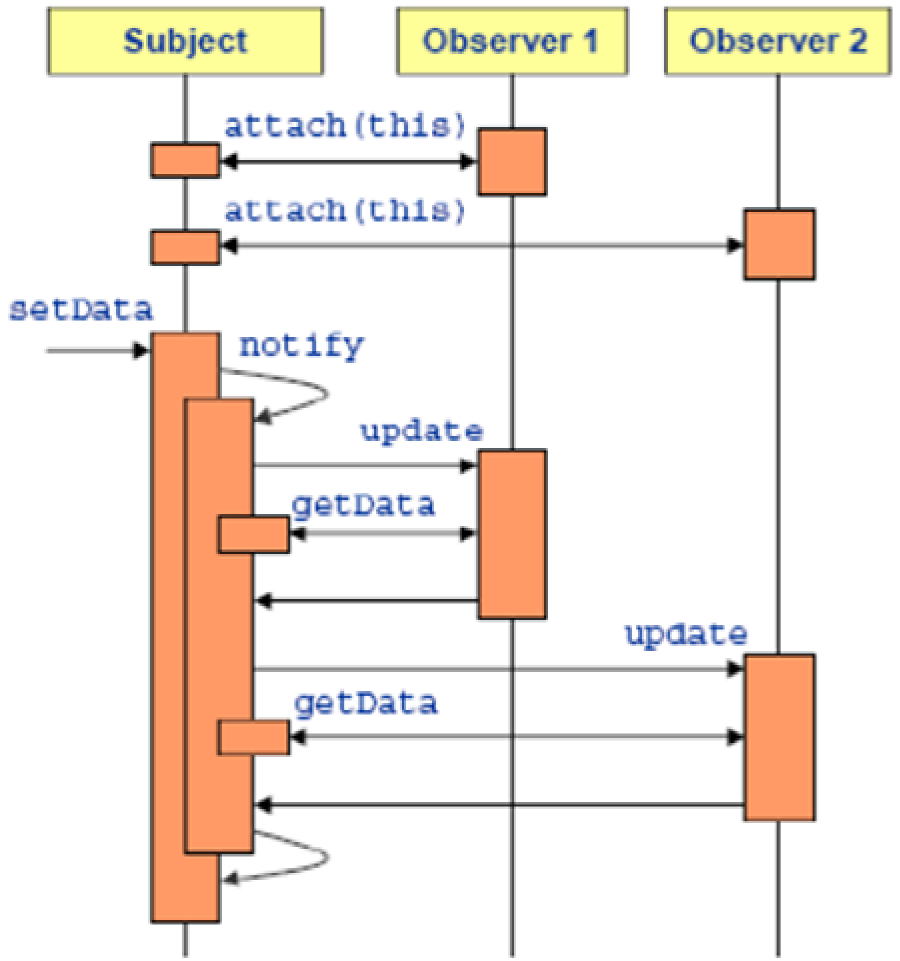
Observer - Structure

- Implement a change propagation mechanism **between** the information provider (the subject) and the information consumers (the observers).
- The subject maintains a registry of observers and notifies all registered observers about changes to its state.
- An observer declares an update function to be called by the subject's change propagation mechanism.
- Concrete observers implement the update function in a system-specific manner.



Observer – Solution dynamics

- The observers register with the subject's change propagation mechanism.
- A client modifies the subject's data.
- The subject starts its change propagation mechanism to call the update function of all registered observers.
- The observers retrieve the changed data from the subject and update themselves.



Observer – Consequences

Benefits

- Defined handling of dependencies between otherwise strongly coupled objects.
- Support for dynamic configuration of a subject with observers.
- Adding new observers does not affect the subject or the change propagation mechanism.

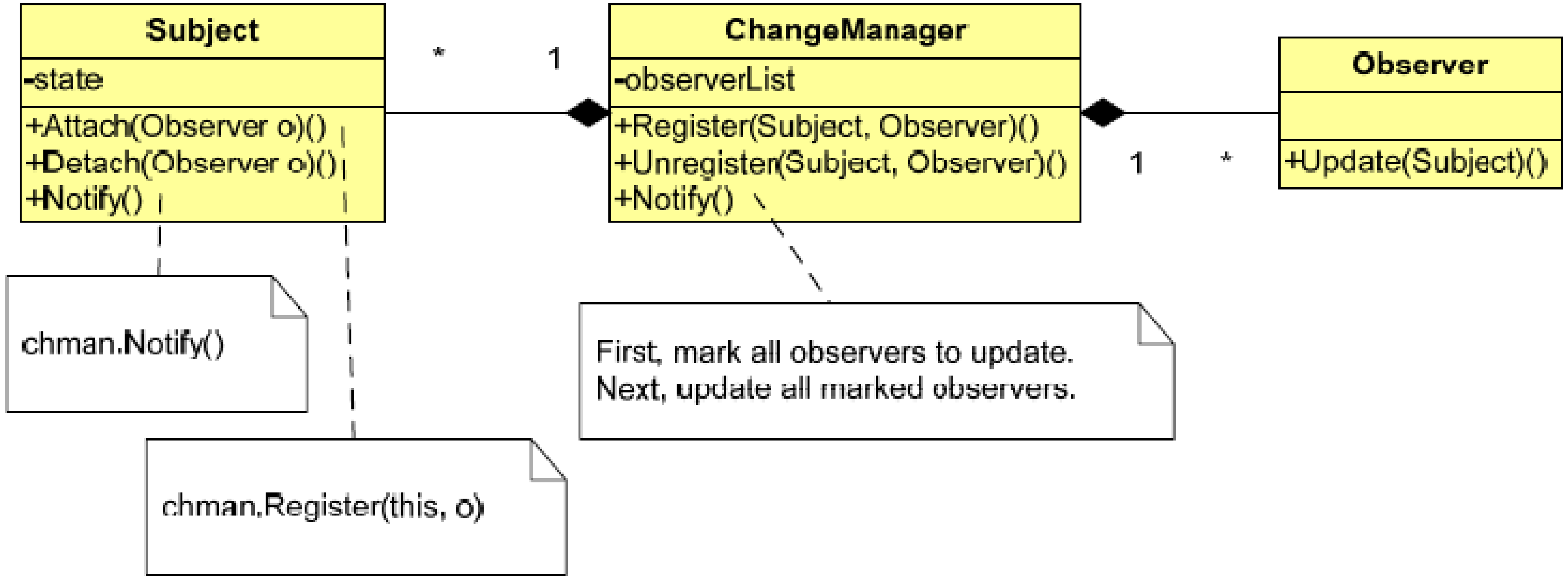
Liabilities

- Unnecessary updates may be received.
- Cascade of updates.
- Indirection –new data value is not directly received; only notification of change is received

Observer – A Variant

Change Manager (Mediator)

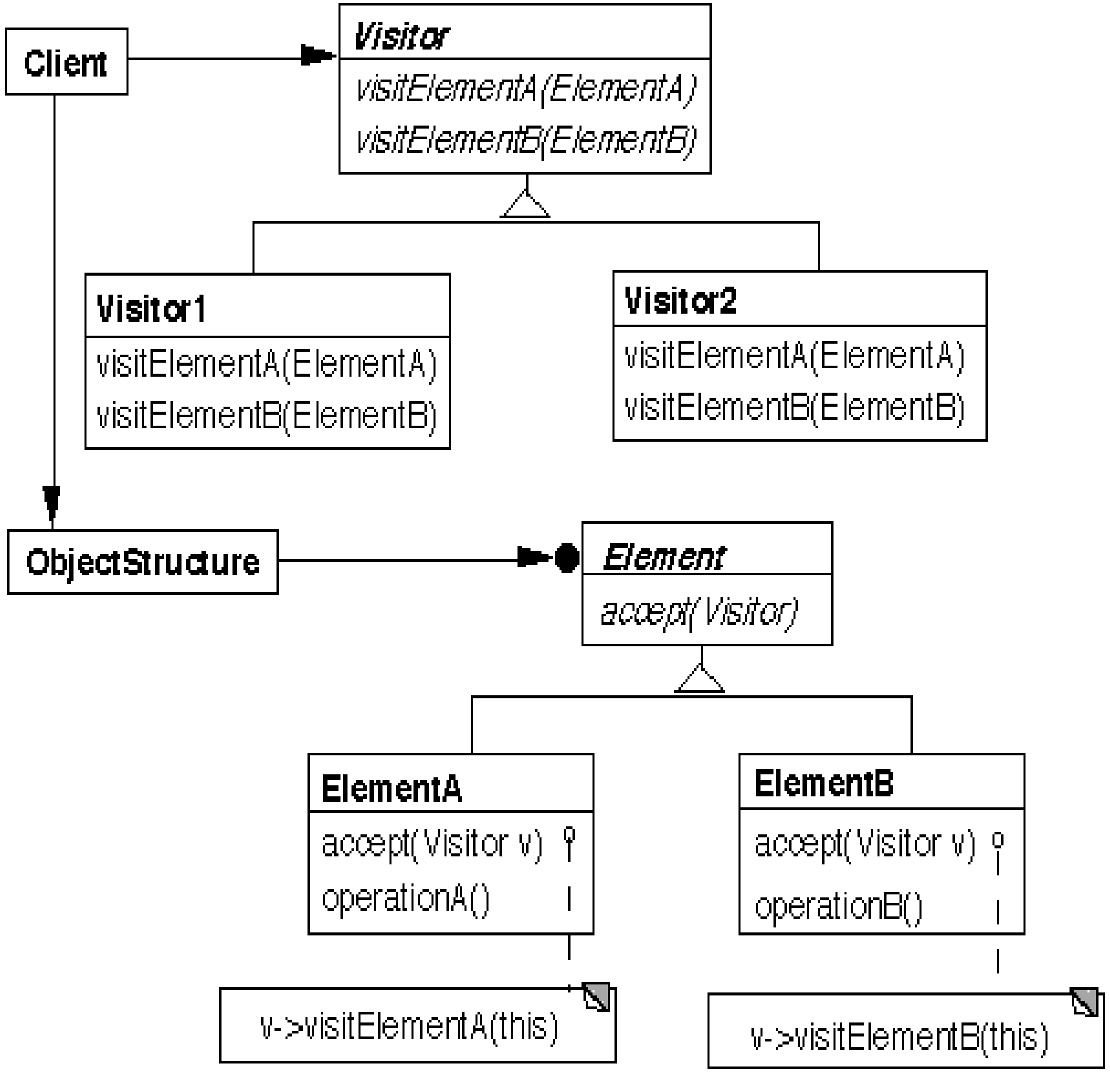
- Helps encapsulate complex dependency relationships between subjects and observers.
- Eliminates the need for subjects to maintain references to their observers.



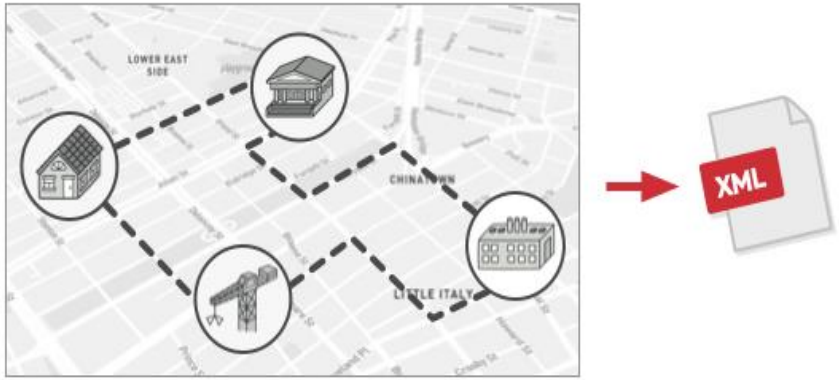
Visitor

Visitor - Intent

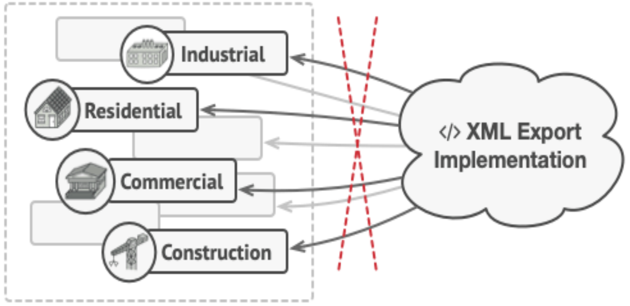
- Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.
- Do the right thing based on the type of two objects.
- Double dispatch



Visitor - Problem



Exporting the graph into XML.



The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
```

- Node
  - Node (Commercial)
  - Node (Construction)
  - Node (Industrial)
  - Node (Residential)

```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
}
```

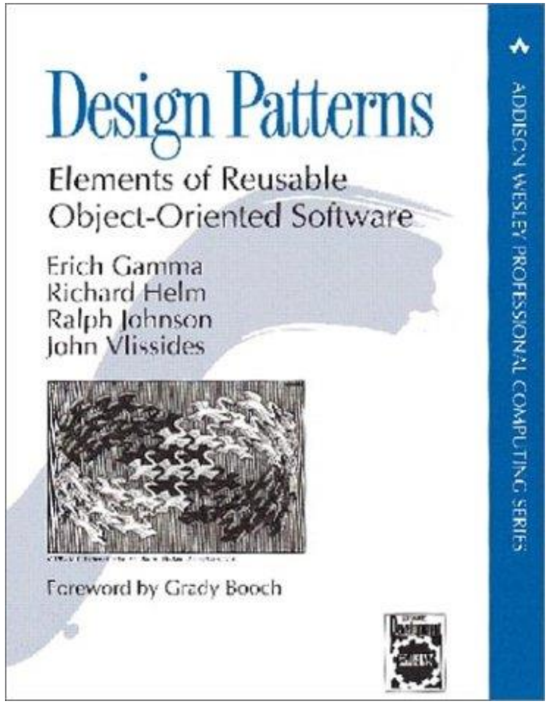
With XML export, the actual implementation will probably be a little bit different across various node classes.

However, the Visitor pattern addresses this problem. It uses a technique called **Double Dispatch**, which helps to execute the proper method on an object without cumbersome conditionals. Instead of letting the client select a proper version of the method to call, how about we delegate this choice to objects we're passing to the visitor as an argument? Since the objects know their own classes, they'll be able to pick a proper method on the visitor less awkwardly. They "accept" a visitor and tell it what visiting method should be executed.

Bibiliography

The Gang Of Four book is the first, and still the most popular pattern book. It contains 23 general purpose design patterns and idioms for:

- *Object creation*: Abstract Factory, Builder Factory Method, Prototype, and Singleton
- *Structural Decomposition*: Composite and Interpreter
- *Organization of Work*: Command, Mediator, and Chain of Responsibility
- *Service Access*: Proxy, Facade, and Iterator
- *Extensibility*: Decorator and Visitor
- *Variation*: Bridge, Strategy, State, and Template Method
- *Adaptation*: Adapter
- *Resource Management*: Memento and Flyweight
- *Communication*: Observer

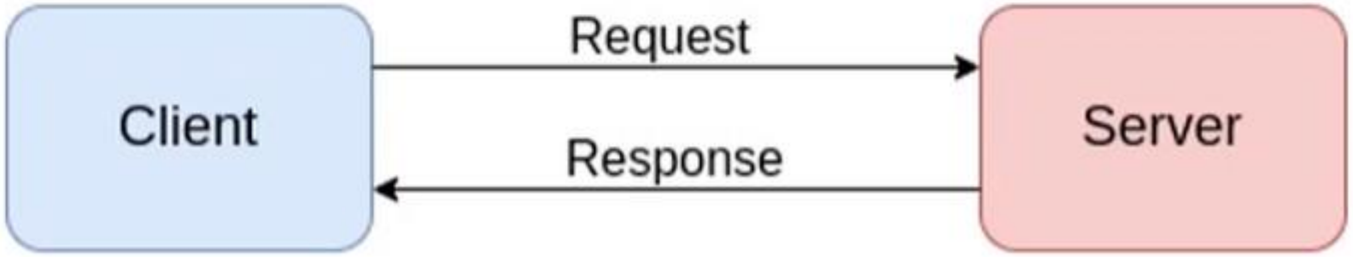


Questions?

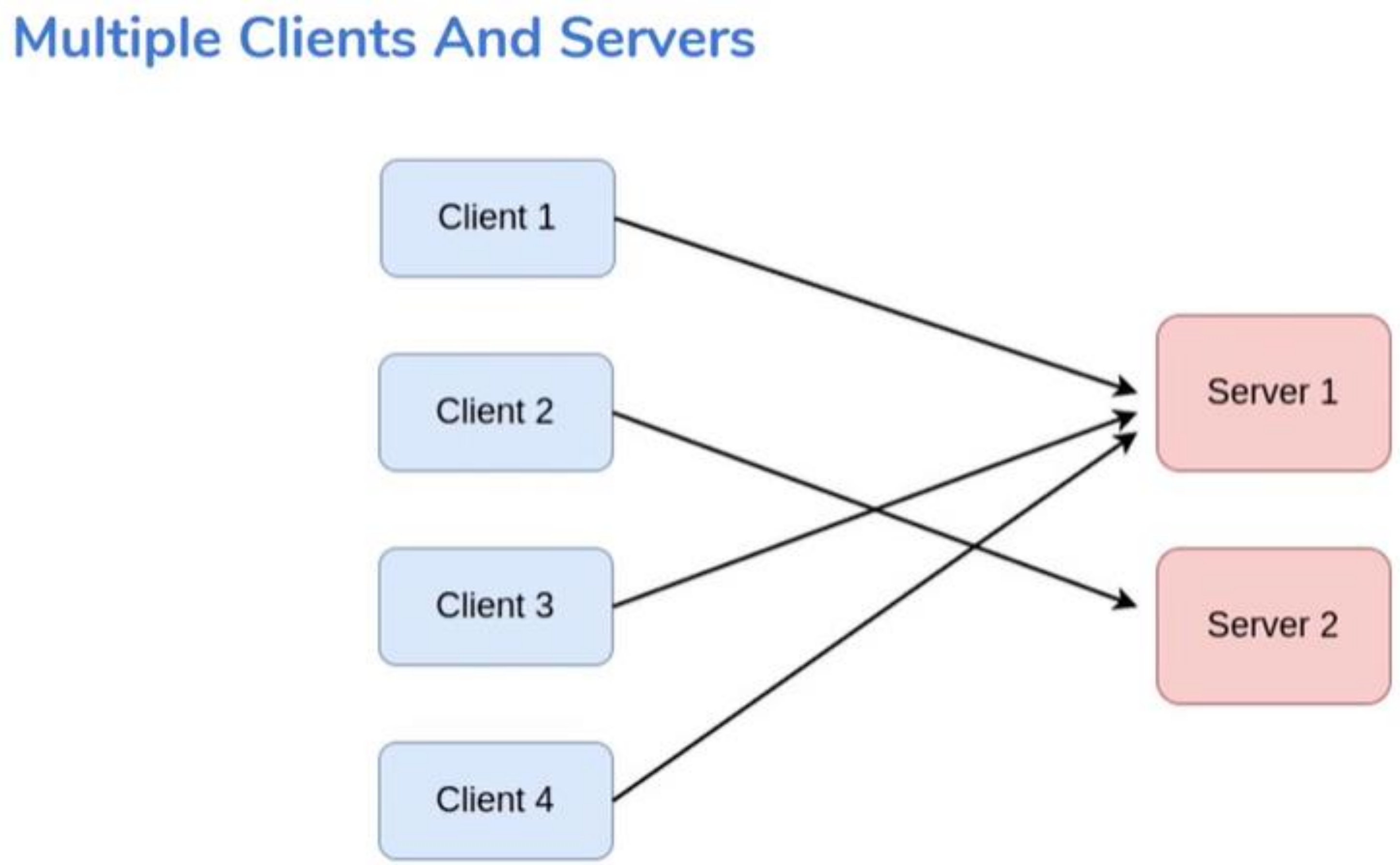
Architectural Patterns

Client Server

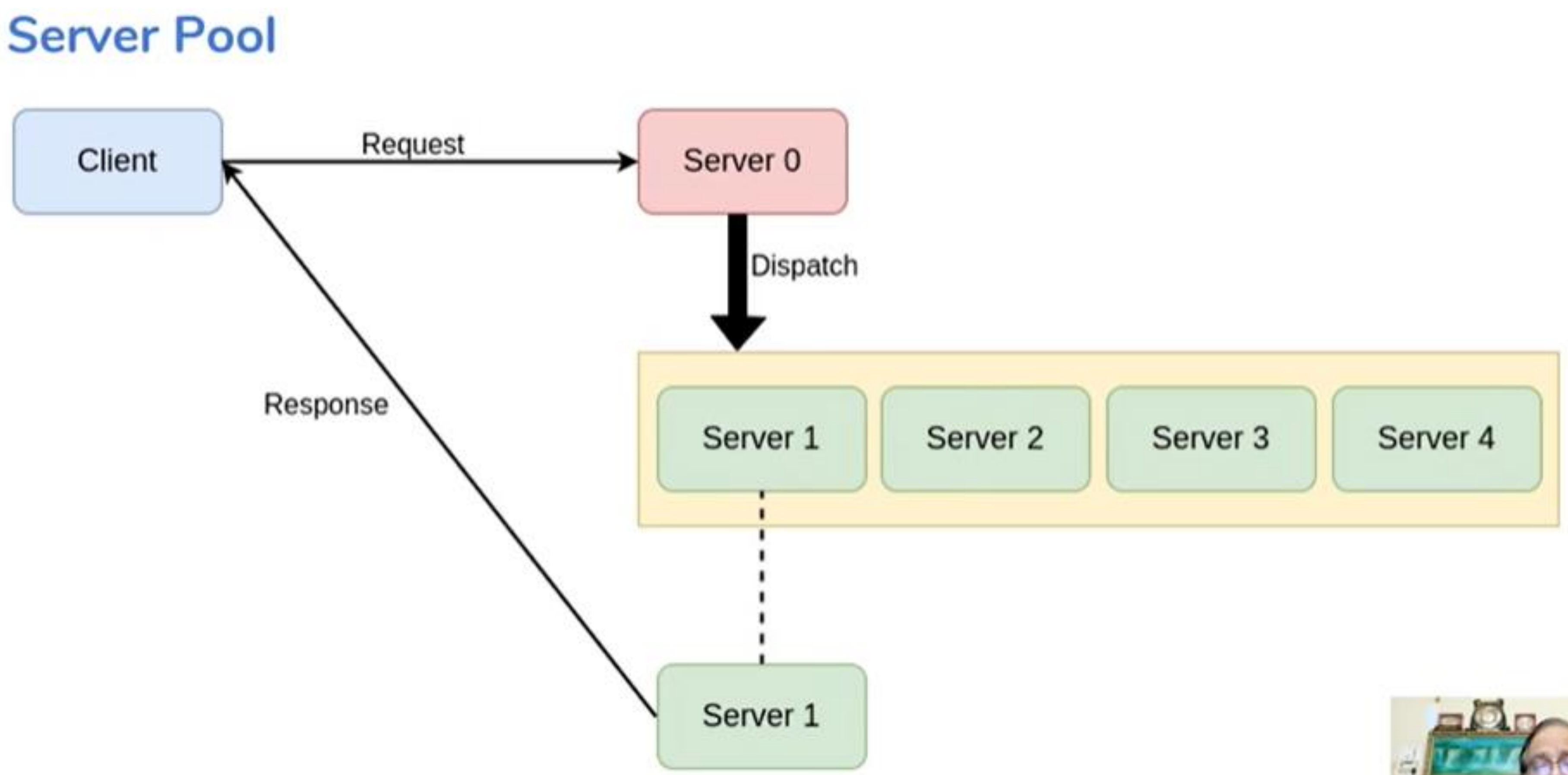
Single Client, Single Server



Client Server

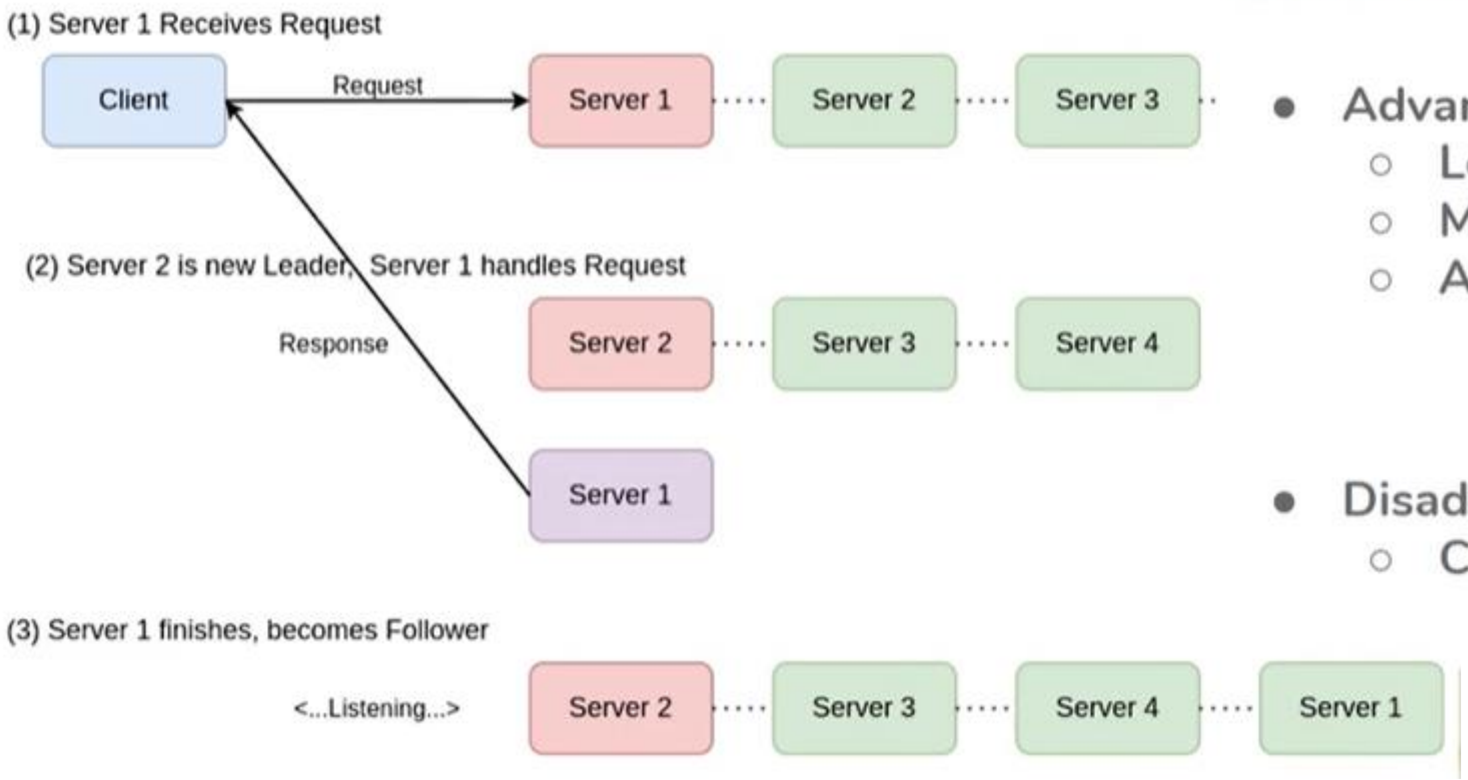


Client Server



Client Server

Leader - Follower



Leader - Follower

- Advantages:
  - Low latency
  - Minimal synchronization needed
  - All servers are equal
- Disadvantages:
  - Complex implementation

Client Server

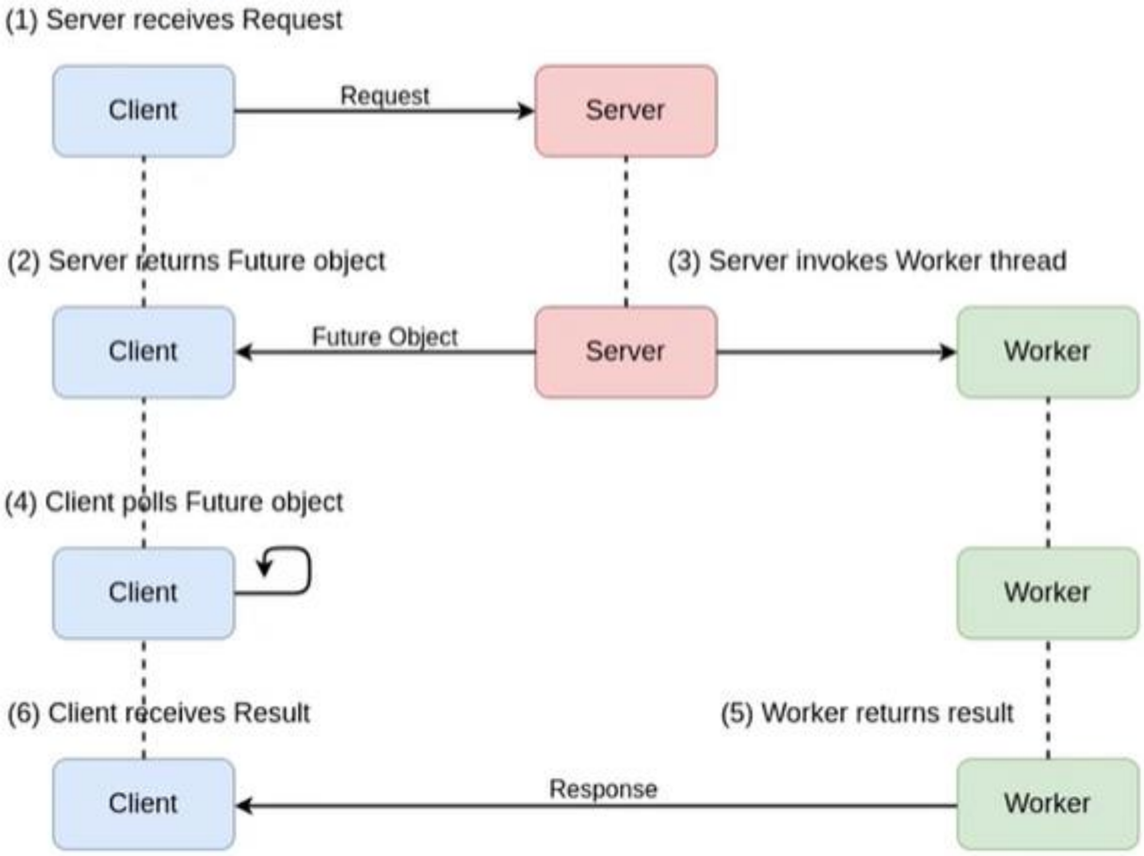
Server Pool

- Different Implementations:
  - Thread Pool
  - Proxy Server
  - Request Handoff

- Examples:
  - HTTP REST Servers
  - Java Socket Connections

Distributed Architecture Patterns

Half Sync - Half Async



Half Sync - Half Async

- Split tasks: High-level, Low-level
  - High-level Layer: Synchronous, simple
  - Low-level Layer: Asynchronous, efficient
  - Queueing Layer: Buffering point
- Examples
  - UNIX, Windows NT Design

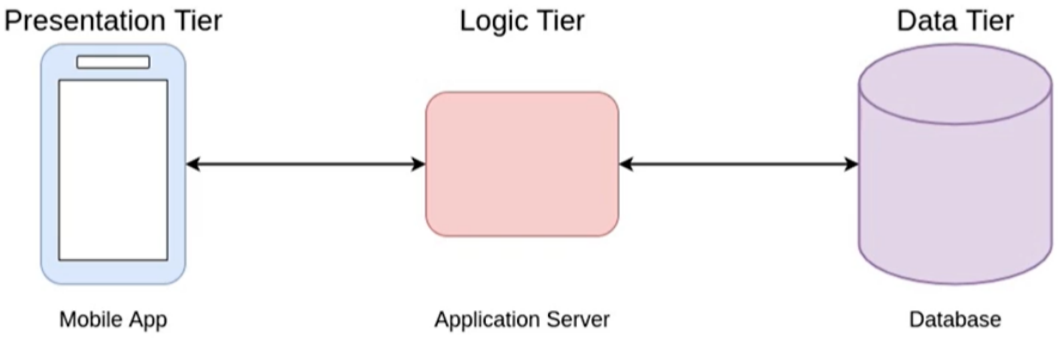
Half Sync - Half Async

- Advantages:
  - Simplified programming
  - Enhanced efficiency
  - Decoupled execution layers
- Disadvantages:
  - Additional synchronization, copying
  - Context-switch overhead

Network lock (NFS and AFS)

Multi-Tier Architecture Patterns

Multi-Tier Architectures



Multi-Tier Architectures

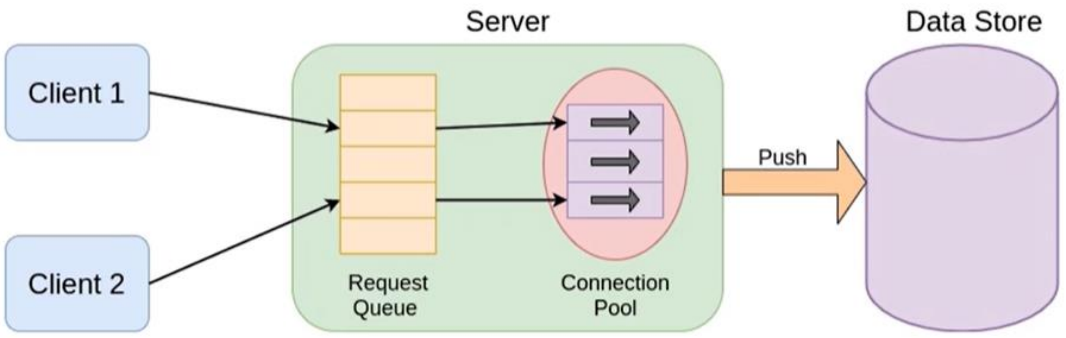
- Client-Server architecture with separate layers:
  - Presentation
  - Processing
  - Data Management
- Motivation:
  - Flexible and Reusable applications

Multi-Tier Architecture Patterns: Connection Pool

- Cache of Database Connections
  - Reusable for future requests
  - No connection creation overhead
  - Cuts down turnaround time
- Mechanisms:
  - Push-based
  - Pull-based

Connection (Push based)

Push-Based Connection Pool



Push-Based Connection Pool

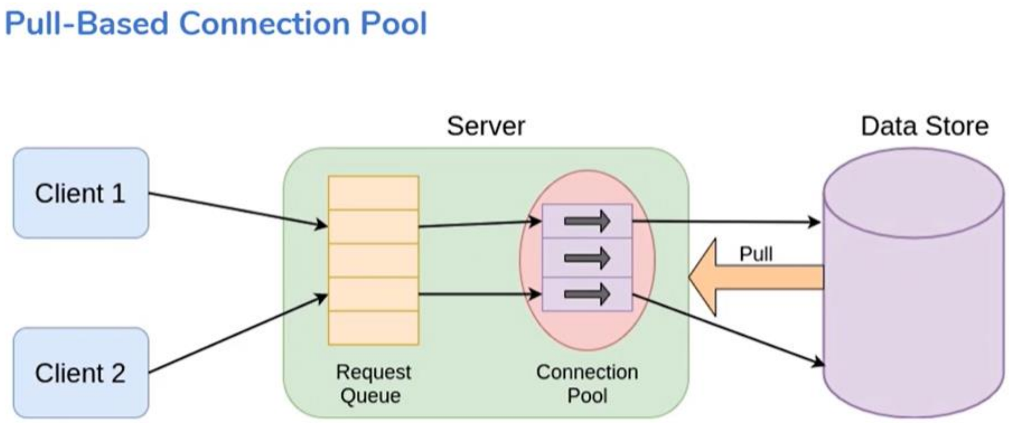
- Mechanism:
  - Server received Client Request
  - Server starts Transaction to Data Layer
  - Data Layer responds
  - Server dispatches Response

Push-Based Connection Pool

- Advantages:
  - Easy matching Response to Request
  - Simpler routing on Client

- Disadvantages:
  - Server needs to be proactive
  - Tricky balancing heavy loads

Connection (Pull based)



Publish-Subscribe Architecture (PubSub)

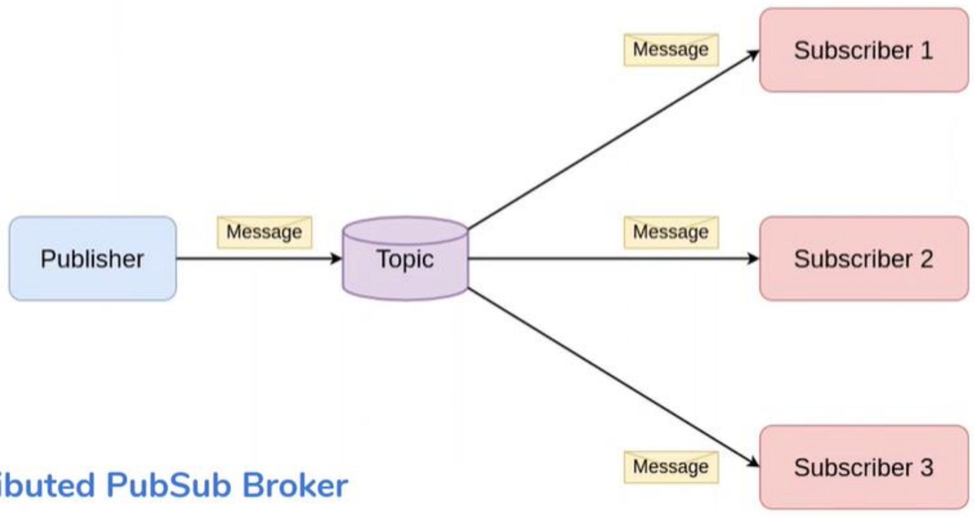
Publish-Subscribe Model (PubSub)

- A Messaging pattern:
  - Publishers: Create and categorize messages
  - Subscribers: Receive messages of interest
  - No direct interaction
- Mechanisms:
  - Topic-Based: Publish messages to logical channels
  - Content-based: Match message attributes to Subscriber

Publish-Subscribe Model

- Advantages:
  - Loose Coupling
  - Better Scalability through caching
- Disadvantages:
  - No timely delivery guarantees
  - Lack of coordination
  - Message Ordering - Processing failures and retries

Publish-Subscribe Model

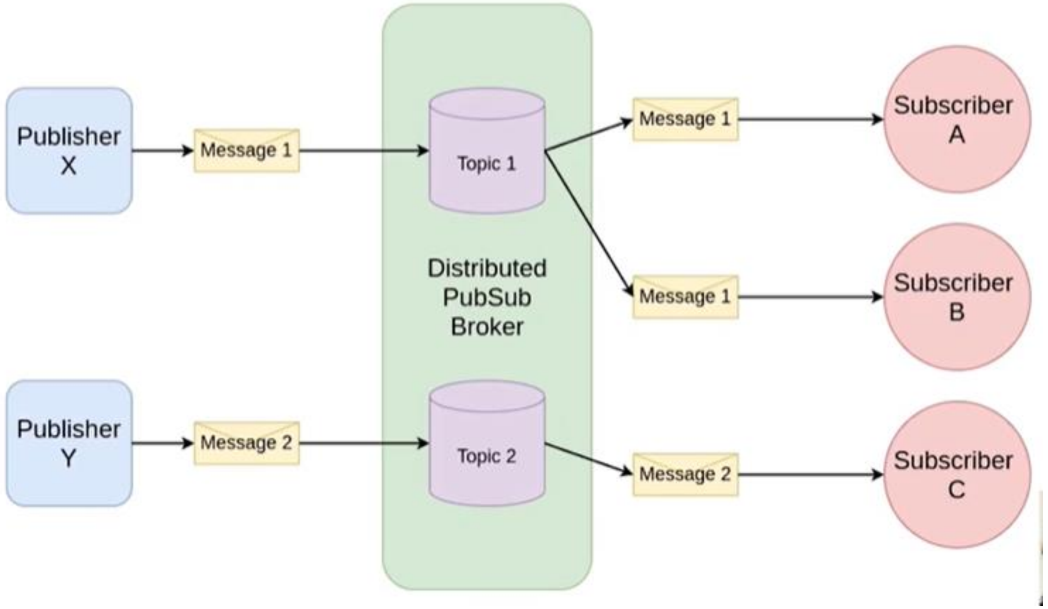


Distributed PubSub Broker

- Translates message from Sender to Receiver
- Other features:
  - Manages message queues
  - Provides message routing
  - Offers transaction management
- Real-world examples
  - Apache Kafka
  - RabbitMQ

Network lock (NFS and AFS)

Distributed Pub-Sub Broker



Example: Apache Kafka

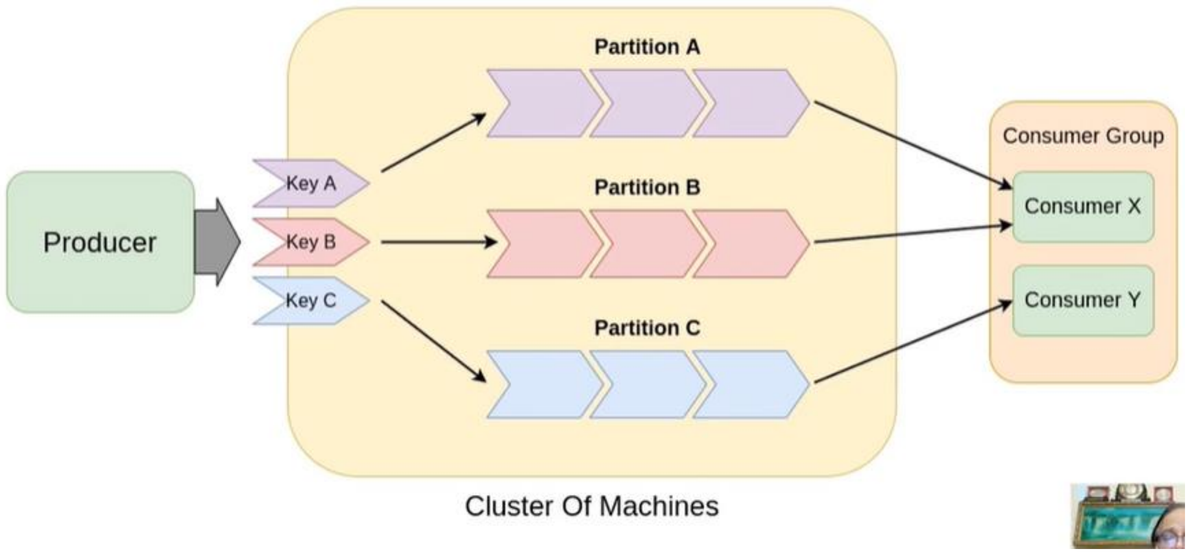
- Message stream processing system
  - Open source
  - Multiple producers, multiple consumers
  - Handles real-time data feeds

Distributed Pub-Sub Broker

- Advantages:
  - Guaranteed message delivery
  - Broker manages discovery
- Disadvantages:
  - Slower, high latency
  - Deployment and maintenance overhead

Kafka – Pub Sub example

Example: Kafka



Example: Kafka

- Stores key-value messages from producers
  - Split data into Partitions
  - Classify data by Topic
- Topics:
  - Treats later messages as updates to older ones
- Within a Partition:
  - Consumers read ordered messages

Example: Kafka

- The “Kafka Pattern”
  - Runs on a cluster of Brokers
  - Distributes Partitions of all Topics
  - Replicate Partitions across Brokers

Microservices Architecture

Microservices Architecture - Overview

- Decompose monolith to Microservices Architecture
- Mapping Microservices to containers - Docker
- Kubernetes as Orchestrator

Microservices Architecture

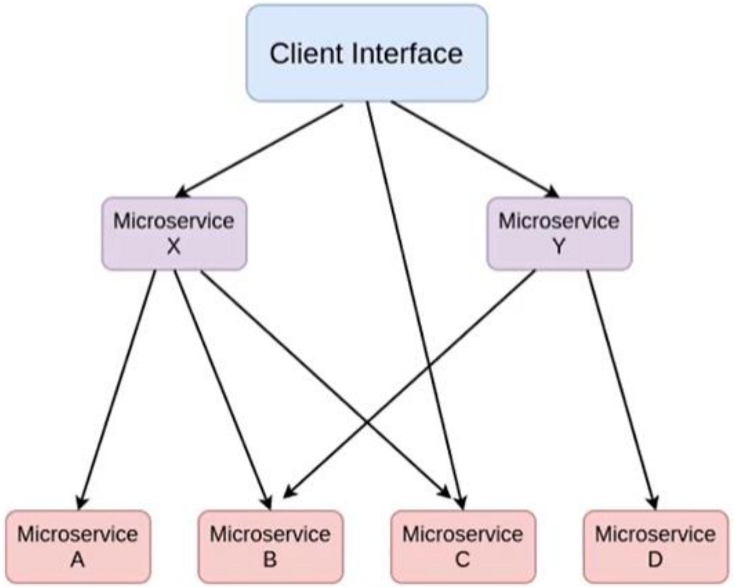
- Application: Collection of loosely-coupled Services
- Monolith: Single, large application
- Microservice:
  - Small size, bound by context
  - Messaging enabled
  - Independently developed and deployed

Microservices Architecture - When to apply

- Rapidly evolving business model - Application structure changes frequently
- Frequent application scaling - Apply functional decomposition
- Tangled dependencies in monolith - Difficult to evolve independently and to separate concerns

Network lock (NFS and AFS)

Microservices Architecture



Microservices Architecture

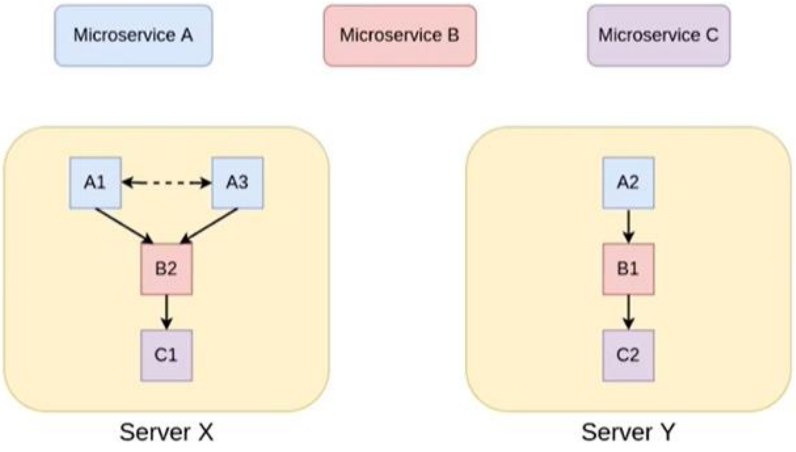
- Microservice Instances:
  - Service instance per Host
  - Service instance per Container

Microservices Architecture - How to decompose

- Business capabilities
- Domain-driven subsystems
- By verb or use-case responsible for specific action
- By noun or resources action on all operations on any given entities
- Classes of service should hold SRP(Single Responsibility Principle)

Network lock (NFS and AFS)

Microservices Architecture



Microservices Architecture - Benefits

- No single point of failure
- Better fault isolation
- Easier and more flexible deployment
- Improved testability
- Maintainability
- Multiple tech stacks can co-exist and evolve

Microservices Architecture - Drawbacks

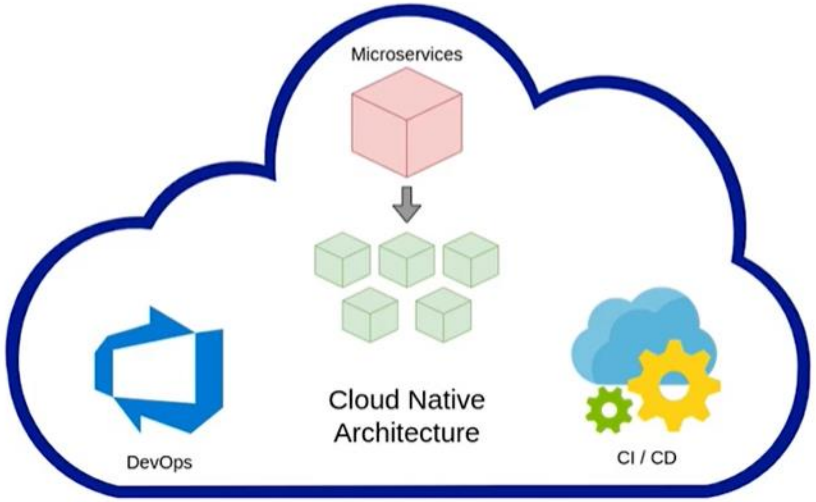
- Inherent complexity in a distributed architecture
- Handling requests spanning multiple services
- Testing interactions between services
- Addressing partial failures is difficult
- Increased memory consumption
- Deployment Complexity

Cloud Native Architecture

Cloud Native

Cloud Native

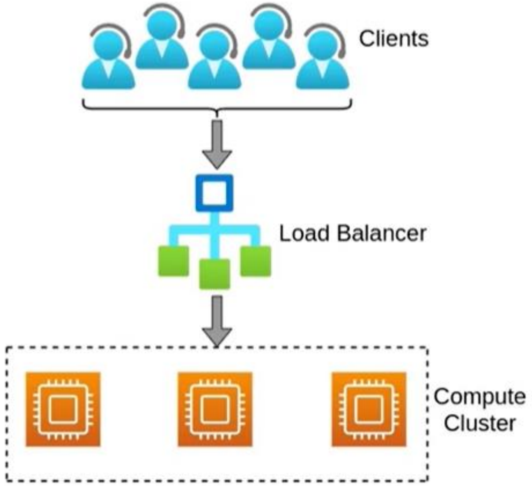
- Design for Automation
  - Scale up, Scale down
  - Monitoring and Recovery
- Service-based Load Balancing
- Managed Services



Cloud Native: Load Balancing

- Distribute workloads across compute resources
  - Scheduling Algorithms
  - Load Balancing Policies
- Service Based Load Balancing
  - Routes client requests to capable servers
  - Flexible to add / subtract servers
- Depends on nature of tasks

Cloud Native: Load Balancing

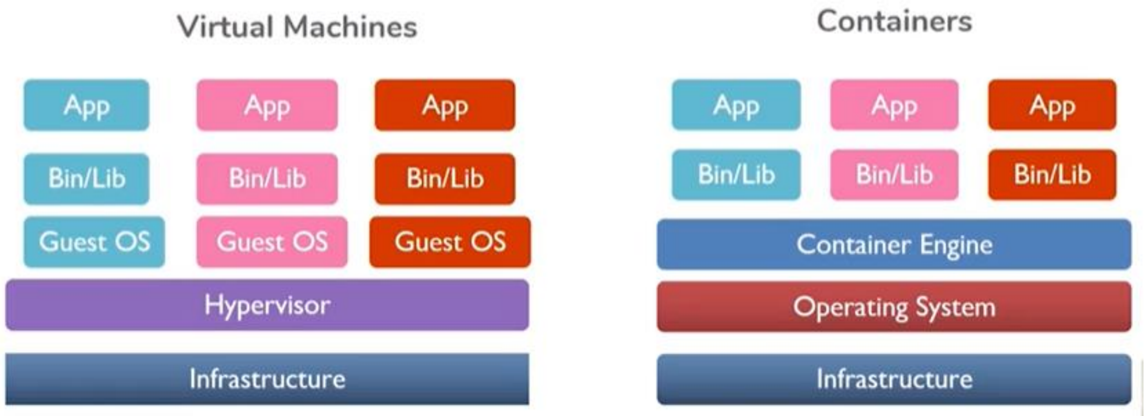


Cloud Native: Load Balancing

- Advantages
  - Scalability
  - High Availability of Services
  - Reliability
- Disadvantages
  - Latency increase during spikes
  - Inflexibility in pre-processing

Cloud Native Architecture

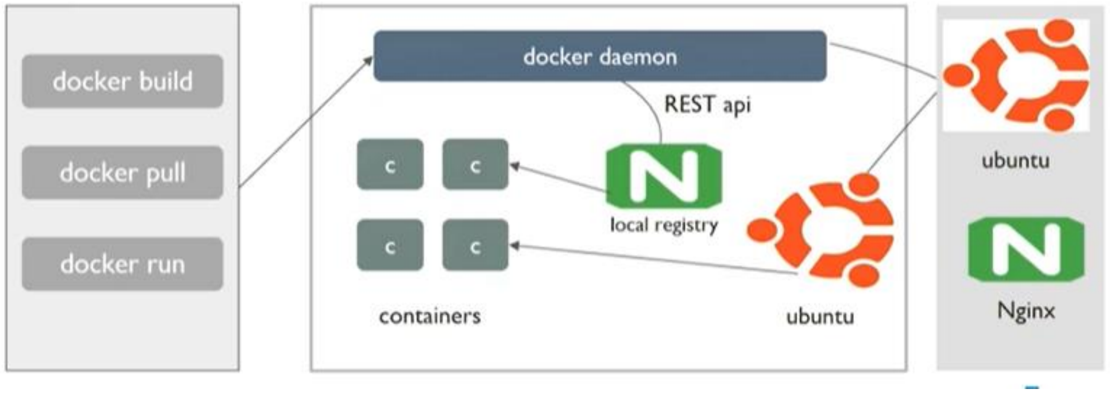
Containers - Docker



Containers - Docker

- Abstract app layer - Packages application and dependencies together
- Isolated workspace using linux namespaces
- pid, net, ipc, mnt, uns
- cgroups - limit the system resources
- union FS - thin file system layers including btfrs, AUFS, vfs and device mapper
- docker engine wraps namespaces, cgroups and union-fs in a container format, default is libcontainer

Containers - Docker Deployment



Containers - Docker and Microservices

- Task isolation - one container per microservice
- Supports multiple tech stacks
- Database separation - data volumes mounted as containers
- Automated monitoring - tools like prometheus, sysdig chisel & in

Container orchestration - Kubernetes

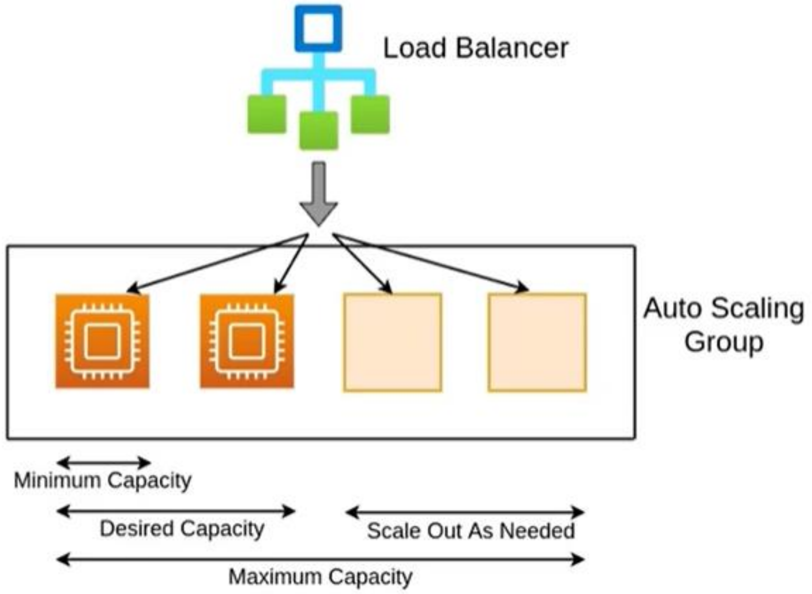
- Deployed as a cluster
- Includes worker nodes running containerized applications
- Control plane
  - kube-apiserver
  - etcd
  - kube-scheduler
  - kube-controller manager
  - cloud-controller-manager
- Compute Components
  - kubelet
  - kube-proxy
  - kubernetes CRI

Cloud Native Architecture

Cloud Native: Auto Scaling

- Compute Resources vary, depending on the load
- The make-up:
  - Auto Scaling Group
  - Server Instances
  - Desired Capacity
  - Metric
  - Scaling Policy

Cloud Native: Auto Scaling



Cloud Native: Auto Scaling

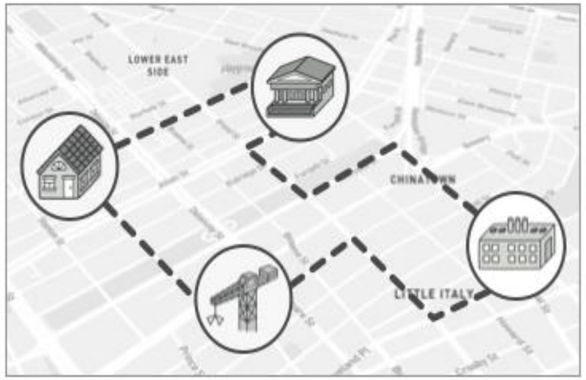
- Scaling Policy:
  - Updates the Group's desired capacity
  - Based on changes to the metric
- Metrics:
  - CPU utilization
  - Memory utilization
  - Network bandwidth

Scaling: Up and Down

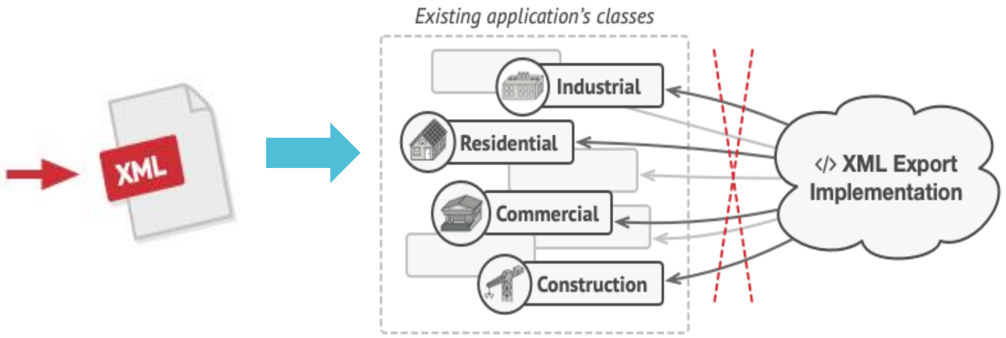
Cloud Native: Auto Scaling

- Advantages
  - Responsive to actual usage patterns
  - Reduces operational costs
  - Better availability
- Disadvantages
  - Hides application inefficiencies
  - Capacity thrashing possible if configured suboptimally

Visitor - Problem

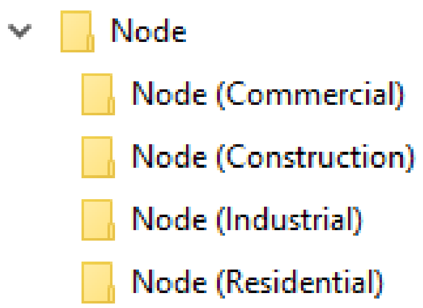


Exporting the graph into XML.



The XML export method had to be added into all node classes, which bore the risk of breaking the whole application if any bugs slipped through along with the change.

```
class ExportVisitor implements Visitor is
    method doForCity(City c) { ... }
    method doForIndustry(Industry f) { ... }
    method doForSightSeeing(SightSeeing ss) { ... }
    // ...
```




```
foreach (Node node in graph)
    if (node instanceof City)
        exportVisitor.doForCity((City) node)
    if (node instanceof Industry)
        exportVisitor.doForIndustry((Industry) node)
    // ...
}
```

With XML export, the actual implementation will probably be a little bit different across various node classes.

Thank you!