# COM S 362
# Object-Oriented Analysis & Design

Proxy, State and Visitor Patterns

## Reading

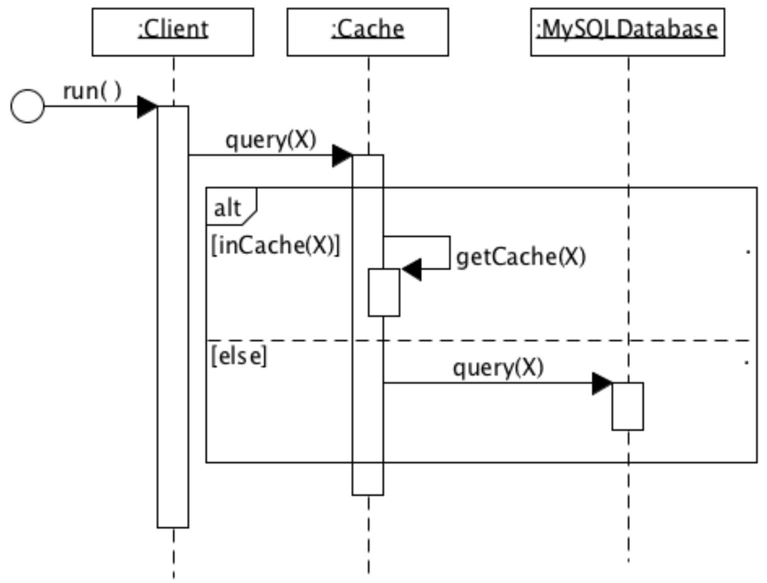
Alexander Shvets. Dive Into Design Patterns, 2020.

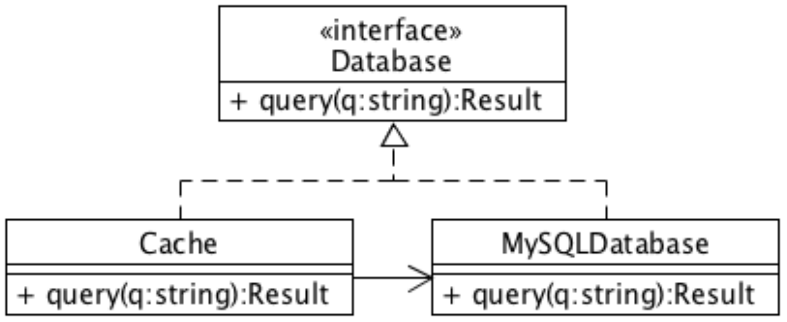
- Proxy
- State
- Visitor

## Structural Patterns Overview

- Adapter
  - Allows objects with incompatible interfaces to collaborate
- Bridge
  - Develop abstraction and implementation separately
- Composite
  - Work with trees of objects as individual objects
- Decorator
  - Attach behaviors to objects by placing them in wrapper objects
- Facade
  - Provide a simplified interface to a library or framework
- Flyweight
  - Fit more objects into limited RAM by sharing common parts
- Proxy
  - Provide a substitute or placeholder for another object

# Proxy Pattern: Problem/Solution

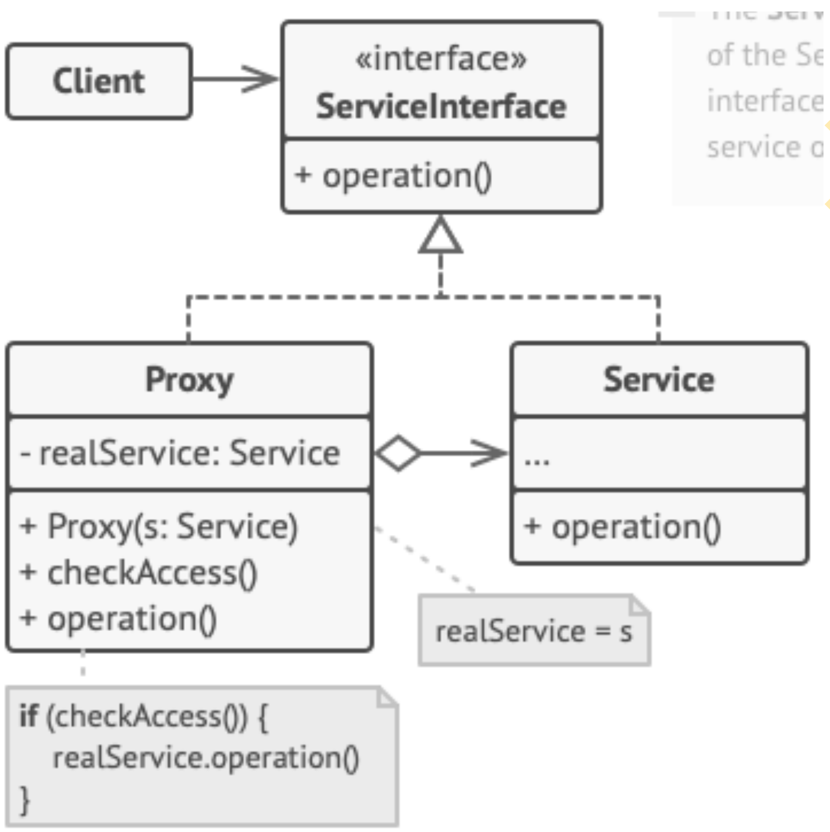




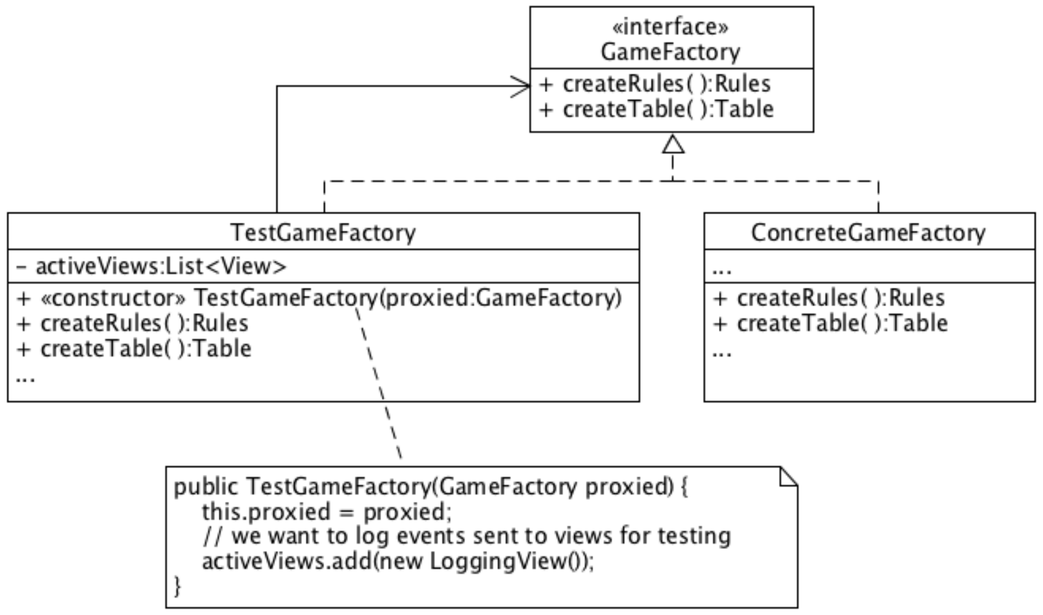
- Problem: You want to provide a cache that reduces the cost of frequent repeated lookups in the database
- Don’t want to rewrite the client to need to know about cache
- Solution: Create an object that has the same interface of the database, it uses the database, but has additional behavior of first looking in cache

## Proxy Pattern: Structure

- Client: only knows about interface
- Proxy: uses a concrete service to provide its behavior, adds additional behavior
- Service: the “real” concrete service where the proxy gets its information
- ServiceInterface: an interface is required so that Proxy and Service are interchangeable



# Proxy Pattern in Cards362



```
public TestGameFactory(GameFactory proxied) {
    this.proxied = proxied;
    // we want to log events sent to views for testing
    activeViews.add(new LoggingView());
}
```

- A common application for proxy is mock objects for testing
- Mock object uses concrete object to provide real behavior while instrumenting method calls for testing

## Proxy Pattern: Discussion

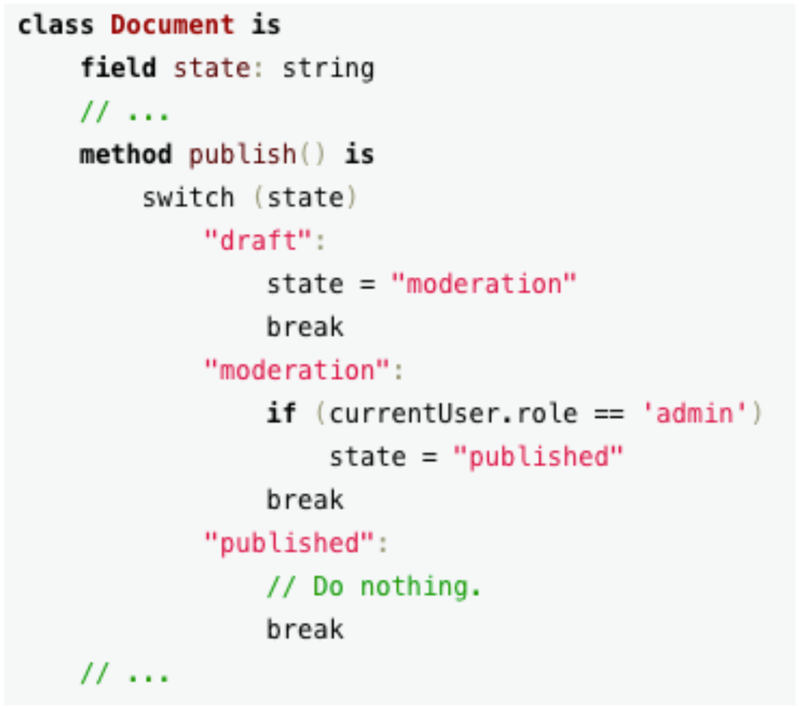
- Substitute a service object without the client knowing the details
- Good for providing cache, filters, resource pools, connection managers, mock objects, etc.
- Can connect multiple proxies in a line

## Behavioral Patterns

- Chain of Responsibility
  - Pass requests along a chain of handlers.
- Command
  - Turn a request (method calls) into an object.
- Iterator
  - Traverse elements of a collection.
- Mediator
  - Reduce dependencies between objects.
- Memento
  - Save and restore the state of an object.
- Observer
  - Publish and subscribe to events.
- State
  - State based behavior.
- Strategy
  - Define a family of algorithms, each in their own class
- Visitor
  - separate algorithms from the objects on which they operate

# State Pattern: Patterns/Solution

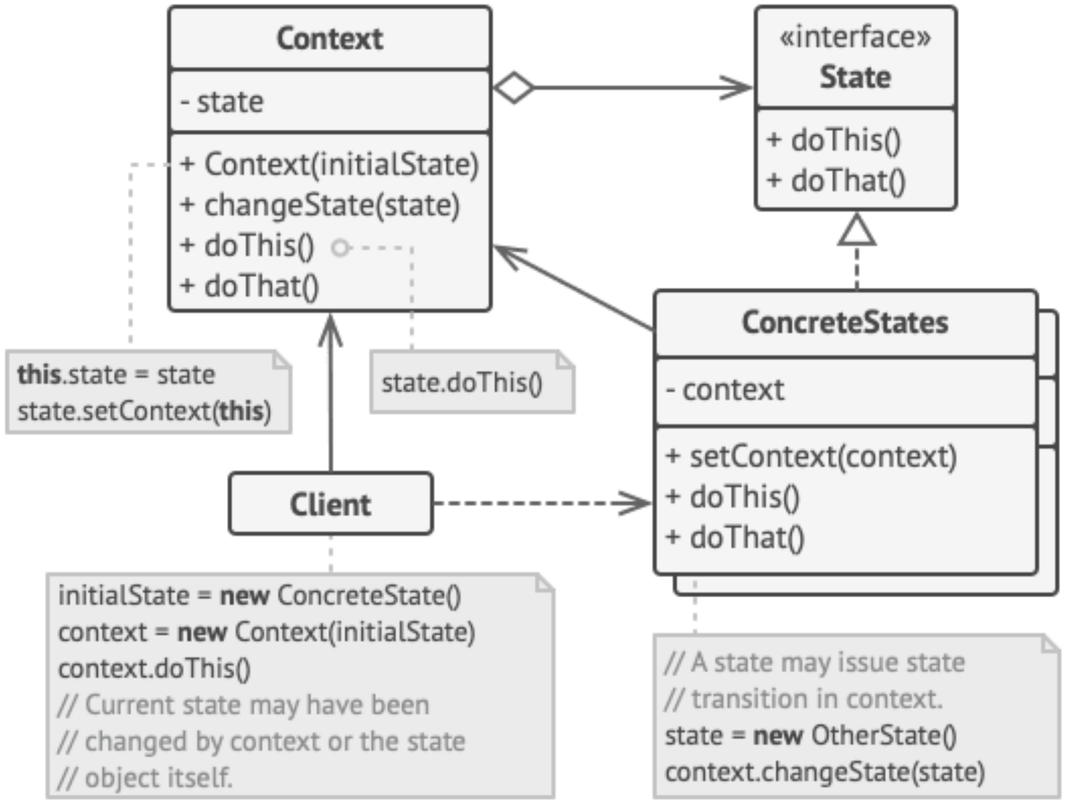
- Problem: You have a document editor that needs to change behavior based on the document state (e.g., draft, moderation or published)
- On the right is the classical setup for implementing a state machine in procedural code, it is not open-closed design
- Solution: Use objects to represent states and provide state specific behavior



```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == 'admin')
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
```

## State Pattern: Structure

- Context: knows current state, depends on state for behavior
- ConcreteStates: provides state specific behavior, knows what state to transition to
- State: interface to remove dependency on ConcreteStates



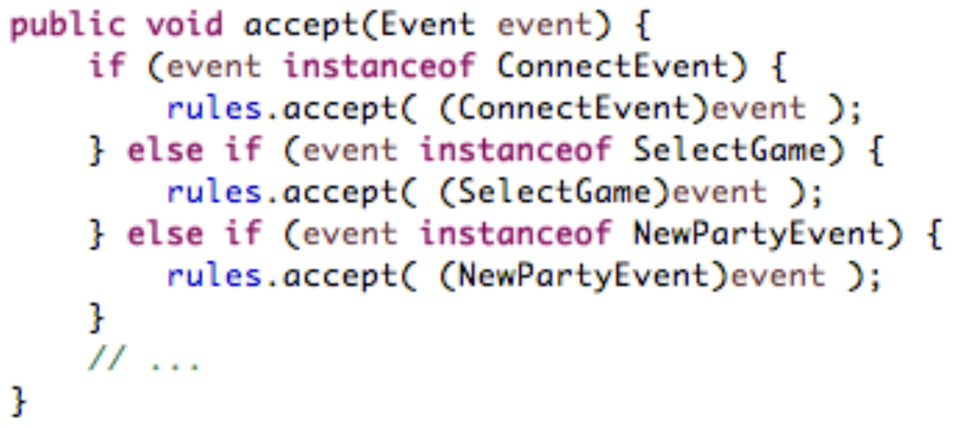
## State Pattern: Discussion

- Provides an object-oriented way to implement a state machine
- State specific behaviors are encapsulated in classes, making it consistent with open-closed principle

## Behavioral Patterns

- Chain of Responsibility
  - Pass requests along a chain of handlers.
- Command
  - Turn a request (method calls) into an object.
- Iterator
  - Traverse elements of a collection.
- Mediator
  - Reduce dependencies between objects.
- Memento
  - Save and restore the state of an object.
- Observer
  - Publish and subscribe to events.
- State
  - State based behavior.
- Strategy
  - Define a family of algorithms, each in their own class
- Visitor
  - separate algorithms from the objects on which they operate

## Visitor Pattern: Problem/Solution

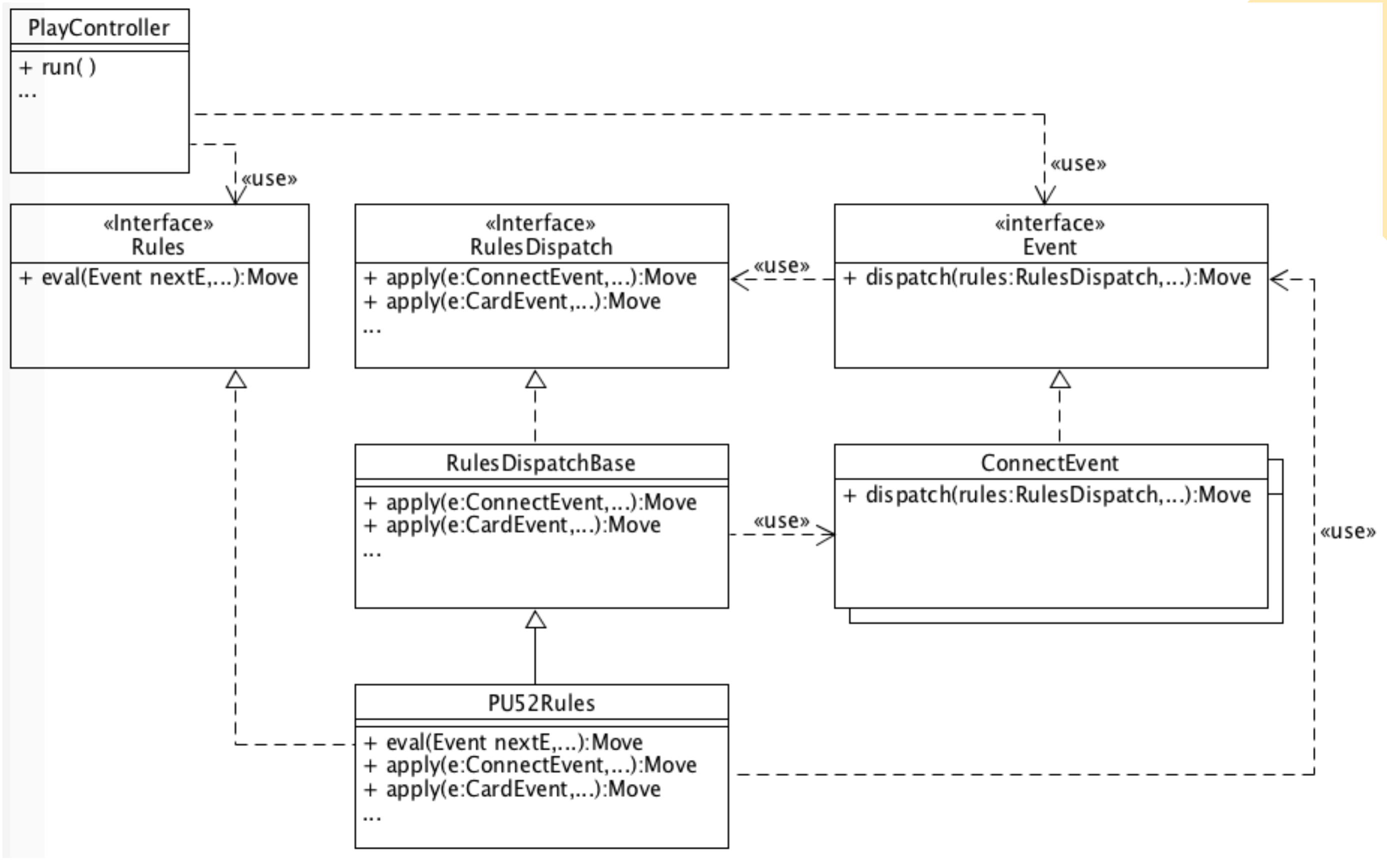


```
public void accept(Event event) {
    if (event instanceof ConnectEvent) {
        rules.accept( (ConnectEvent)event );
    } else if (event instanceof SelectGame) {
        rules.accept( (SelectGame)event );
    } else if (event instanceof NewPartyEvent) {
        rules.accept( (NewPartyEvent)event );
    }
    // ...
}
```

- Problem: You have an event driven system, events are objects and a client needs to apply them to a concrete rule object
- Want to avoid creating a large method that tests for each kind of event
- Solution: Add a small method to the Event interface, it calls back the Rule object, because the Event is a concrete object it can be matched to an overloaded method

# Visitor Pattern in Cards362

- RulesDispatch is a visitor interface
- PU52Rules is a concrete visitor
- A visitor overloads methods to visit different types of elements


PlayController
+ run( )
...
«use»
«use»
«Interface»
Rules
+ eval(Event nextE,...):Move
«Interface»
RulesDispatch
+ apply(e:ConnectEvent,...):Move
+ apply(e:CardEvent,...):Move
...
«use»
«interface»
Event
+ dispatch(rules:RulesDispatch,...):Move
RulesDispatchBase
+ apply(e:ConnectEvent,...):Move
+ apply(e:CardEvent,...):Move
...
«use»
ConnectEvent
+ dispatch(rules:RulesDispatch,...):Move
«use»
PU52Rules
+ eval(Event nextE,...):Move
+ apply(e:ConnectEvent,...):Move
+ apply(e:CardEvent,...):Move
...