Assignment No. 5

Title: Implement the State Design Pattern using Java.

Aim: Map the participants for the state design pattern from a given problem description and implement with a suitable object oriented language.

Objectives: From the given problem statement draw a class diagram for state design pattern and implement it

Pre-work: Prepare a class diagram from the given problem description using UML2.0 notations.

Theory:

- Intent
- Problem
- Discussion
- Structure (State Design Pattern Diagrammatic Representation)
- Context Object
- State
- Concrete State
- Summary

Output:

1. Example Problem Statement:
2. Represent Class Diagram:
3. Java Code

Intent

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

Problem

A monolithic object's behavior is a function of its state, and it must change its behavior at run-time depending on that state. Or, an application is characterized by large and numerous case statements that vector flow of control based on the state of the application.

Discussion

The State pattern is a solution to the problem of how to make behavior depend on state.

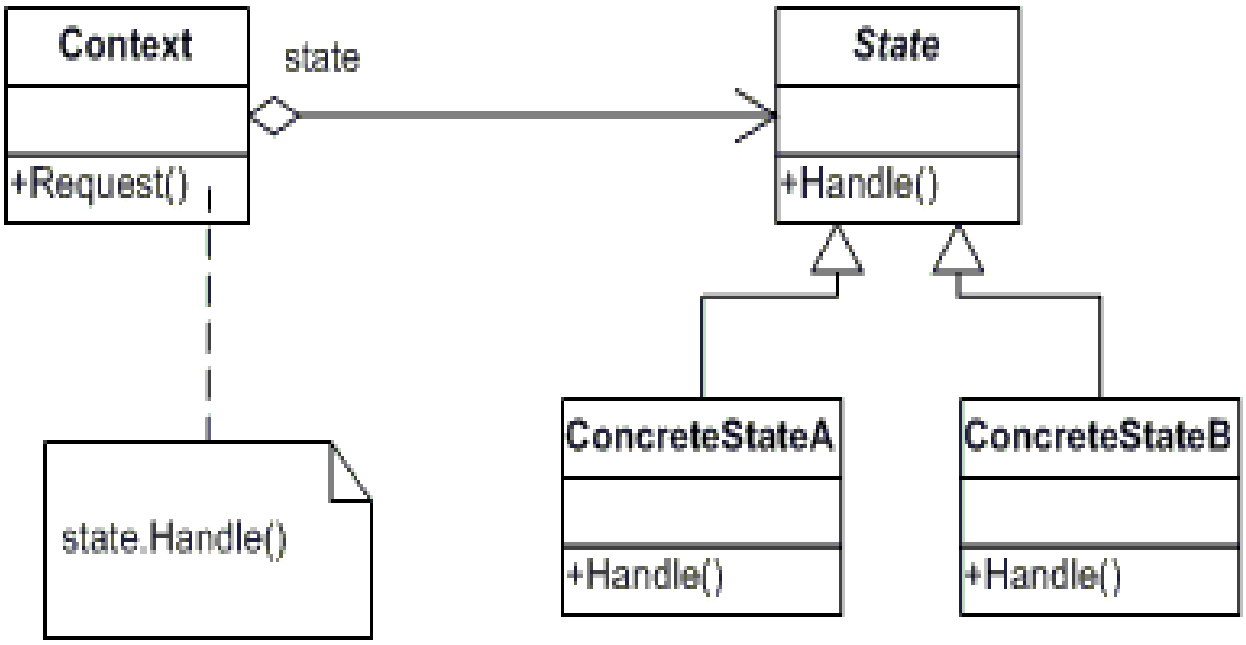
- Define a "context" class to present a single interface to the outside world.
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain a pointer to the current "state" in the "context" class.
- To change the state of the state machine, change the current "state" pointer.

The State pattern does not specify where the state transitions will be defined. The choices are two: the "context" object, or each individual State derived class. The advantage of the latter option is ease of adding new State derived classes. The disadvantage is each State derived class has knowledge of (coupling to) its siblings, which introduces dependencies between subclasses.

A table-driven approach to designing finite state machines does a good job of specifying state transitions, but it is difficult to add actions to accompany the state transitions. The pattern-based approach uses code (instead of data structures) to specify state transitions, but it does a good job of accommodating state transition actions.

Structure

The state machine's interface is encapsulated in the "wrapper" class. The wrappee hierarchy's interface mirrors the wrapper's interface with the exception of one additional parameter. The extra parameter allows wrappee derived classes to call back to the wrapper class as necessary. Complexity that would otherwise drag down the wrapper class is neatly compartmented and encapsulated in a polymorphic hierarchy to which the wrapper object delegates.



Context Object

Context is an instance of a class that owns (contains) the state. The context is an object that represents a thing that can have more than one state. In fact, it could have many different states. There is really no limit. It is perfectly fine to have

many possible state objects even into the hundreds. It is coming to have context objects with only a handful of possible states, though.

The Context object has at least one method to process requests and passes these requests along to the state objects for processing. The context has no clue on what the possible states are. The context must not be aware of the meaning of these different states. It is important that the context object does not do any manipulation of the states (no state changes). The only exception is that the context may set an initial state at startup and therefore must be aware of the existence of that initial state. This initial state can be set in code or come from an external configuration.

The only concern that the context has is to pass the request to the underlying state object for processing. The big advantage of not knowing what states the context could be in is that you can add as many new states as required over time. This makes maintaining the context super simple and super flexible. A true time saver and a step closer to being rich beyond your wildest dreams (almost).

State

The State class is an abstract class. It is usually an abstract class and not an interface (Interface). This class is the base class for all possible states. The reason why this class is usually an abstract class and not an interface is because there are usually common actions required to apply to all states. These global methods can be implemented in this base class. Since you can't do any implementation in Interfaces, abstract classes are perfect for this. Even if you do not have any initial global base methods, use abstract classes anyways because you never know if you might need base methods later on.

The State class defines all possible method signatures that all states must implement. This is extremely important to keep the maintenance of all possible states as simple as possible. Since all states will implement these methods signatures and if you forget to implement a new method, the compiler will warn you at compile time. An awesome safety net.

Concrete State

The Concrete State object implements the actual state behavior for the context object. It inherits from the base State class. The Concrete State class must implement all methods from the abstract base class State.

The Concrete State object has all the business knowledge required to make decisions about its state behavior. It makes decisions on when and how it should switch from one state to another. It has knowledge of other possible Concrete State objects so that it can switch to another state if required.

The Concrete State object can even check other context objects and their states to make business decisions. Many times, an object may have more than one context object. When this happens, a Concrete State object may need to access these different states and make a decision based on active states. This allows for complicated scenarios but fairly easy to implement using the state design pattern. You will see an example later in this article that shows multiple context objects and their states and the need to work together.

The Concrete State object also is capable of handling before and after transitioning to states. Being aware of a transition about to happen is an extremely powerful feature. For example, this can be used for logging, audit recording, security, firing off external services, kicking of workflows, etc. and many other purposes.

The Concrete State object allows the full use of a programming language when compared to state machines. Nothing is more powerful in abstract logic and conditionals coupled with object orientation as a computer programming language compared to state machines and their implementations.

As you add new methods to the abstract base class over time, each Concrete State class will need to implement that method. This forces you to think from the point of view of the current state.

"How should state ConcreteStateA react when this method is called?"

As you implement the behavior for a method, you can be rest assured that this is the only place in the entire system that will handle this request when

ConcreteStateA is the active state. You know exactly where to go to maintain that code. Maintainability is king in software development.

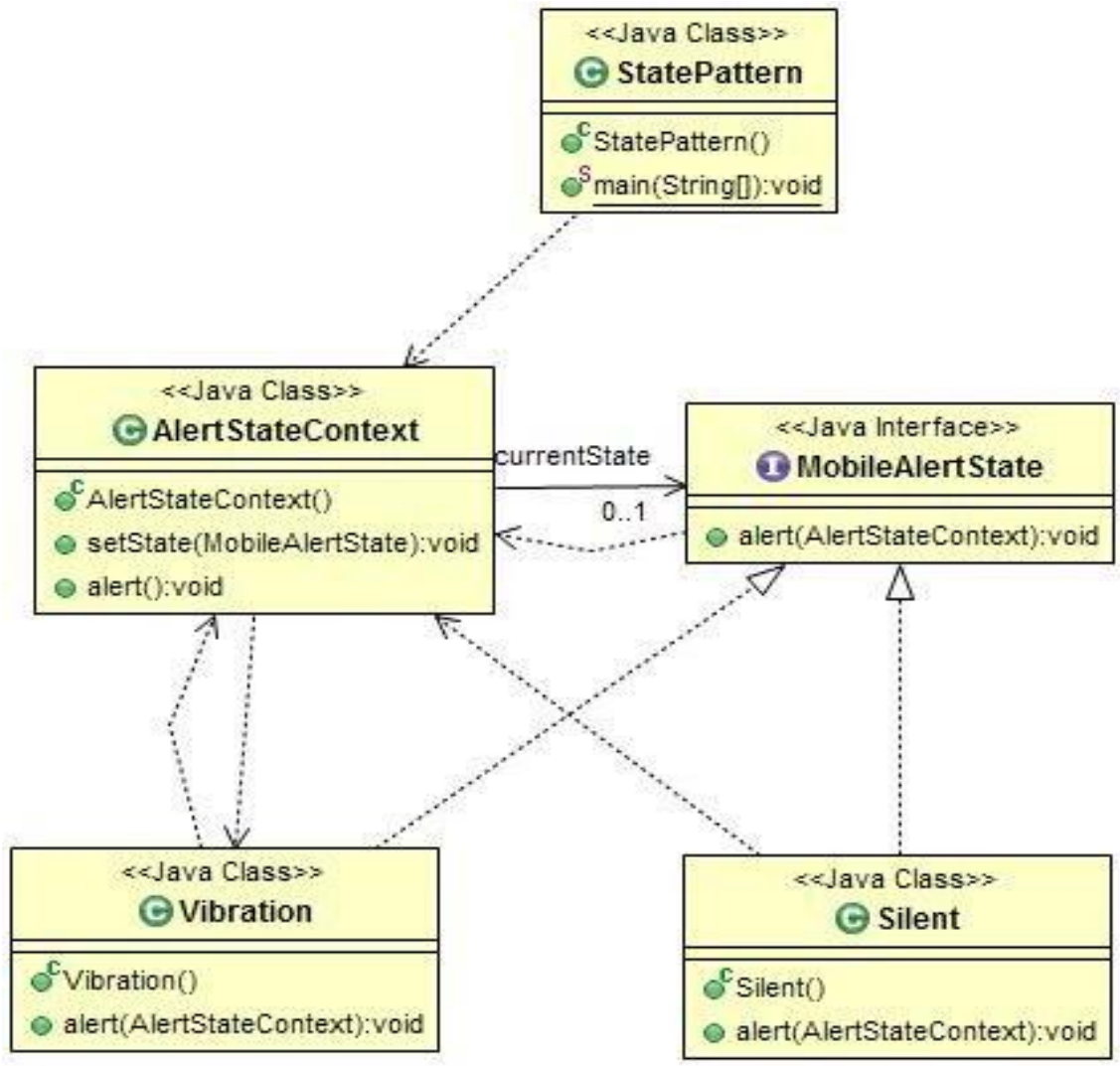
Summary

To summarize, you will need a context and a few states that ideally derive from an abstract base class to create a flexible state solution. If you got switch statements or a lot of If statements in your code, you got opportunities to simplify by using the state design pattern. If you are using state machines, you got an awesome opportunity to simplify your code and safe time and money.

Example 1:

Consider scenario using a mobile. With respect to alerts, a mobile can be in different states. For example, vibration and silent. Based on this alert state, behavior of the mobile changes when an alert is to be done.

Class Diagram:



Java Code:

Example 2:

Suppose we want to implement a TV Remote with a simple button to perform action, if the State is ON, it will turn on the TV and if state is OFF, it will turn off the TV.

Class Diagram:

Java Code