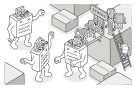
Refactoring Guru uses cookies to improve your experience. We hope you agree with this, as some of these cookies are necessary for the site to function properly. Please read our Cookie Policy for more details. Agree and Close

State

Intent

State is a behavioral design pattern that lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.



Problem

The State pattern is strongly related to the concept of a Finite State Machine.

Finite State Machine

The main idea is that, at any given moment, there's a finite number of states which a program can be in. Within any unique state, the program behaves differently, and the program may be switched from one state to another instantaneously. However, depending on a current state, the program may or may not switch to other states. These switching rules, called transitions, are also finite and predetermined.

You can also apply this approach to objects. Imagine that we have a `Document` class. A document can be in one of three states: `Draft`, `Moderation` and `Published`. The `ChangeRight()` method of the document works a little bit differently in each state:

- In `Draft`, it moves the document to `Moderation`.
- In `Moderation`, it makes the document public, but only if the current user is an administrator.
- In `Published`, it doesn't do anything yet.

Allowed states and conditions for a document object

State machines are usually implemented with lots of conditional operators (`if` or `switch`) that select the appropriate behavior depending on the current state of the object. Usually, this "state" is just a set of values of the object's fields. Even if you hadn't heard about finite state machines before, you've probably implemented a state at least once. Does this following code structure ring a bell?

```
class Document
    text: string
    primaryLine: int
    secondLine: int
    method lineAt(int n): int
        if (primaryLine == n)
            return primaryLine
        else if (secondLine == n)
            return secondLine
        else if (primaryLine > n && secondLine > n)
            return null
        else
            return primaryLine
    method ChangeRight()
        if (primaryLine == 1 && secondLine == 2)
            return
    // ...
```

The biggest weakness of a state machine based on conditionals reveals itself once we start adding more and more states and state-dependent behavior to the `Document` class. Most methods will contain monstrous conditionals that pick the proper behavior of a method according to the current state. Code like this is very difficult to maintain because every change to the transition logic may require changing state conditionals in every method.

The problem tends to get bigger as a project evolves. It's quite difficult to predict all possible states and transitions at the design stage. Hence, a text-state machine built with a primitive set of conditionals can grow into a bloated mess over time.

Solution

The State pattern suggests that you create new classes for all possible states of an object and extract all state-specific behaviors into these classes.

Instead of implementing all behaviors on its own, the original object, called context, stores a reference to one of the state objects that represents the current state, and delegates all the state-related work to that object.

Machine's delegation mechanism in a vending machine

To transition into a certain state, another state, replace the active state object with another object that represents the new state. This is possible only if all state classes follow the same interface and the context doesn't care about these objects through that interface.

This structure may look similar to the Strategy pattern, but there's one key difference. In the State pattern, the particular states may be aware of each other and initiate transitions from one state to another, whereas strategies almost never know about each other.

Real-World Analogy

The business unit workflow in great simplifying behaves differently depending on the current status of the device:

- When the phone is undamaged, pressing fixes leads to assessing various functions.
- When the phone is turned, pressing any fixes leads to the unlock screen.
- When the phone's charge is low, pressing any button shows the charging screen.

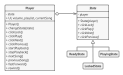
Structure

1. **Context** stores a reference to one of the concrete state objects and delegates to it all state-specific work. The context communicates with the state object via the state interface. The context exposes a setter that passing in a new state object.
2. The **State** interface declares the state-specific methods. These methods should make sense for all concrete states because you don't want some of your states to have useless methods that will in turn be called.
3. **Concrete States** provide their own implementations for the state-specific methods. To avoid duplication of similar code across multiple states, you may provide intermediate abstract classes that encapsulate some common behavior.

   State objects may store a backreference to the context object. Through this reference, the state can fetch any required info from the context object, as well as initiate state transitions.
4. Both context and concrete states can set the next state of this context and perform the actual state transition by replacing the state object linked to the context.

Pseudocode

In this example, the **Game** player's turn has some contents of the media player behave differently, depending on the current playback state.



Primitive UI changing depending on the current playback state

The main object of this player is always stored in a state object that performs most of the work for the player. Some actions replace the current state object of the player with another, which changes the way the player reacts to user interactions.

```
// The AudioPlayer class acts as a context. It also maintains a
// reference to an instance of one of the state classes that
// represents the current state of the media player.
class AudioPlayer
    UI ui
    int volume, int[] playlist, currentSong

    constructor AudioPlayer() ->
        this.ui = new UserInterface()
        // ...other initializations...
        // Initial state is set to locked.
        // Which necessitates the creation of each state
        // in constructor. Alternatively, you can create states on demand
        // when they are needed.
        ui.lockButton.setOnClick(this.clickLock())
        ui.playButton.setOnClick(this.clickPlay())
        ui.nextButton.setOnClick(this.clickNext())
        ui.previousButton.setOnClick(this.clickPrevious())

    // Other objects must be able to switch the audio player's
    // active state.
    method changeUI(UserInterface ui): void
        this.ui = ui

    // All actual programs execution is the active state.
    method clickLock():
        ui.setLockIcon(true)
    method clickPlay():
        ui.setPlayBtnCaption("Pause")
    method clickNext():
        ui.setSongTitle("Next song")
    method clickPrevious():
        ui.setSongTitle("Previous song")
    method startPlayback():
        // ...
    method stopPlayback():
        // ...
    method nextSong():
        // ...
    method previousSong():
        // ...
    method getSong():
        // ...
    method setVolume(int level):
        // ...

// We have state classes declared below that all implement
// common interface implement each of the methods's behaviors in
// context of the audio player being in this state. Sometimes, in order
// the functionality to transition the context to another state,
// state classes.
abstract class State
    protected AudioPlayer player;

    // Extended method allows context through this state parameters. This
    // way both a state field and some method context were if it's
    // required.
    constructor State(player) ->
        this.player = player

    abstract method clickLock();
    abstract method clickPlay();
    abstract method clickNext();
    abstract method clickPrevious();

    // Locking the volume regardless various behaviors associated with a
    // locked player.
class LockedPlayer extends State ->
    // When you attach a locked player, it may execute all of the
    // commands.
    override clickLock() ->
        if (player.volume > 0)
            player.lastVolume = player.volume;
        else
            player.lastVolume = new AudioPlayer.DefaultVolume;
        // Unlocking player.
        player.volume = player.lastVolume
        // Switch to playing state.
        var playingState = new PlayingState(player);
        player.changeUI(playingState.ui);
    override clickPlay() ->
        // Cannot play when locked.
    override clickNext() ->
        // Cannot do anything.
    override clickPrevious() ->
        // Cannot do anything.

// Playing state allows toggling between lock and pause.
class PlayingState extends State ->
    override clickLock() ->
        var lockedState = new LockedState(player);
        player.volume = 0;
        player.changeUI(lockedState.ui);
    override clickPlay() ->
        var pausedState = new PausedState(player);
        player.changeUI(pausedState.ui);
    override clickNext() ->
        // Switch to next track.
    override clickPrevious() ->
        // Switch to previous track.

class PausedState extends State ->
    override clickLock() ->
        var lockedState = new LockedState(player);
        player.volume = 0;
        player.changeUI(lockedState.ui);
    override clickPlay() ->
        var playingState = new PlayingState(player);
        player.changeUI(playingState.ui);
    override clickNext() ->
        // Switch to next track.
    override clickPrevious() ->
        // Switch to previous track.

class Stopped extends State ->
    override clickLock() ->
        var lockedState = new LockedState(player);
        player.volume = 0;
        player.changeUI(lockedState.ui);
    override clickPlay() ->
        var playingState = new PlayingState(player);
        player.changeUI(playingState.ui);
    override clickNext() ->
        // Switch to next track.
    override clickPrevious() ->
        // Switch to previous track.
```

Applicability

Use the State pattern when you have an object that behaves differently depending on its current state, the number of states is enormous, and the state-specific code changes frequently.

The pattern suggests that you extract all state-specific code into a set of distinct classes. As a result, you can add new states or change existing ones independently of each other, reducing the maintenance cost.

Use the pattern when you have a class polluted with massive conditionals that alter how this class behaves according to the current values of this class's fields.

The State pattern lets you extract branches of these conditionals into methods of corresponding state classes. While doing so, you can also clean temporary fields and helper methods involved in state-specific code out of your main class.

Use State when you have a lot of duplicate code across similar states and transitions of a condition-based state machine.

The State pattern lets you compose hierarchies of state classes and reduce duplication by extracting common code into abstract base classes.

How to Implement

1. Decide what class will act as the context. It could be an existing class which actually has the state dependent code, or a new class, if the state-specific code is distributed across multiple classes.
2. Declare the state interface. Although it may mirror all the methods declared in the context, aim only for those that may contain state-specific behavior.
3. For every actual state, create a class that stores those the state interface. Then go over the methods of the context and extract all code related to that state into your newly created class.

   While moving the code to the state class, you might discover that it depends on private members of the context. There are several workarounds:

   - Make these fields or methods public
   - Turn the behavior you're extracting into a public method in the context and call it from the state class. This way is ugly but quick, and you can always fix it later.
   - Nest the state classes into the context class, but only if your programming language supports nesting classes.
4. In the context class, add a reference field of the state interface type and a public setter that allows overriding the value of that field.
5. Go over the methods of the context again and replace empty state conditionals with calls to corresponding methods of the state object.
6. To switch the state of the context, create an instance of one of the state classes and pass it to the context. You can do this within the context itself, or in various states, or in the client. Wherever this is done, the class becomes dependent on the concrete state class it instantiates.

Pros and Cons

- Single Responsibility Principle. Organize the code related to particular states into separate classes.
- Open/Closed Principle. Introduce new states without changing existing state classes or the context.
- Simplify the code of the context by eliminating bulky state machine conditionals.
- Applying this pattern can be overkill if a state machine has only a few states or rarely changes.

Relations with Other Patterns

- **Bridge**, **State**, **Strategy** (and to some degree **Adapter**) have very similar structures. Indeed, all of these patterns are based on composition, which is changing work to other objects. However, they all solve different problems. A pattern isn't just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.
- **State** can be considered as an extension of **Strategy**. Both patterns are based on composition: they change the behavior of the context by delegating some work to helper objects. Strategy makes these objects completely independent and unaware of each other. However, State doesn't restrict dependencies between concrete states, letting them alter the state of the context at will.

Code Examples

Support our free website and own the eBook!
- 65 design patterns and 8 principles explained in detail
- 1400 pages
- With audio/video lectures, easy to read, jargon-free pages
- Lifetime updates, 24/7 support, 30-day money-back guarantee
- Not a PDF but DRM-free ebook available in 5 languages
- Free version of the book is always available for the human users

Support us!

© 2014-2022 Refactoring.Guru. All rights reserved.
Illustrations by Dmitry Zhart
Refactoring.Guru, OOP.Guru, UI.Guru and UMLDistillery.Guru are trademarks of Refactoring.Guru.
Terms & Conditions
Privacy Policy
Content Usage Policy