Purpose / Motivation:

The main purpose of the state design pattern is to allow objects to change their behaviour based on the current state. The state information is determined at and can be changed at runtime, which allows behaviour to be modified dynamically, making it seem as if the object has changed classes.

Intended use cases:

The state design pattern can be used in the following two use cases:

1. A monolithic class whose behaviour depends on runtime information. This would be a class that has many fields representing the runtime state and methods that are very large because they perform different things depending on what the state information is.
2. An application that has large/numerous case statements which change flow of control based on runtime state information. Not necessarily a class, but any sort of application that has a lot of switch/case/if/else statements based on a set of variables can be transformed to use the state design pattern.

Vocabulary:

Programs that use the state design pattern consist of a context class, a state interface, and several concrete state classes. The context class represents the object whose behaviour depends on its internal state. Therefore, the context class has a field that is a state interface. The state interface abstracts properties and behaviour common to all states in which the context can be. Each of these states is represented by a concrete class that implements the interface. Any behaviour specific to a certain state is defined only in the concrete class for that state.

Structure / Runtime Behaviour:

See *Figure 1* for a graphical representation of the state design pattern.

Runtime behaviour of the state pattern is determined by the context which contains the state and which concrete state it is currently maintaining. Generally the context will begin by initializing a pre-defined internal state, and new states will be transitioned to depending on the current state and rules governing transitions from the current state.

Implementation of state transition rules can either be contained directly in the concrete sub-classes of the state interface, or by creating a state transition table which is used to look up subsequent states. That is, each state knows explicitly the conditions under which it should transition, and which state specifically it should transition to; or each concrete state is agnostic as to the conditions under which it should transition, and which states it could transition to, instead delegating to the state transition table to govern these interactions.

Using a transition table serves to reduce dependencies between concrete states and creates a central point of control for governing state transitions, but has the effect of reducing explicitness by making transitions between specific states less obvious. Note that using a table implementation does not necessarily eliminate coupling between states, however it may reduce coupling to a degree, as states may only have to refer to or calculate a location in a table, instead of having to depend on a direct reference to another concrete state. That is, the coupling is now contained inside of the transition table, instead of the concrete state objects. Implementations that involve a state transition table are also marginally less efficient than those that place state transition rules in concrete state classes because the program must look up the transition it should make in response to each event.

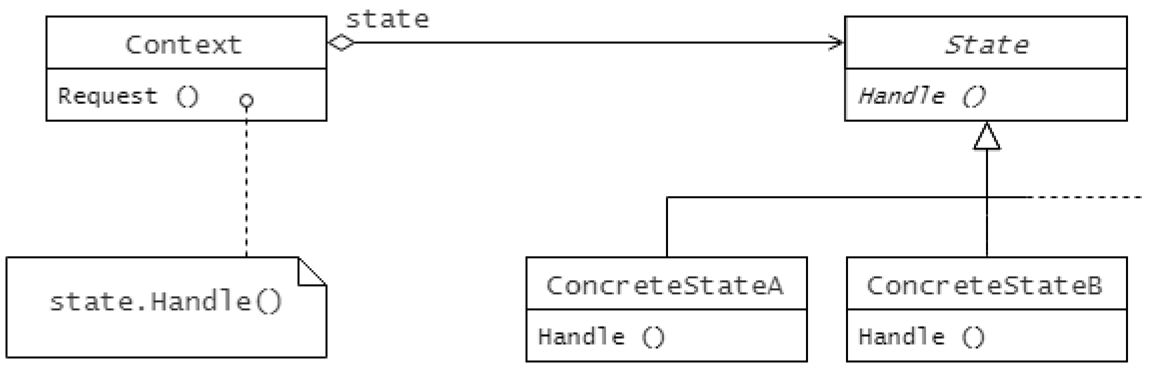

Figure 1:
The state design pattern as seen in
'Design Patterns: Elements of Reusable Object-Oriented Software'

Positive and Negative Consequences:

Because each state is implemented in a separate class, the state design pattern greatly decreases coupling. As a result, it is very easy to add states to support behaviour that the original implementation does not include. Furthermore, the state design pattern improves cohesion as behaviour associated with a single state is grouped together in one location. A disadvantage of the state design pattern is that it may result in many classes and a large amount of code compared to an implementation with a monolithic if/else statement.

Improvements to Non-functional Properties:

Readability / Maintainability

The state design pattern improves readability, and hence maintainability, by representing each state in a separate class.

Complexity

The state design pattern significantly decreases complexity of monolithic classes that rely on switch/case or if/else statements to change runtime state. Without the state design pattern, a program with many states involves a huge number of methods in a monolithic class and/or an if/else statement with a huge number of cases. In contrast, if such a program uses the state design pattern, no class has an excessive amount of code.

Scalability

The design pattern supports scalability. A system utilizing the state pattern will perform just as well with few states as it would with many; even if a greater number of states results in an increase in complexity, as long as there are no programming errors in implementing a large number of states, the system will have no difficulty handling a vast increase in the number of states that could be dealt with.

Heterogeneity

The separation of state-specific properties and behaviour into distinct classes also enhances heterogeneity. Each concrete state is a highly focussed representation of the behaviour related to that specific state.

Reliability

Because low coupling results in clear code that is easy to understand, the state design pattern makes it less likely for programmers to make mistakes and also enables them to quickly find and resolve bugs that do arise. Therefore, the state design pattern improves reliability.

Robustness / Fault-Tolerance / Survivability

A key aspect of the state design pattern is that the same action performed on different states results in different outcomes. That is, an action can be unexpected or a fault for one state though it is normal and acceptable for another. It is very difficult to concisely communicate using a monolithic if/else statement or multiple flags which actions lead to unexpected or erroneous results from which states. However, because the context tracks its current state, the state design pattern can easily transition to the correct state in these cases. For example, if an action occurs that the current state does not know how to handle, the context can transition into a safety state that pauses program execution and requests input from the user to decide the next step to take. Therefore, the state design pattern improves robustness. Similarly, if an error occurs, the context can transition to an error state and fail gracefully, ensuring fault-tolerance and survivability. The program can keep track of a certain number of previous states so it can provide a trace to the user upon a failure or other unanticipated event, allowing the user to easily debug the cause of the issue.

Safety
For safety-critical systems, the state design pattern makes it intuitive for developers to address all cases in which the program should halt and notify the user of the error that has occurred, since the code clearly outlines all the states and the actions that can occur within each state. Because the code is organized this way, another advantage for safety is that programmers are less likely to introduce bugs or fail to notice existing ones than if all the logic for state behaviour and transitions is defined in a monolithic if/else statement or by methods whose execution depends on numerous flags.

Hindrances to Non-functional Properties:
Maintainability
Because a separate class must exist for each state, a context with many possible states may require programmers to write much more code than if they implement the program using a monolithic class or if/else statement. Nonetheless, it is important to note that although the state design pattern may result in a large amount of code, this property also improves the code's clarity, which in turn improves maintainability.

Evolvability
The state design pattern hinders evolvability. For example, if new functionality is introduced to the state interface, developers must implement code for each concrete state class, whereas they only need to add the code to one class if they use a monolithic class or if/else statement instead. This disadvantage will vary in severity depending on the language that is used to implement the state pattern, as some less restrictive object oriented languages may allow the developer to create an empty method implementation in the parent class and have only those classes that require the new method to implement a functional version. In a monolithic if/else statement, however, such code only needs to be added in the blocks that represent the states that use the new functionality, no matter the language that is used for implementation.

Security
Because the state design pattern sometimes requires programmers to copy code among several classes, it also inhibits security.

Description of In-Class Example:

Consider a game where a player is constantly in battle with an enemy. In the battle, the player has only 2 actions:

1. Attack: Reduce enemy health by current state's stats.
2. Use Item: The player has a backpack where items can be placed to be activated. Only 2 items exist (Berserk item and Invincibility item - descriptions for items can be found below). Using items will overwrite any current player state (i.e. using items on top of each other will not combine the item abilities).

The player has 4 potential states:

1. Normal: Attacking reduces enemy health by 1 unit. Getting attacked reduces player health by 1 unit. This state will be the default at the beginning of every battle with an enemy.
2. Dead: No actions will be available in this state. This state is automatically activated when player health reaches 0.
3. Berserker: Attacking reduces enemy health by 3 units. Getting attacked reduces player health by 0.5 units. State activated by using Berserk item from backpack.
4. Invincible: Attacking does half the total health of the enemy in damage. Getting attacked does no damage. State activated by using Invincibility item from backpack.

In this example, the State Interface would simply consist of Attack(), GettingAttacked(), and UseItem(String itemName). The 4 states found above would be the ConcreteState classes.

1. The Normal state would return 1 unit for Attack() and -1 unit for GettingAttacked(). UseItem(String itemName) will switch the player state based on itemName and remove itemName from the backpack.
2. The Dead state would do nothing for each of these.
3. The Berserker state would return 3 units for Attack() and -0.5 units for GettingAttacked(). UseItem(String itemName) will switch the player state based on itemName and remove itemName from the backpack.
4. The Invincibility state would return 50% of enemy total health for Attack() and 0 units for GettingAttacked(). UseItem(String itemName) will switch the player state based on itemName and remove itemName from the backpack.

Positive Properties State Design Pattern Provides to Example:

A major benefit of the state design pattern is that it greatly reduces coupling by encapsulating the specific behaviour of each state in a separate class. Such a pattern is scalable because the program is able to run efficiently even when it has multiple states. In the example, we do not include the Berserker and Invincible states at the start of the game, but add these after we show that without such items, the player loses immediately. The example also shows that the state to which the context transitions depends not only on the action performed on the context, but also on the context’s current state. For instance, if the player’s current state is Normal and the player attacks the enemy, the enemy’s health decreases by 1 unit. However, if the player is in the Berserker state and attacks, the enemy’s health is reduced by 3 units.