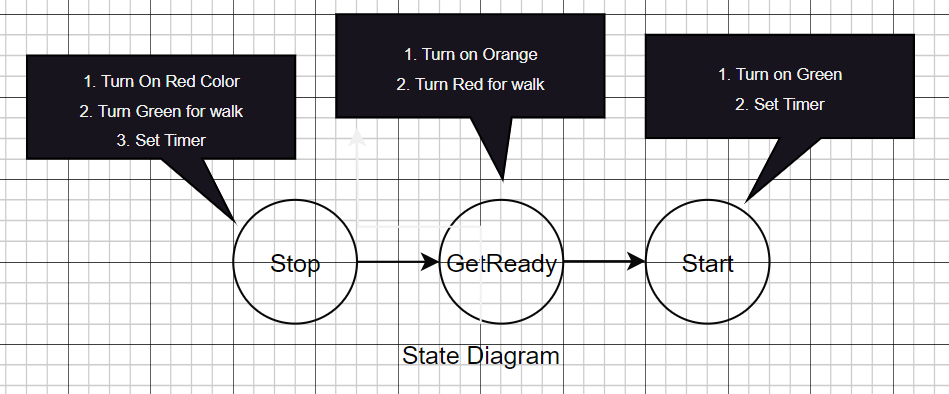
**State Design Pattern:** The State Design Pattern is a [behavioral design pattern](https://www.geeksforgeeks.org/behavioral-design-patterns/) that allows an object to change its behaviour when its internal state changes. This pattern is particularly useful when an object’s behaviour depends on its state, and the state can change during the object’s lifecycle.

This pattern focuses on managing state transitions and coordinating state-specific behaviour’s.

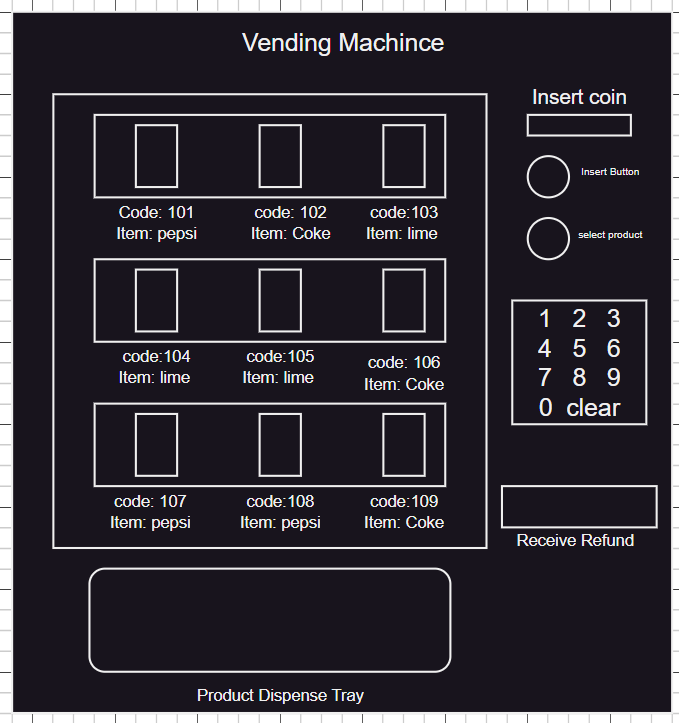
The State Design Pattern allows an object to alter its behaviour when its internal state changes, making it easier to manage state-specific behaviour.

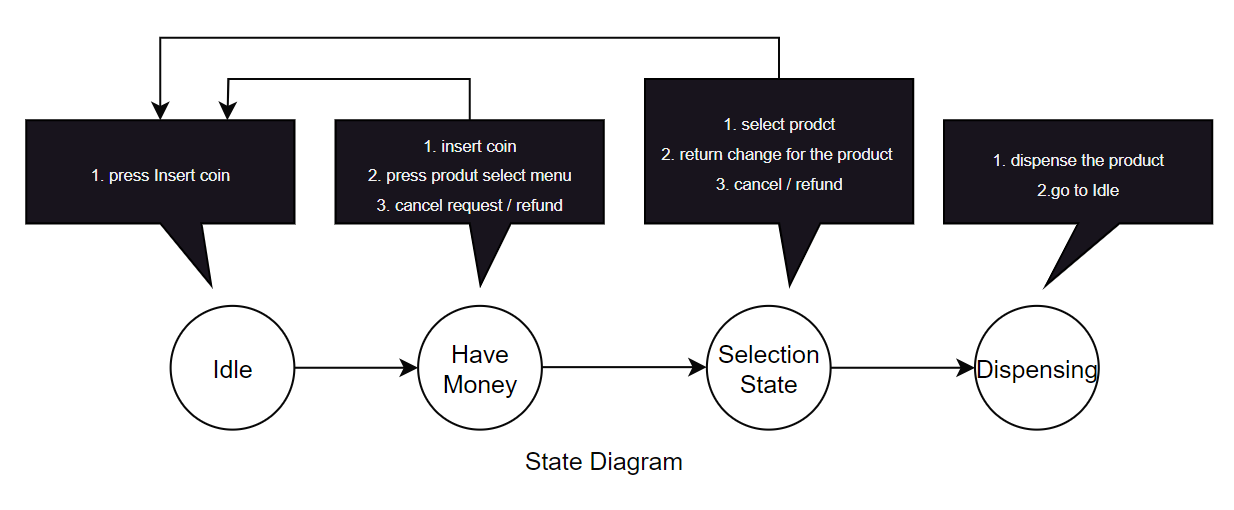
**Real world analogies for state Design pattern are:**

1. Traffic Signal



1. Vending Machine





**How State Design Pattern will help while building this system:**

* **Modelling Different States:**
  + The State design pattern allows us to model each state of the vending machine (e.g., ready, product selected, payment pending, out of stock) as a separate class.
  + This separation of concerns makes the codebase more organized and maintainable.
* **Encapsulation of State-specific Behavior:**
  + Each state class encapsulates its specific behavior. For example, the ReadyState class handles the behavior when the machine is ready for product selection, while the PaymentPendingState class handles behavior related to processing payments.
  + This encapsulation helps in managing complex state-dependent logic and promotes better code readability.
* **Dynamic State Transition:**
  + The State pattern facilitates dynamic state transitions. For instance, when a user selects a product, the vending machine transitions from the ReadyState to the ProductSelectedState, and further transitions occur based on the user’s actions.
  + This dynamic behavior allows the vending machine to adapt to different scenarios and handle state changes seamlessly.
* **Code Reusability:**
  + By implementing states as separate classes, the State pattern promotes code reusability. States can be reused across different contexts or vending machine implementations without modification.
  + This reusability reduces code duplication and promotes a more modular and scalable design.
* **Maintainability and Flexibility:**
  + As the vending machine requirements evolve or new states need to be added, the State pattern makes it easier to extend the system.
  + Modifications or additions to state-specific behavior can be made without affecting other parts of the codebase, leading to improved maintainability and flexibility.

**When to use the State Design Pattern**

The State design pattern is beneficial when you encounter situations with objects whose behavior changes dynamically based on their internal state. Here are some key indicators:

* **Multiple states with distinct behaviors:** If your object exists in several states (e.g., On/Off, Open/Closed, Started/Stopped), and each state dictates unique behaviors, the State pattern can encapsulate this logic effectively.
* **Complex conditional logic:** When conditional statements (if-else or switch-case) become extensive and complex within your object, the State pattern helps organize and separate state-specific behavior into individual classes, enhancing readability and maintainability.
* **Frequent state changes:** If your object transitions between states frequently, the State pattern provides a clear mechanism for managing these transitions and their associated actions.
* **Adding new states easily:** If you anticipate adding new states in the future, the State pattern facilitates this by allowing you to create new state classes without affecting existing ones.

**When not to use the State Design Pattern**

While the State pattern offers advantages, it’s not always the best solution. Here are some cases where it might be overkill:

* **Few states with simple behavior:** If your object has only a few simple states with minimal behavioral differences, the overhead of the State pattern outweighs its benefits. In such cases, simpler conditional logic within the object itself might suffice.
* **Performance-critical scenarios:** The pattern can introduce additional object creation and method calls, potentially impacting performance. If performance is paramount, a different approach might be more suitable.
* **Over-engineering simple problems:** Don’t apply the pattern just for the sake of using a design pattern. If your logic is clear and maintainable without it, stick with the simpler solution.

Ultimately, the decision to use the State pattern depends on the specific context and complexity of your problem. Consider the trade-offs between code organization, maintainability, performance, and development effort before applying it.

