# State Pattern

## Description of the design problem

1. For current design, most of controllers are really big, especially for the TransactionController. In this situation, it is really difficult to modification and maintenance. In another word if a programmer wants to modify TransactionController or something which is associated with it. Nearly, all usages of it need to be retested. It violates the Open for extension, Close for modification Principle.
2. The TransactionController has too many responsibilities. It need to control the transaction process, maintenance process. It violates the Single Responsibility Principle.
3. Code redundancy problem, in some situation, some codes are useless.

## Candidate design patterns considered

State Pattern

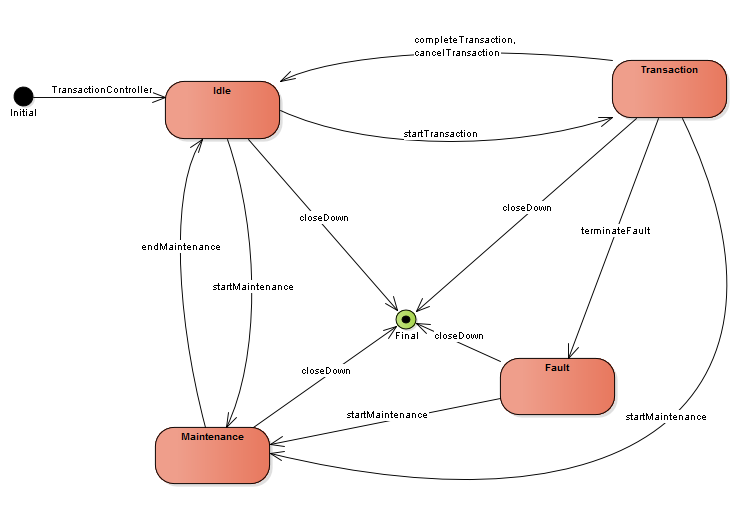
## Motivation to choose a pattern that would solve the problem including support for new requirement s or changes to existing problems

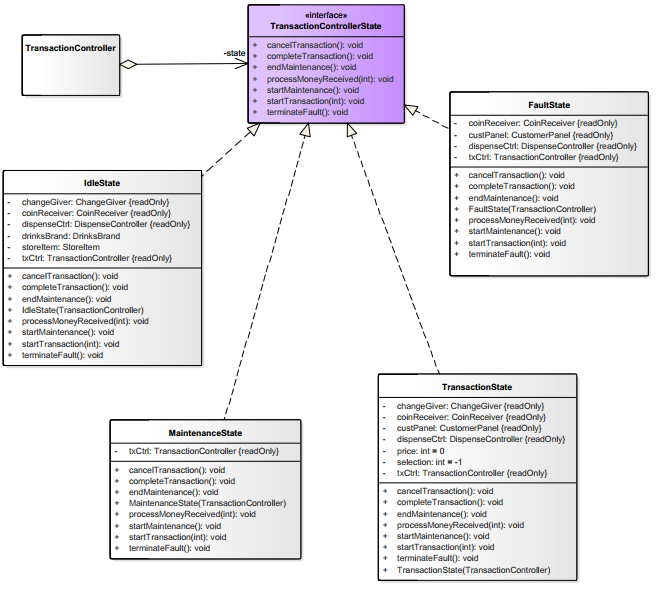
In this case State Pattern is suitable.

It localizes state-specific behavior and partitions behavior for different states. The State pattern puts all behavior associated with a particular state into one object. Because all state-specific code lives in a State subclass, new states and transitions can be added easily by defining new subclass. In this way, although the number of class and object is increased, it’s imposes structure on the code and makes its intent clearer.

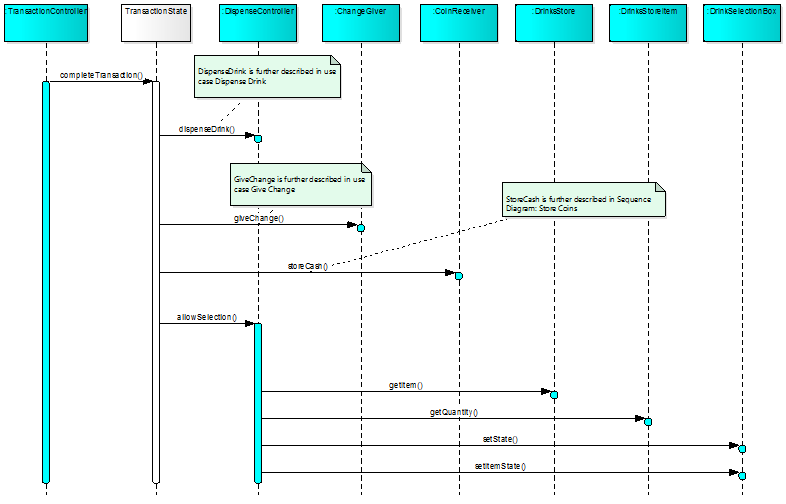
It makes state transitions explicit. Introducing separate object for different states makes the transitions more explicit. Also, State objects can protect the Context from inconsistent internal statesm because state transitions are atomic.

## Structure of the pattern (you should map the participants to your applications classes/objects)





## Collaborations among the participants (specific to your application objects)



## Implementation decision that you have taken

1. Who defines the state transitions?

In this case, using the State subclasses themselves specify their successor state is better, because in this way it’s more flexible and appropriate when to make the transition.

Decentralizing the transition logic in this way makes it easy to modify or extend the logic by defining new State subclasses, but it will cause the dependencies between subclasses.

1. A table based alternation.

Using a table can change the transition criteria by modifying data instead of changing codes, however:

* A table look-up is often less efficient than a virtual function call.
* Putting transition logic into a uniform, tabular format makes the transition criteria less explicit and therefore explicit and therefore harder to understand.
* It’s usually difficult to add actions to accompany the state transitions.

The virtual class is considered to implement the State Pattern because in this case it focuses on the state-specific behavior, not defining state transitions.

1. Creating and destroying State objects

Actually, there are two options: (1) to create State objects only when they are needed and destroy them thereafter versus (2) creating them ahead of time and never destroying them.

For option 1, it’s preferable when the states that will be entered aren’t know at run-time, and context s change state infrequently.

Option 2 is better when state changes occur rapidly. It can avoid destroying states which means there are no destruction costs at all.

In this case, Option 1 is accepted. Because for Option 2, the Context must keep references that might be entered

1. Using dynamic inheritances.

This is not possible in most object-oriented programming language. Exceptions include Self and other delegation-based programming languages. This mechanism lets objects change their behavior and amounts to changing their class.