State Pattern

Introduction

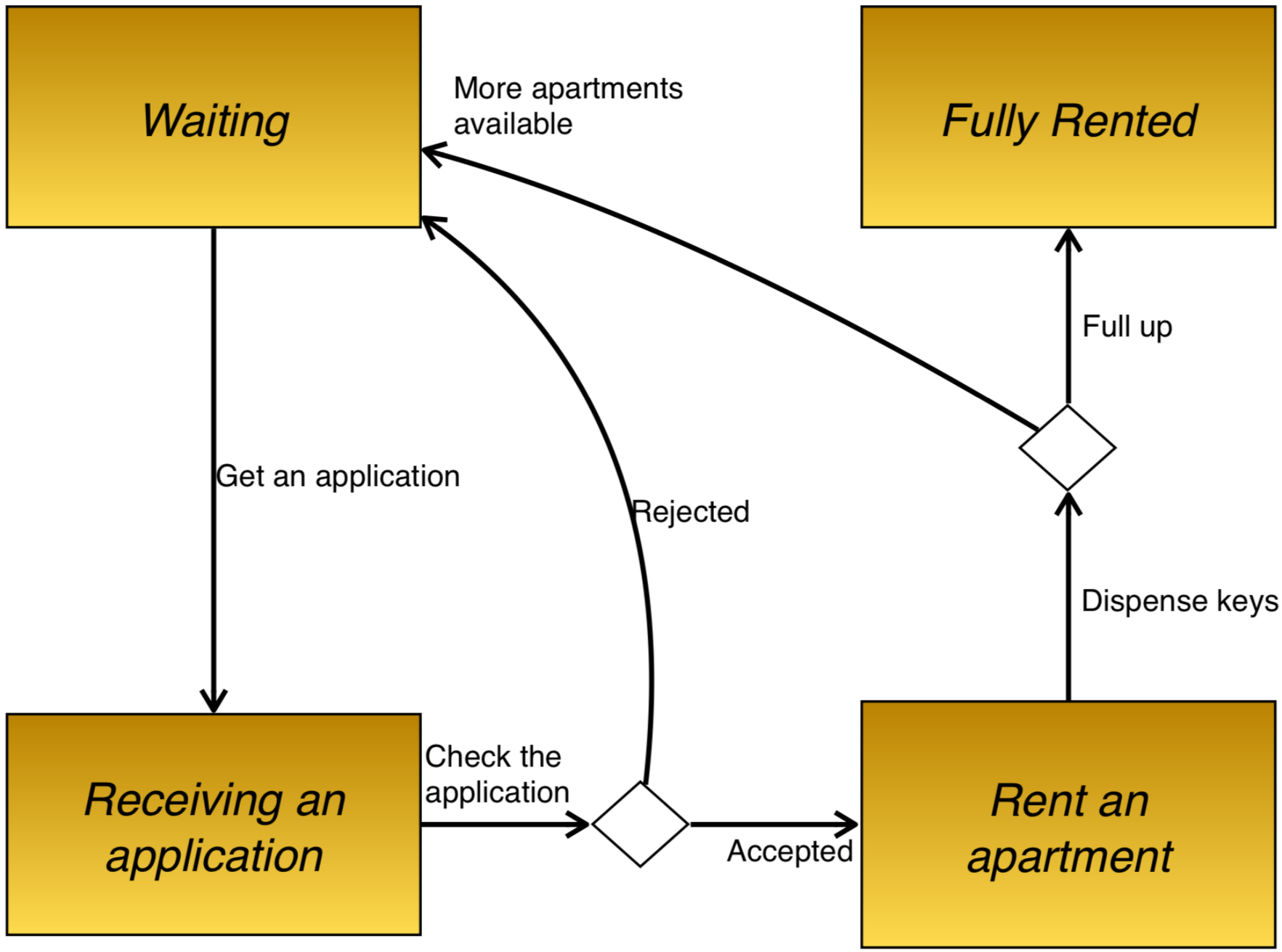
- We use the state pattern to resolve a classic problem that may be represented by State Machine. The purposes of this handout are as follows:
  - Use the state pattern.
  - Let the states decide the outcome.

Case Study - Apartment - Vending machine

- Company XYZ will convert all of its apartment complexes to use rental machines that will accept rental applications and distribute the keys. Thus, new tenants can submit applications to a machine and if they are approved, obtain their keys from this machine. Otherwise, the machine would have to tell the tenant, he was rejected and return to wait. And if the machine rents out an apartment, the machine should check if there are any apartments left in the complex and stop renting if there are none.

The rental machine has four states

- Waiting for a new tenant
- Receiving an application
- Renting an apartment
- Fully rented



The State Pattern

- A state machine is an abstract machine with two key elements: states and transitions.
    - A state refers to the current (active) status of a system.
    - A transition is a shifting of one state into another. The transition is triggered by an event or trigger condition. Generally, an action or set of actions takes place before or after a transition.

First Attempt - Using methods to hold state

- Let's write the application focused on a set of methods that do different things according to the present state. We use one constant for each possible state of the rental machine, plus an internal variable name called state. It contains the current state (which is defined at WAITING, when the application starts):

First Attempt - Using methods to hold state

```
class State(enum.Enum):
    FULLY_RENTED = 1
    WAITING = 2
    GOT_APPLICATION = 3
    APARTMENT_RENTED = 4

class RentalMethods:
    def __init__(self, n):
        self.numberApartments = n
        self.state = State.WAITING

    def getApplication(self):
        if self.state == State.FULLY_RENTED:
            print("Sorry, we're fully rented.")
        elif self.state == State.WAITING:
            self.state = State.GOT_APPLICATION
            print("Thanks for the application.")
        elif self.state == State.GOT_APPLICATION:
            print("We already got your application.")
        elif self.state == APARTMENT_RENTED:
            print("Hang on, we're renting you an apartment.")

    def checkApplication(self):
        if self.state == State.FULLY_RENTED:
            print("Sorry, we're fully rented.")
        elif self.state == State.WAITING:
            print("You have to submit an application.")
        elif self.state == State.GOT_APPLICATION:
            # simulate the chances for the application to be approved.
            approved = random.randint(0, 9) > 3

            if approved == True:
                print("Congratulations, you were approved.")
                self.state = State.APARTMENT_RENTED
                self.rentApartment()
            else:
                print("Sorry, you were not approved.")
                state = State.WAITING;
        elif self.state == State.APARTMENT_RENTED:
            print("Hang on, we're renting you an apartment")
```

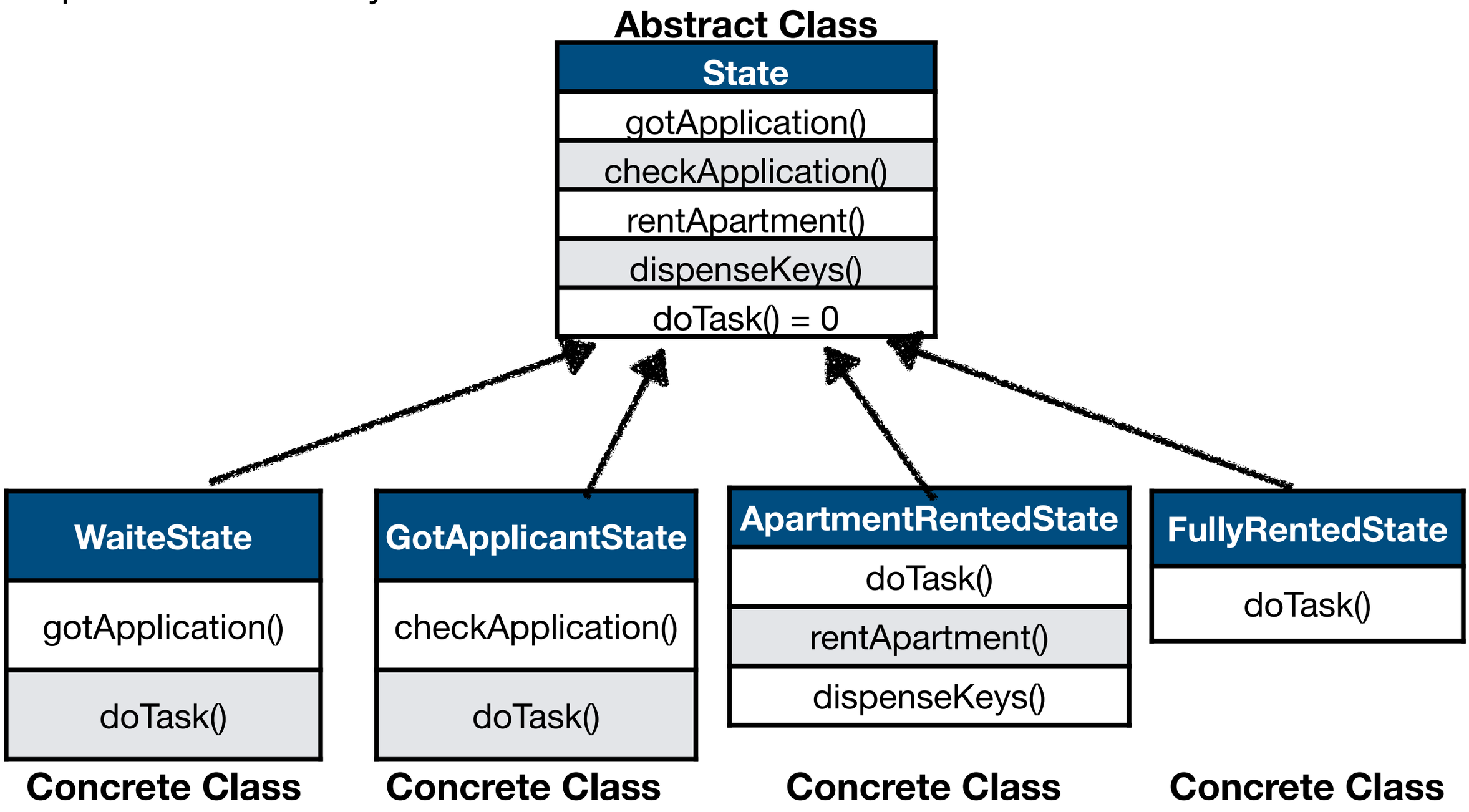
Using objects to encapsulate state

- There is a problem with the method based approach. As you add more states, each method becomes longer and longer and each method has to be rewritten for all the new states. What we should do is to encapsulate the states.

- Using objects to encapsulate state, we give each state it's own class. That way you can call methods like dispenseKeys or checkApplication on a state object anywhere in your code. All you have to do is load the correct state object into a reference variable of its abstract class and call various methods of that variable.

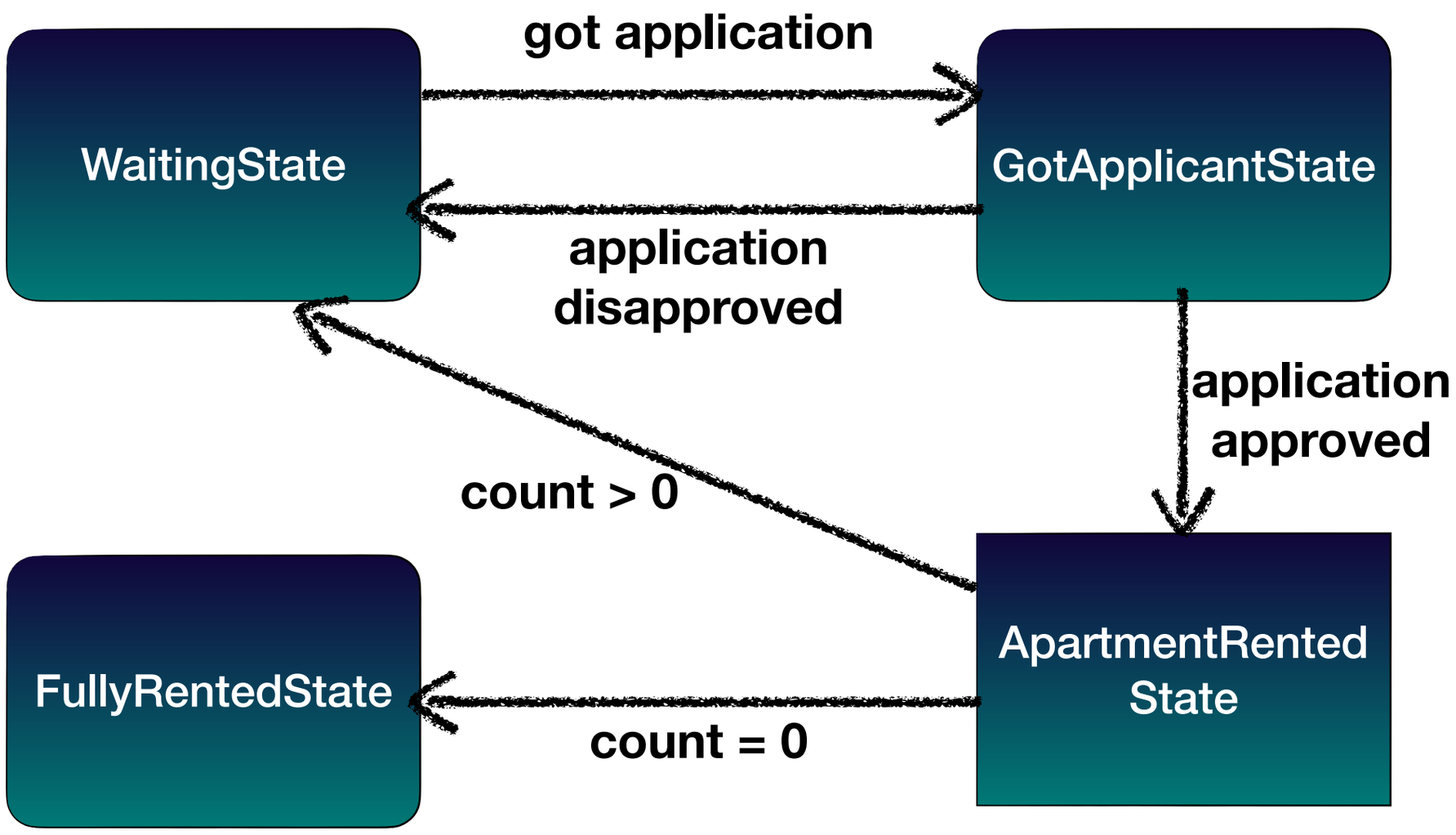
A Typical State Pattern Design

- The state design pattern is generally implemented using a parent State class that holds the common functionality of all states. And a number of concrete classes derived from State, where each derived class contains only the state-specific required functionality.



States and Transitions

- This state machine is implemented based on the following state machine diagram.



Rental Machine State Classes - Implementation

```
class State(ABC):
    def gotApplication(self):
        return self.doTask()

    def checkApplication(self):
        return self.doTask()

    def rentApartment(self):
        return self.doTask()

    def dispenseKeys(self):
        return self.doTask()

    @abstractmethod
    def doTask(self):
        pass
```

```
class WaitingState(State):
    def __init__(self, machine):
        self.machine = machine

    def gotApplication(self):
        self.machine.setState(self.machine.getGotApplicationState())
        return "Thanks for the application!"

    def doTask(self):
        return "You may not have applied or your application is pending."
```

```
class GotApplicantState(State):
    def __init__(self, machine):
        self.machine = machine

    def checkApplication(self):
        # simulate the chances for the application to be approved.
        approved = random.randint(0, 9) > 3

        if approved == True:
            self.machine.setState(self.machine.getApartmentRentedState())
            return "Congratulations, you were approved."
        else:
            self.machine.setState(self.machine.getWaitingState())
            return "Sorry, you were not approved."

    def doTask(self):
        return "You must have your application checked."
```

Rental Machine Class - Implementation

```
class RentingMachine(RentingMachineInterface):
    def __init__(self, n):
        self.count = n;
        self.waitingState = WaitingState(self)
        self.gotApplicationState = GotApplicantState(self)
        self.apartmentRentedState = ApartmentRentedState(self)
        self.fullyRentedState =FullyRentedState(self)
        self.state = self.waitingState;

    def gotApplication(self):
        print(self.state.gotApplication())

    def checkApplication(self):
        print(self.state.checkApplication())

    def rentApartment(self):
        print(self.state.rentApartment())
        print(self.state.dispenseKeys())

    def getWaitingState(self):
        return self.waitingState

    def getApartmentRentedState(self):
        return self.apartmentRentedState

    def getGotApplicationState(self):
        return self.gotApplicationState

    def getFullyRentedState(self):
        return self.fullyRentedState

    def getCount(self):
        return self.count;

    def setCount(self, n):
        self.count = n

    def setState(self, state):
        self.state = state
```

Other Examples - Vending Machine

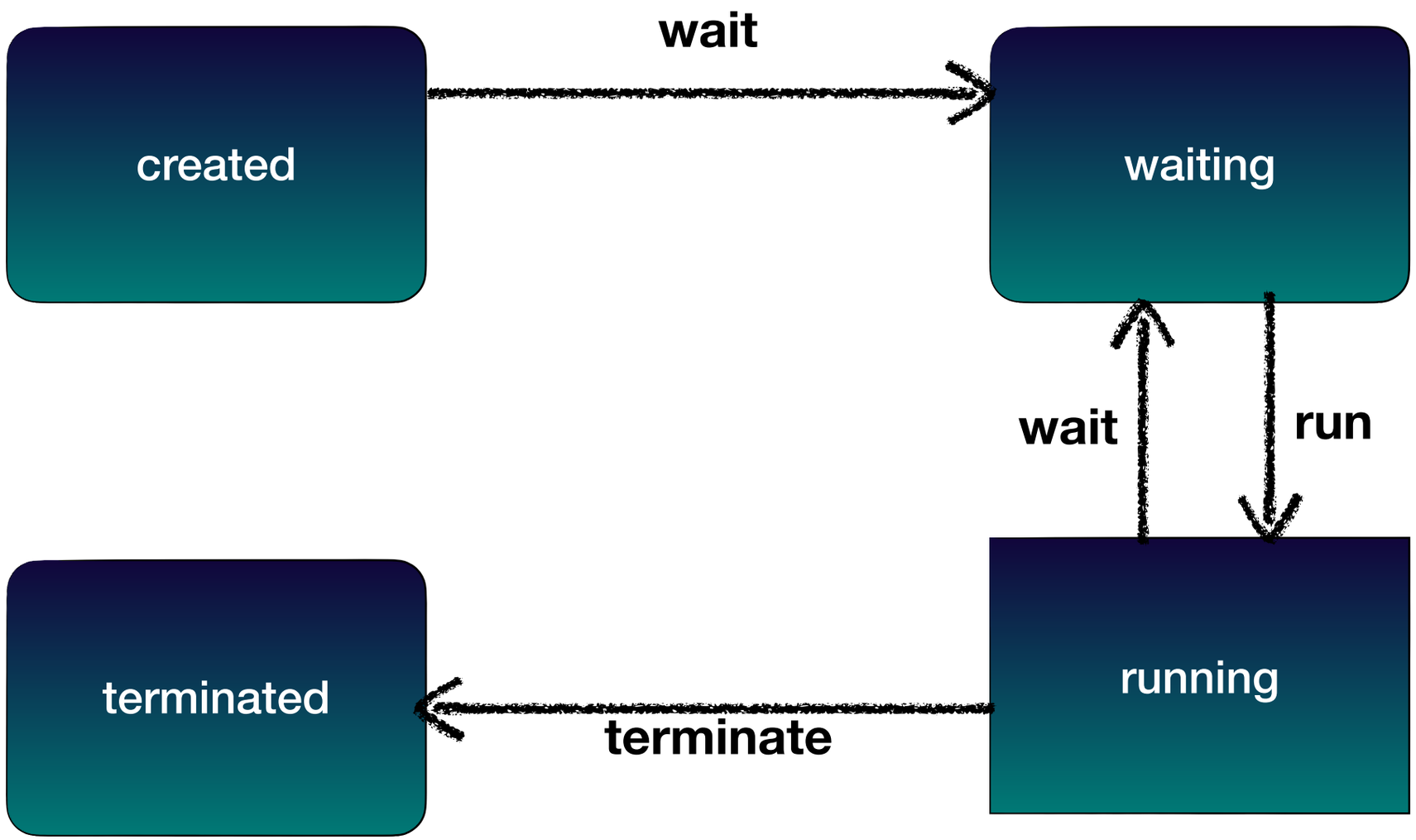
- An automatic vending machine is an example of the state pattern in daily life. The vending machine have different states and respond differently to the amount of money we put in. Depending on what we choose and the money we put in, the machine can do the following:
    - Refuse our selection because the product we asked for is not available.
    - Refuse our selection because the amount of money we put in was inadequate.
    - Deliver the product and give no change because we inserted the exact amount
    - Deliver the product and send back the change.

Using Python “state_machine” Module

- We can use existing Python modules which allow us not only to create state machines, but also to do it in the Pythonic way. A module that seems quite useful to me is state_machine. Before proceeding, if state_machine is not already installed in your system, you can install it using the **pip install state_machine** command.
- The state_machine module comes with the following decorators:
  - The @acts_as_state_machine decorator is used to decorate a class that is supposed to be a state machine.
  - Decorators @before and @after that can be used to perform actions before or after a transition happens, respectively.

States and Transitions

- This state machine is implemented based on the following state machine diagram.



Demonstration of the use of state_machine module

```
from state_machine import (State, Event, acts_as_state_machine,
after, before, InvalidStateTransition)

@acts_as_state_machine
class Process:
    # define 4 states
    created = State(initial=True)
    waiting = State()
    running = State()
    terminated = State()

    # define transitions
    wait = Event(from_states=(created, running), to_state=waiting)
    run = Event(from_states=waiting,to_state=running)
    terminate = Event(from_states=running,to_state=terminated)

    def __init__(self, name):
        self.name = name

    @after('wait')
    def wait_info(self):
        print(f'{self.name} entered waiting mode')

    @after('run')
    def run_info(self):
        print(f'{self.name} is running')

    @before('terminate')
    def terminate_info(self):
        print(f'{self.name} terminated')
```

```
def makeTransition(process, event, event_name):
    try:
        event()
    except  InvalidStateTransition as err:
        print(f'Error: transition of {process.name} from
{process.current_state} to {event_name} failed')

def state_info(process):
    print(f'state of {process.name}: {process.current_state}')

def main():
    RUNNING = 'running'
    WAITING = 'waiting'
    TERMINATED = 'terminated'

    p1 = Process('process1')
    p2 = Process('process2')
    [state_info(p) for p in (p1, p2)]

    print()
    makeTransition(p1, p1.wait, WAITING)
    makeTransition(p2, p2.terminate, TERMINATED)
    [state_info(p) for p in (p1, p2)]

    print()
    makeTransition(p1, p1.run, RUNNING)
    makeTransition(p2, p2.wait, WAITING)
    [state_info(p) for p in (p1, p2)]
```

Online Ordering State Machine

- This state machine is implemented based on the following state machine diagram.