State Diagrams

Contents

[1 State Diagram Basics 1](#_Toc118709692)

[2 Example – Guard Conditions 3](#_Toc118709693)

[3 Example – Actions and Activities 3](#_Toc118709694)

[4 Example – *Time-out* Trigger: 4](#_Toc118709695)

[5 Example – Nested Substates 4](#_Toc118709696)

[6 Implementing State with an Enum 5](#_Toc118709697)

[7 StarUML References 6](#_Toc118709698)

[8 State Design Pattern 7](#_Toc118709699)

[9 Exercises 8](#_Toc118709700)

[Appendix 1 Exercise Solutions 9](#_Toc118709701)

To make this document easier to read, it is recommended that you turn off spell checking and grammar checking in Word:

1. Choose: File, Option, Proofing
2. At the very bottom, check: “Hide spelling errors…” and “Hide grammar errors…”

# State Diagram Basics

A *state diagram* is a UML diagram that is used to describe the behavior of a system (or some part of a system, an object, *etc*) in response to external stimuli.

A *state* is the status of a system, the internal conditions at a particular time.

A *state diagram* models what *states* a system can be in at any point in time and how it *transitions* between states. Examples:

|  |  |
| --- | --- |
| Garage Door | Tic-Tac-Toe |
| E:\Data-Classes\CS 4321 - Fall 2017\topics\05_UML\state2.jpg | E:\Data-Classes\CS 4321 - Software Engineering 1\Topics\Ch08-UML 2\Fall12\state 1.jpg |

State diagram details:

* A state diagram is a directed graph where the nodes are states and the arcs are transitions.
* A state is represented by a rounded rectangle containing the name of the state. Special states:
* A solid black circle represents the *start state*
* A solid black circle with a ring around it represents an *end state*
* A *transition* represents a change of state in response to an (usually external) event. It is considered to occur instantaneously.
* The label on each transition is the event that causes the change of state.
* At any given point in time, the system or object is in a certain *state*. It remains in this state until an event occurs that forces the state to change. Being in a state means that the object will behave in a *specific way* in response to any events that occur. For example, if a garage door is in the *Closed* state and you press the “Open” button, the door opens. However, if a garage door is in the *Open* state and you press “Open”, nothing happens. Thus, some events will not cause the system to change state.

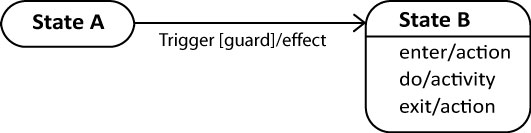
State diagrams are frequently used to model reactive (event-driven) systems:

* Video or pinball game
* Webserver
* Car transmission
* Keyless entry for car

But can also be used to model business processes and entities:

* Bank account
* Fedex order
* A document that needs review and approval at multiple levels (course substitution, purchase request, *etc.*)

Syntax for a state diagram:



* *Trigger* is the cause of the transition, which could be a signal, an event, a change in some condition, or the passage of time.
* *Guard* is a condition which must be true in order for the trigger to cause the transition.
* *Effect* is an action that occurs when a transition occurs.
* *Action –* something that takes place effectively *instantaneously* when a particular transition is taken and/or entry (exit) into (from) a particular state. Specifies a discrete amount of work that gets done.
* *Activity* – something that takes place while the system is *in* a state. It takes a period of time. The system may take a transition out of the state in response to completion of the activity. Some other outgoing transition may result in: the interruption of the activity, and/or an early exit from the state.

# Example – Guard Conditions

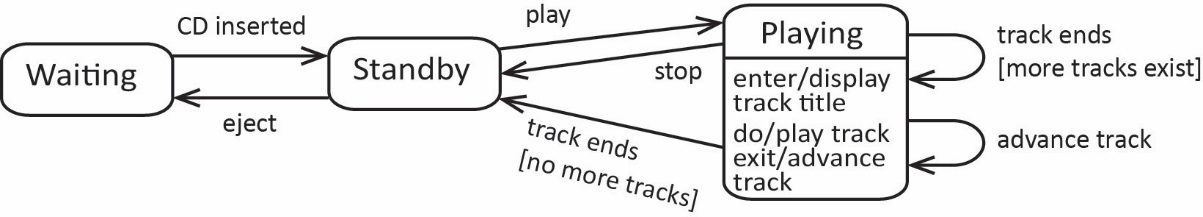
A *guard* is a logical statement that can be placed on a trigger and must be true for the trigger to cause a transition to another state.

|  |  |
| --- | --- |
| BankAccount object | OnlineOrder object |
| E:\Data-Classes\CS 4321 - Fall 2017\topics\05_UML\state1.jpg | E:\Data-Classes\CS 4321 - Fall 2017\topics\05_UML\s4.jpg |

# Example – Actions and Activities

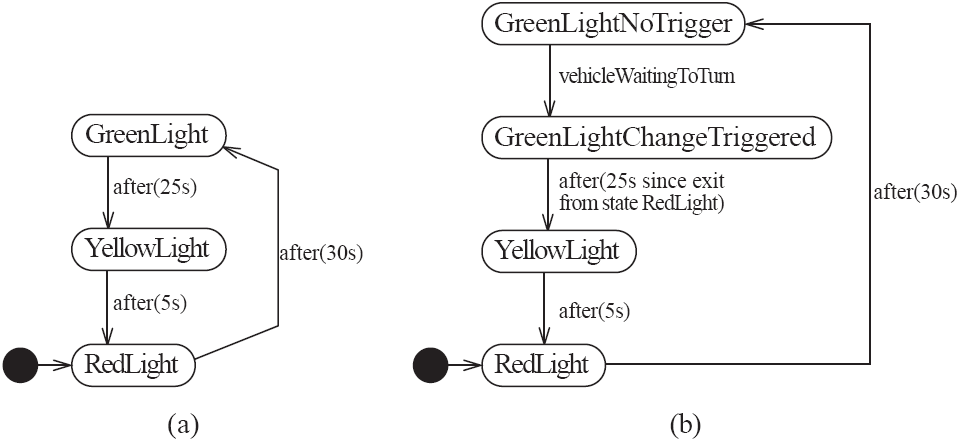
An *action* is a task that takes place when a transition is made into or out of a state and is assumed to take place instantaneously.

An *activity* is a task that takes place while the system is *in* a state. It takes a period of time. The system may take a transition out of the state in response to completion of the activity. Some other outgoing transition may result in: the interruption of the activity, and/or an early exit from the state.



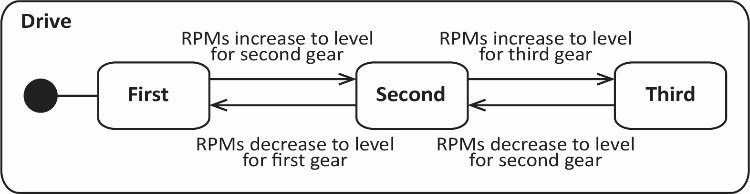
# Example – *Time-out* Trigger:

A *time-out* trigger is one that causes a transition to a new state after a specified period of time.

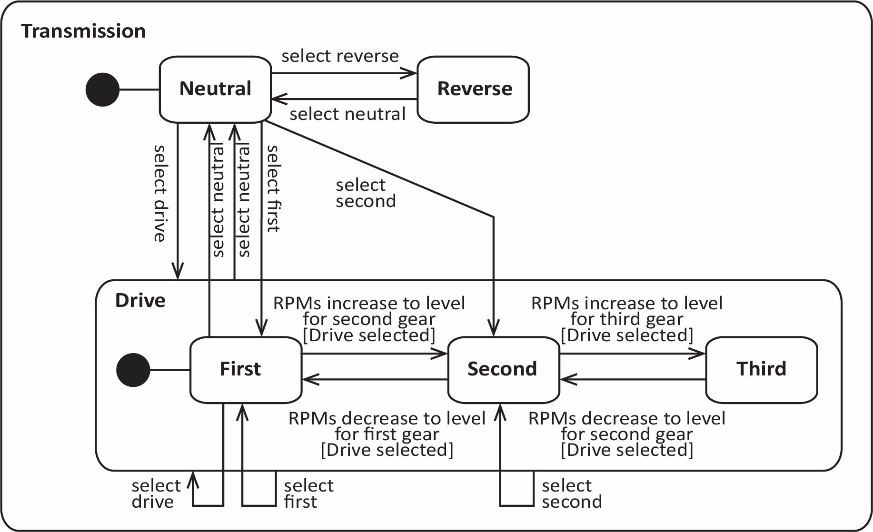


# Example – Nested Substates

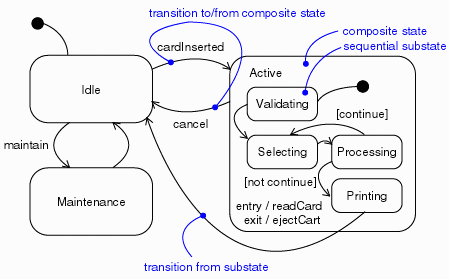
A car with an automatic transmission is placed in *drive* to drive the car. A state diagram for *drive* would represent the gears that the car can be in as states. For example:



When we view the transmission as a whole, we can treat *Drive* as a *nested substate*:



Example – State diagram for an ATM with nested substate, Active.



Source: <http://www.michael-richardson.com/processes/rup_for_sqa/core.base_rup/guidances/guidelines/statechart_diagram_640B5D0B.html>

# Implementing State with an Enum

1. Define an enum to define the states that a *UserAccount* can be in:

**public** **enum** UserStatus {

***PENDING***,

***ACTIVE***,

***SUSPENDED***,

***DELETED***

}

1. Use enum in *UserAccount* class:

**public** **class** UserAccount {

**private** String userId;

**private** String password;

**private** UserStatus status;

**private** **boolean** isLoggedIn;

**public** UserAccount(String userId, String password) {

**this**.userId = userId;

**this**.password = password;

**this**.status = UserStatus.***PENDING***;

**this**.isLoggedIn = **false**;

}

**public** UserStatus getStatus() {

**return** status;

}

**public** **boolean** setStatus(UserStatus newStatus) {

**if**(**this**.status==UserStatus.***DELETED***) {

**return** **false**;

}

**if**(newStatus == UserStatus.***SUSPENDED*** ||

newStatus == UserStatus.***DELETED***) {

isLoggedIn = **false**;

}

**this**.status = newStatus;

**return** **true**;

}

**public** **boolean** login(String userId, String password) {

**if**(**this**.userId.equals(userId) &&

**this**.password.equals(password) &&

**this**.status == UserStatus.***ACTIVE***) {

isLoggedIn = **true**;

**return** **true**;

}

**return** **false**;

}

@Override

**public** String toString() {

**return** "userId=" + userId + ", password=" + password +

", status=" + status + ", isLoggedIn=" + isLoggedIn;

}

}

1. Use the *UserAccount* class:

UserAccount user = **new** UserAccount("archie72", "Sar9b2e");

user.setStatus(UserStatus.***ACTIVE***);

user.login("archie72", "Sar9b2e");

user.setStatus(UserStatus.***SUSPENDED***);

**switch**(user.getStatus()) {

**case** ***ACTIVE***:

System.***out***.println("UserStatus is Active");

**break**;

**case** ***DELETED***:

System.***out***.println("UserStatus is Deleted");

**break**;

**case** ***PENDING***:

System.***out***.println("UserStatus is Pending");

**break**;

**case** ***SUSPENDED***:

System.***out***.println("UserStatus is Suspended");

**break**;

}

# StarUML References

<https://docs.staruml.io/working-with-uml-diagrams/statechart-diagram>

<https://staruml.readthedocs.io/en/latest/modeling-with-uml/working-with-statechart-diagram.html>

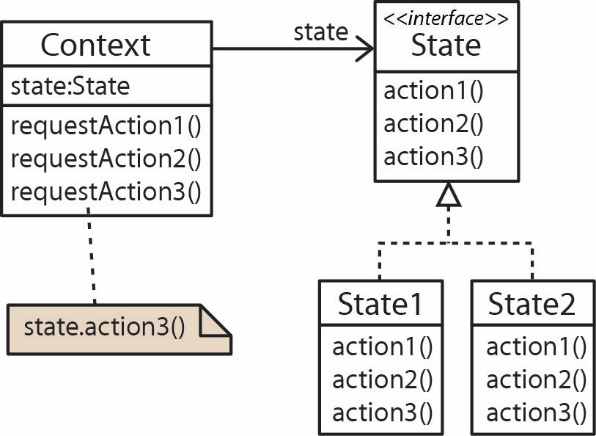
# State Design Pattern

Not required Fall 2022.

Suppose we have a *Context* that can have different *states* and is expected to respond to different requests. However, the actual response to a request depends on the state. A natural approach is:

|  |  |
| --- | --- |
| E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\10-State\pics\aa.jpg | What are the problems with this approach?   1. Not object-oriented 2. Not well encapsulated 3. Doesn’t follow the open-closed principle\* 4. Not flexible 5. Behavior is not intrinsic |

\* The *open-closed* principle is a design principle. It says a class should be *open* for extension, but *closed* for modification. Thus, we should seek to seek to design classes that allow its behavior to be extended without modifying its source code.

The *State* pattern is a design pattern and can be useful in implementing a system based on state. Steps to implement:

1. Identify the behaviors that depend on the state and make a *State* interface (or abstract class) that defines these behaviors.
2. For each state, define a class that implements the *State* interface. Code the behaviors assuming a particular state.
3. Associate the *Context* with the *State*.
4. Code the behaviors in the *Context* to delegate to the concrete state.

One important consideration is how does the state get changed? There are two approaches:

|  |  |  |
| --- | --- | --- |
| Independent States – The context can handle state changes. | Dependent States – State changes take place in the States themselves. | |
| E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\10-State\pics\aa3.jpg | E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\10-State\pics\aa4.jpg | |
|  |  | |
| * Advantage: States are not coupled to one another. | | * Advantage: Ease of adding States. |
| * Disadvantage: Harder a to add a State. | | * Disadvantage: Coupling between States. |

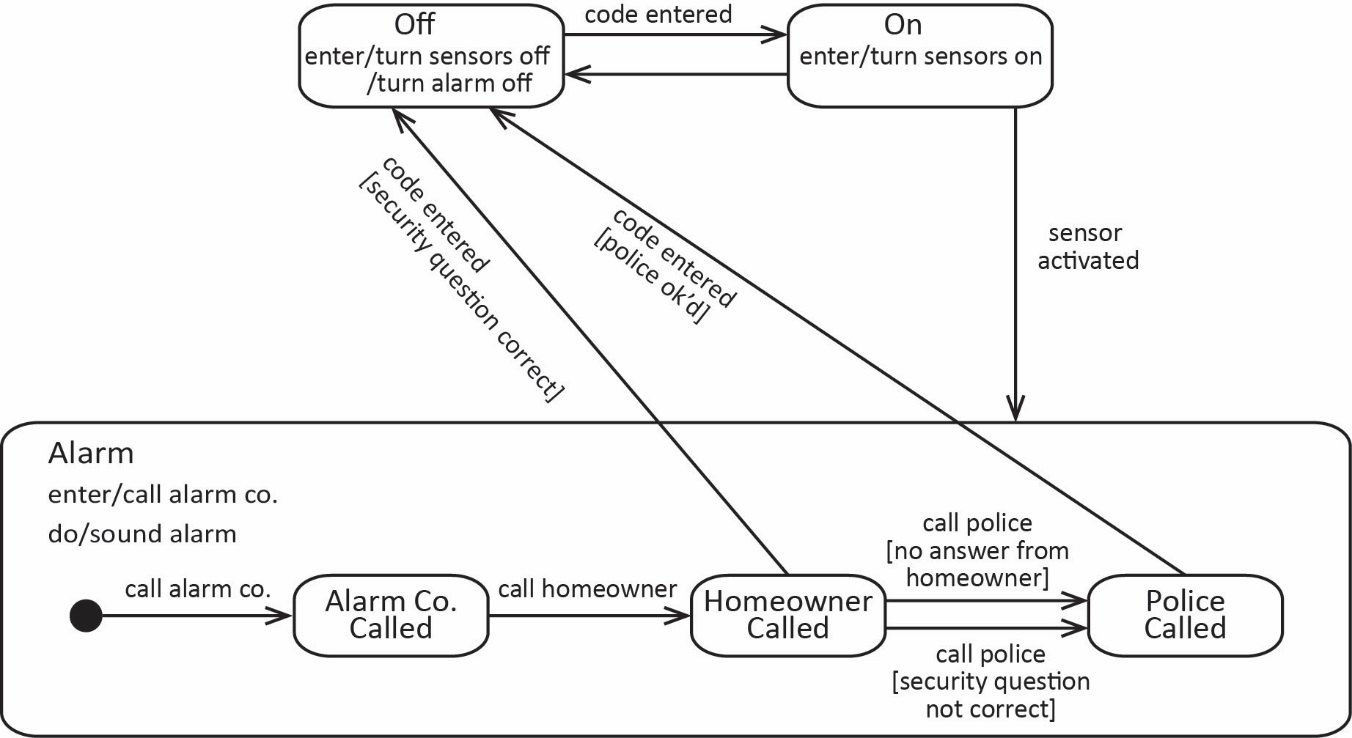
# Exercises

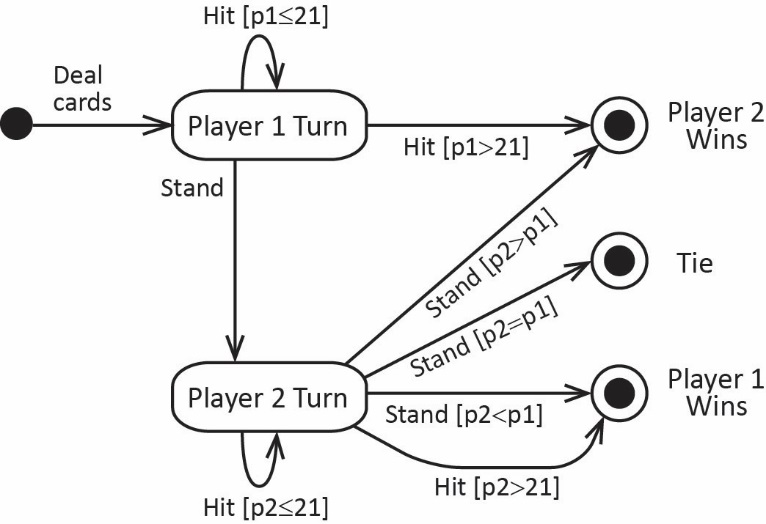
Not required Fall 2023

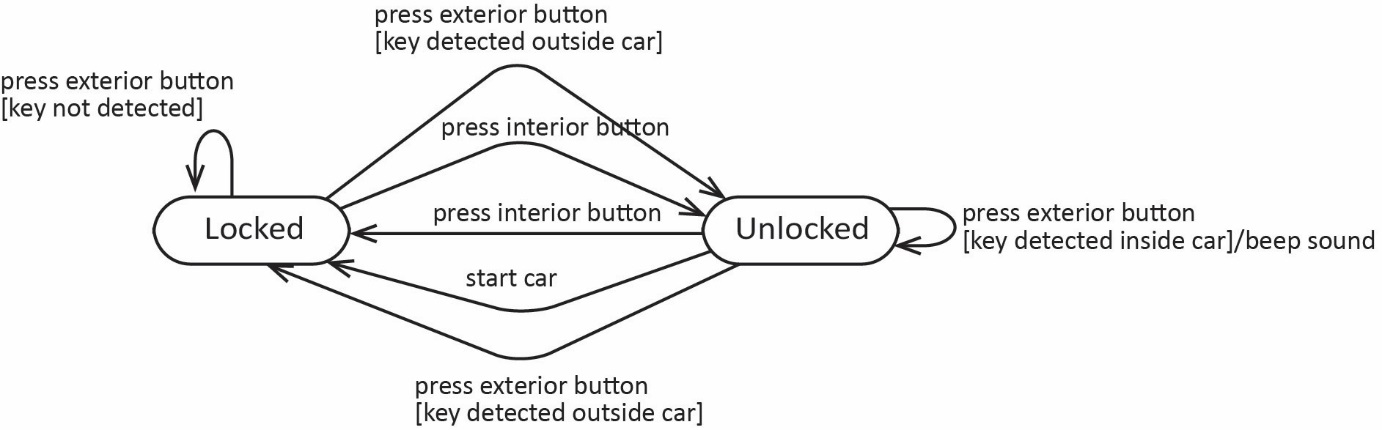
1. Briefly discuss what a State diagram is used for.
2. Develop a state diagram for a home alarm system that works in the following way. There is a console inside the house where you can turn on and off the alarm system by typing a code into a console. When the system is turned on the house is “armed” which means that any door or window that is opened will sound an alarm. If the alarm sounds, the monitoring company will automatically be called. The monitoring company will then phone the homeowner. If the homeowner can answer a security question then the monitoring company remotely turns off the alarm. If the call is not answered or the homeowner cannot answer the security question, the alarm continues to sound and the police are called. When the police OK things the monitoring company turns the alarm off remotely.
3. Draw a state diagram for the following (modified) card game Blackjack. In this version two players, *player1* and *player2* play against one another. The goal of the game is to obtain a card total higher than the other player, but not over 21. Initially, two cards are dealt to each player face down. *Player1* looks at her cards and decides to either *hit* or *stand.* A *hit* means that she is requesting another card in hopes of getting closer to 21. If the *hit* forces her hand over 21 then she automatically loses. *Player1* can continue to *hit* or she can *stand.* If she *stands* then it is *Player2’s* turn and he can do exactly as *Player1: hit, stand,* or *bust* (in which case *Player2* automatically loses). When *Player2* stands then both players lay their cards face up. Whoever has the highest hand wins, or if they are equal then the game ends with a tie.
4. Draw a state diagram for the following (simplified) description of an electronic car entry system. A car has a button on the exterior handle that is used to lock and unlock the car. Similarly, there is an interior button that locks and unlocks the car. If the car is locked and the owner presses the exterior button, the doors will unlock if the electronic key is detected outside the car. If the key is not detected, the door will not unlock. Once insdie the car and the car is started, the doors are automatically locked. At any time the owner is inside the car, she can press the interior button to lock (or unlock) the car. If the car is unlocked, and the owner presses the exterior button, the doors will be locked if the key is detected outside the car. Otherwise, if the key is detected inside the car, the car will remain unlocked and a beep sound will be emitted (in order to keep from locking the key in the car).
5. Suppose we have a *Customer* who can purchase *Product*s. Customers start with *Bronze* status and pay full price for products until they reach a cumulative purchase total of $5000. Then, they achieve *Silver* status and receive a 10% discount on purchases until they reach a cumulative total of $10,000. Then, they achieve *Gold* status and receive a 20% discount on purchases. Model this situation with a class diagram using the state design pattern. Annotate the class diagram with some code fragments that show how the methods are implemented. You can use independent or dependent states.

Appendix

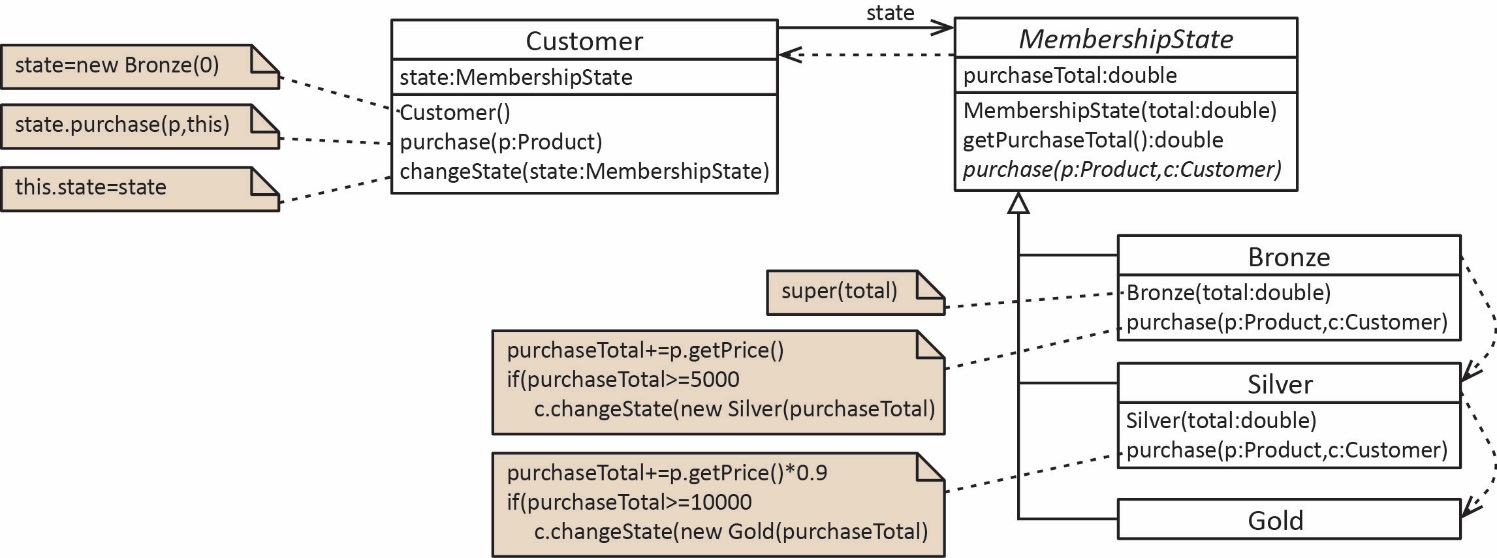
1. Exercise Solutions
2. See notes



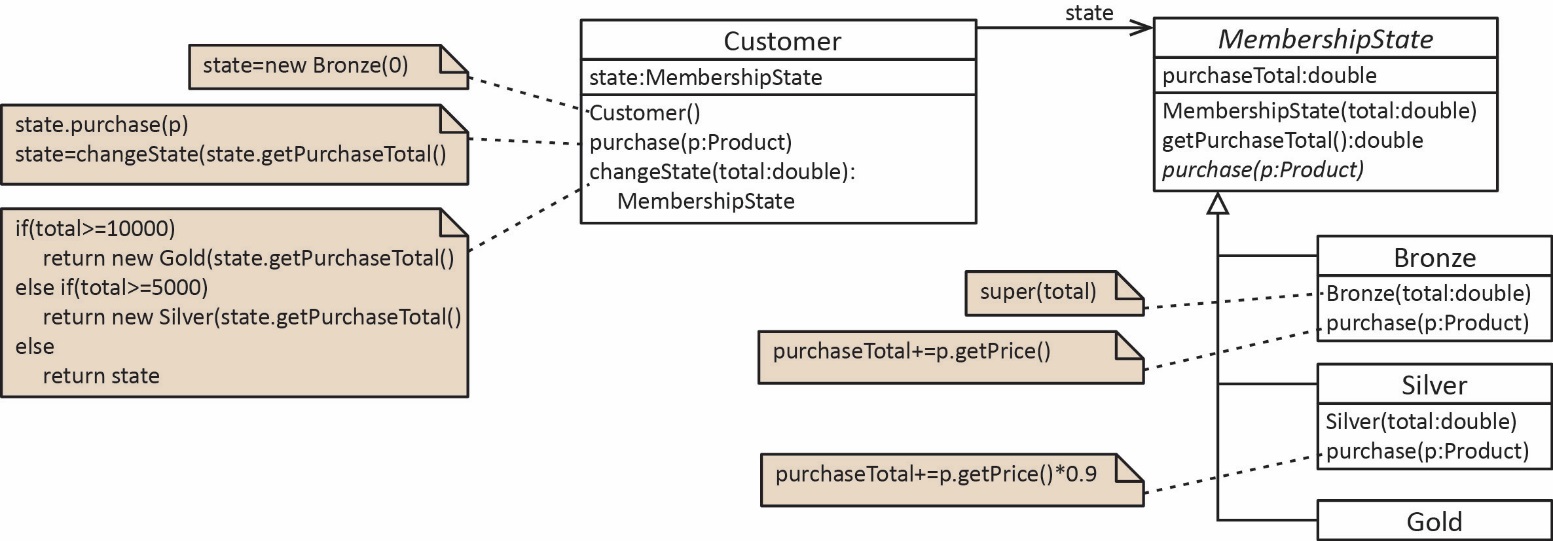




Dependent States – States handle transitions



Independent States – States handle transitions



|  |  |
| --- | --- |
| Driver | Output |
| **public** **static** **void** main(String[] args) {  Customer c = **new** Customer();  c.purchase(**new** Product(2000.0));  System.***out***.println(c);  c.purchase(**new** Product(4000.0));  System.***out***.println(c);  c.purchase(**new** Product(2000.0));  System.***out***.println(c);  c.purchase(**new** Product(3000.0));  System.***out***.println(c);  } | Purchase total=$2,000.00, state=Bronze  Purchase total=$6,000.00, state=Silver  Purchase total=$7,800.00, state=Silver  Purchase total=$10,500.00, state=Gold |