**Chaper 10 Notes – State Pattern**

**Motivating Example**

1. 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 |

1. The *State* pattern can be useful here. The State pattern is applicable when:
2. An object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
3. Operations have large, multipart conditional statements that depend on the object's state. The State pattern puts each branch of the conditional in a separate class.
4. The *State* pattern creates a separate *State* class for each state that encapsulates state specific data and behaviors. Thus, we take a situation as shown on the left and turn it into the design shown on the right.

|  |  |
| --- | --- |
| Before State Pattern | After State Pattern |
| E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\10-State\pics\aa.jpg | E:\Data-Classes\CS 4322 - Software Engineering 2\Notes\10-State\a5.jpg |

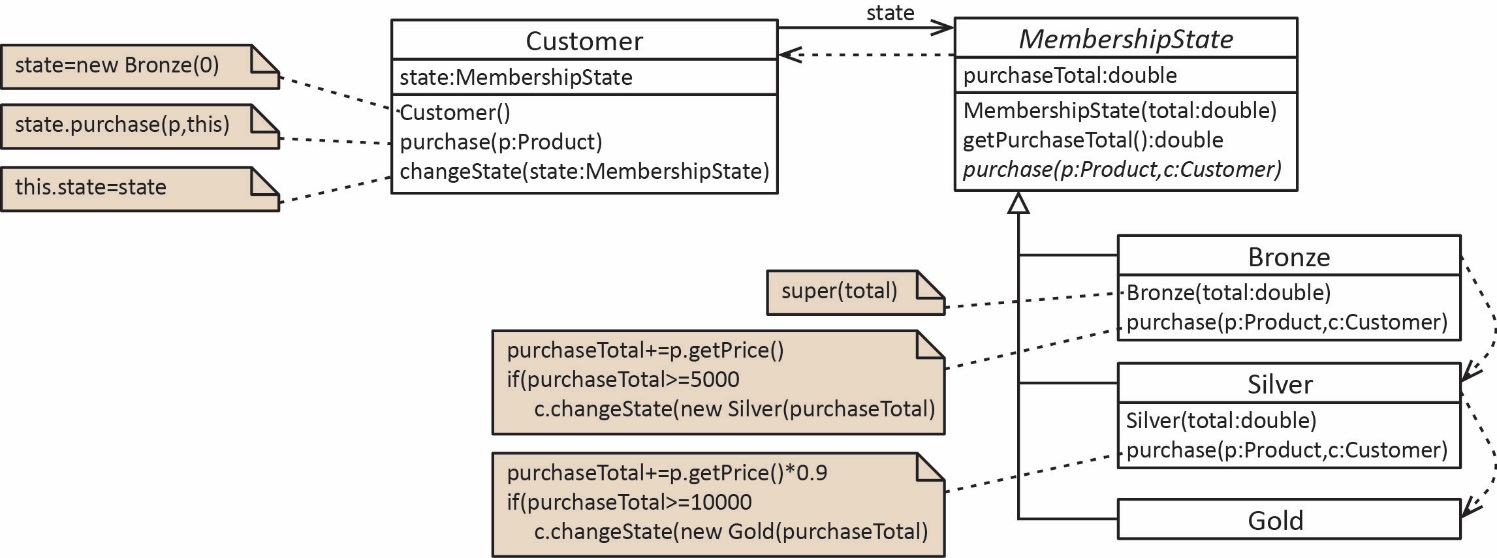
1. The State Pattern, “allows an object to alter its behavior when its internal state changes. The object will appear to change classes.” [GOF, p.305]
2. How does it *alter its behavior* when an internal state changes? An object is composed with an abstract state object. A change of state means that the object is composed with a different subclass which implements the behaviors differently. Thus, behaviors are altered.
3. How does the object *appear to change classes*? You ask the Context the same question at different times and get possible different answers. Thus, it is not the “same” Context it appears to have changed (classes).
4. One important consideration is to ask 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. |

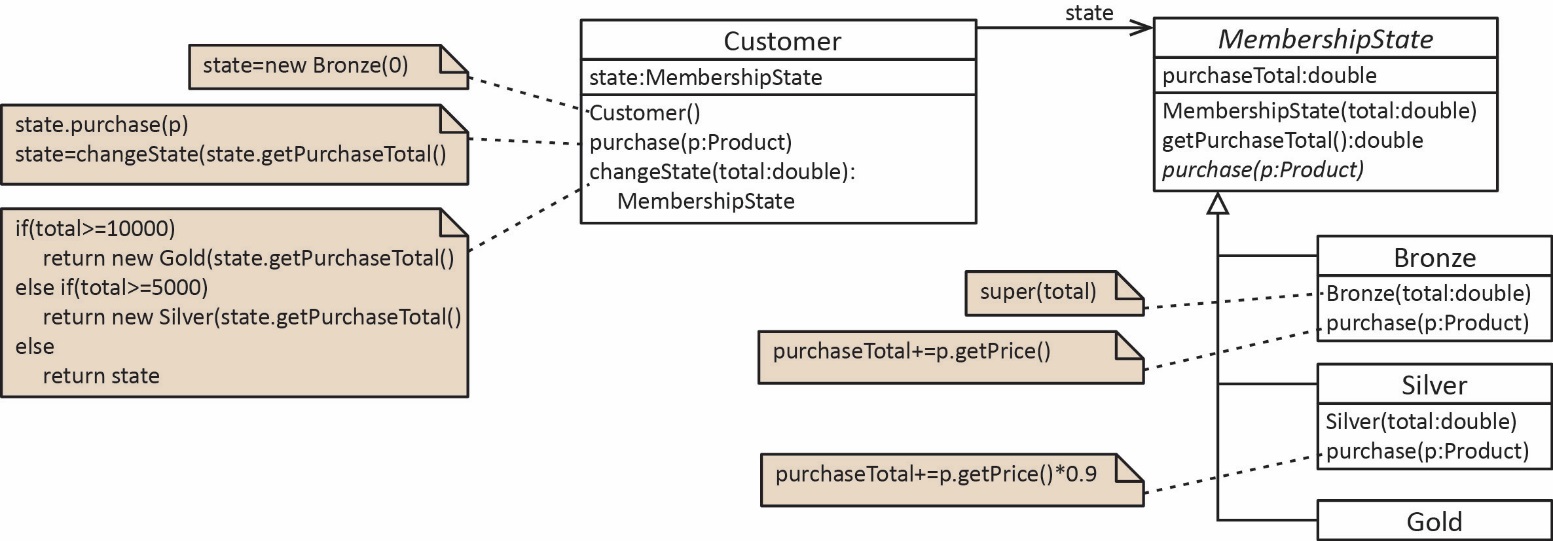
**Example**

1. 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.

Dependent States – States handle transitions

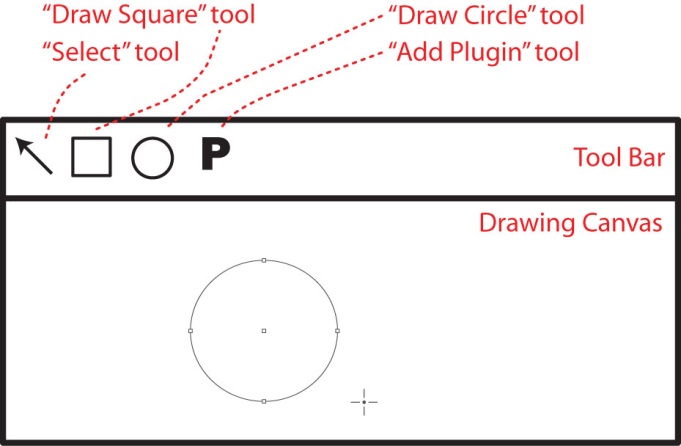


Independent States – States handle transitions

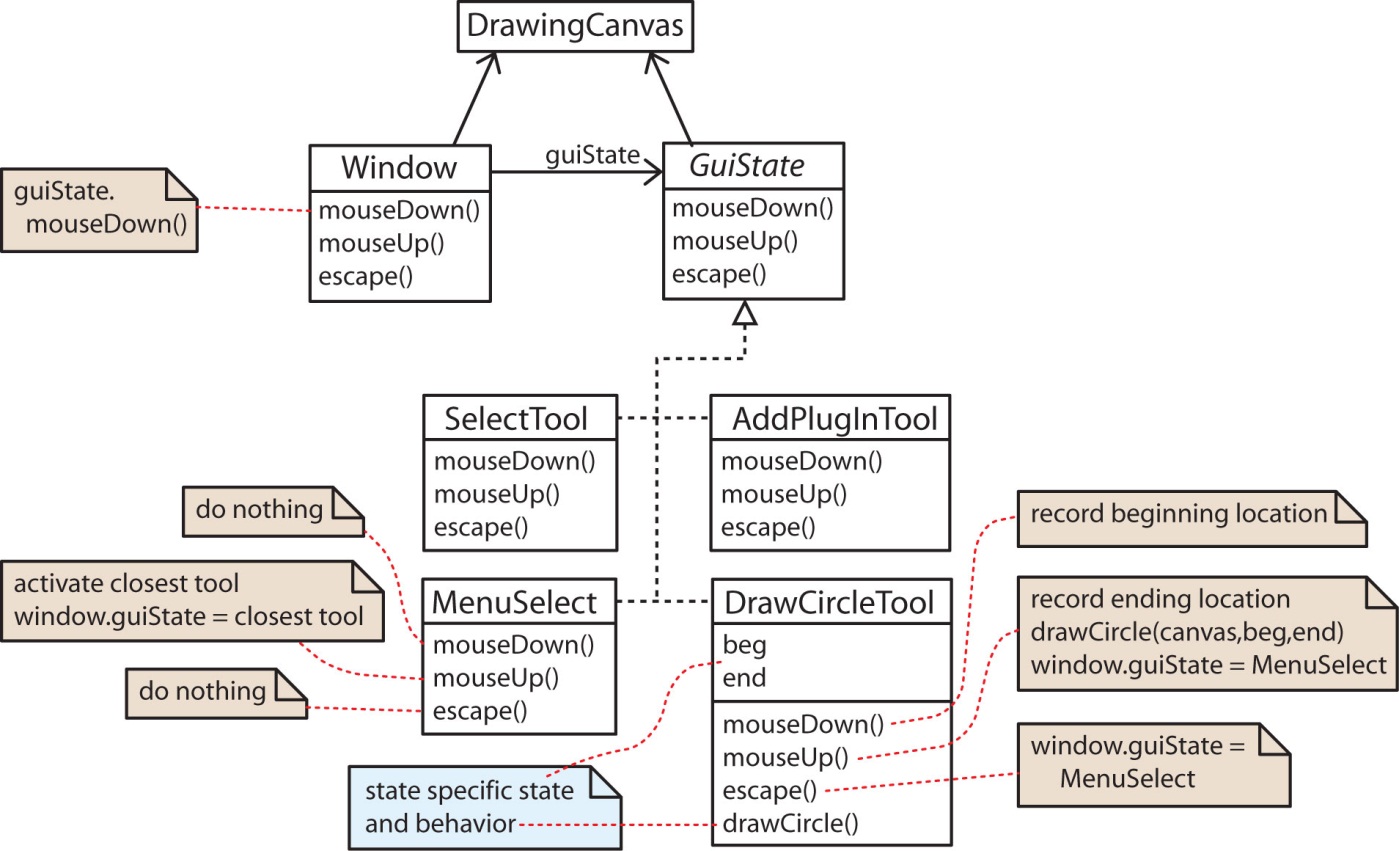


|  |  |
| --- | --- |
| Driver | Output |
| **public** **static** **void** main(String[] args) {  Customer c = **new** Customer();  c.purchase(**new** Product(2000.0));  System.***out***.println(c);  c.purchase(**new** Product(4000.0));  System.***out***.println(c);  c.purchase(**new** Product(2000.0));  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 |

1. Requirement: Develop a drawing program. The program has a menu bar with various tools that are used to manipulate a drawing. It also allows plugins to be developed and added to the Tool Bar.

****

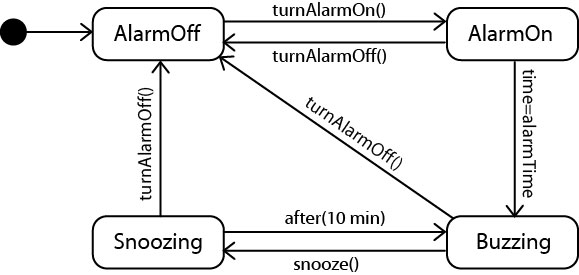
1. Applying the state pattern to the drawing problem.



**Examples**

1. Alarm Clock

**State Diagram**



**Class Diagram**

1. Suppose the AlarmClock is in the *BuzzingState* and snooze is pressed: 1, 2, and 3 occur
2. Next, the time is set: 4 and 5 occur (assume not time to activate buzzer)
3. Finally, the alarm is turned off (while snoozing): 6 and 7 occur

