CpE-684

Advanced Real Time Systems

Assignment - 2

Refined Design of AGV System   
and ATM Controller

By

|  |  |
| --- | --- |
| Nirmal Santhosh Avvaru | Rohitha Reddy Matta |
| Sai Sravanthi Lakkimsetty | Mounica Kommineni |

Table of Contents

[List of Figures 3](#_Toc435123084)

[**Abstract** 1](#_Toc435123085)

[**Part I** 2](#_Toc435123086)

[Before Refinement 2](#_Toc435123087)

[State Pattern 3](#_Toc435123088)

[Observer Pattern 5](#_Toc435123089)

[**PART II** 7](#_Toc435123090)

[Basic Statechart Pattern 7](#_Toc435123091)

[Hierarchial Statechart Pattern 7](#_Toc435123092)

[References 12](#_Toc435123093)

# List of Figures

[Figure 1 Unrefined class diagram of the AGV system 2](#_Toc435032352)

[Figure 2 Refined class diagram after applying the state pattern 3](#_Toc435032353)

[Figure 3 Refined Class diagram after applying observer pattern and state pattern 6](#_Toc435032354)

[Figure 4 Refined Design Using State Chart Pattern Language 9](#_Toc435032355)

# **Abstract**

This document presents a refined design for the AGV Control System and ATM controller. The first part of the document explains the refined initial class diagram for the AGV Control System using design patterns. A design pattern is a documented good design solution to a design problem. The refinement steps are described based on the patterns applied in a sequence. It mainly concentrates on enhancing the design quality (i.e., reduce coupling, increase cohesion, and reduce component complexity) using state pattern and observer pattern. A state pattern is a behavioral software design pattern used to capture the varying behavior for the same object based on its internal state, whereas, the observer pattern is used to implement distributed event handling system. The part II of the document outlines the development of a refined design of the ATM controller class of objects using the Hierarchial Statechart pattern language.

# **Part I**

## Before Refinement

The class diagram produced in Assignment-1 for an AGV Control System is given below (figure1). In the class diagram, the AGV system interacts with the Supervisory System, Motor, Arrival Sensor, Robot arm, Display system and Timer. Since it is difficult to obtain a quality design from the initial design, this class diagram is further refined using design patterns in the following sections. New components are defined and existing components are refined to enhance the quality of the design.

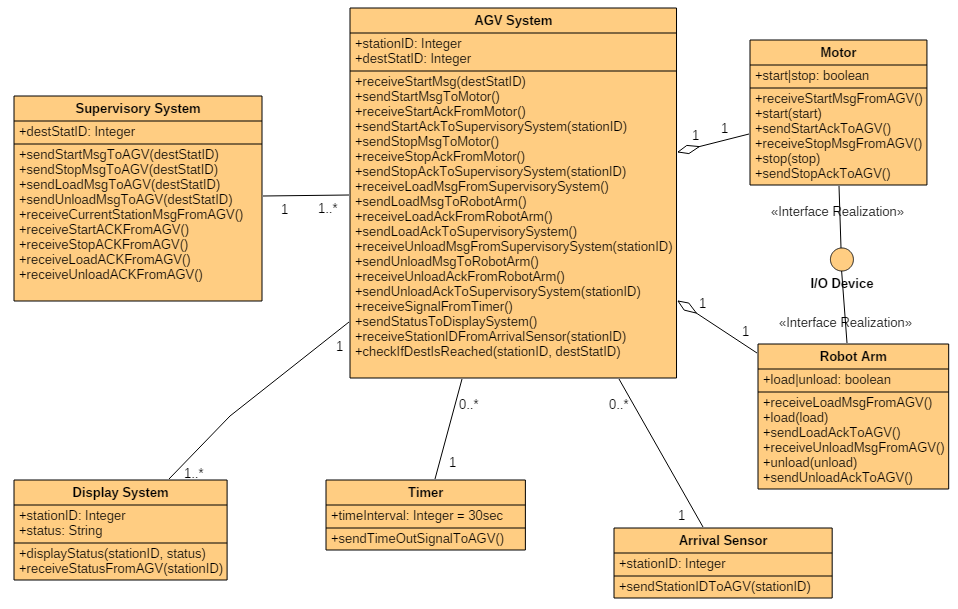


Figure 1 Unrefined class diagram of the AGV system

## State Pattern

The state pattern is a solution to the problem of how to make the behavior of an object depend on its state. When the internal state changes, state pattern allows an object to change it behavior. The state objects are active one at a time depending on the actual state of the context object. State pattern is a solution to the problem of how to make state and behavior depend on each other.

The design structure of a state pattern consists of a “context” class which provides a single interface to the outside world, a state abstract base class and various “states” of the state machine as derived classes of the state base class.

From the state diagram in the assignment in 1, we know that the AGV system can be in one of the below states at any point of time:

* Idle
* Start
* Stop
* Moving
* Load/Unload

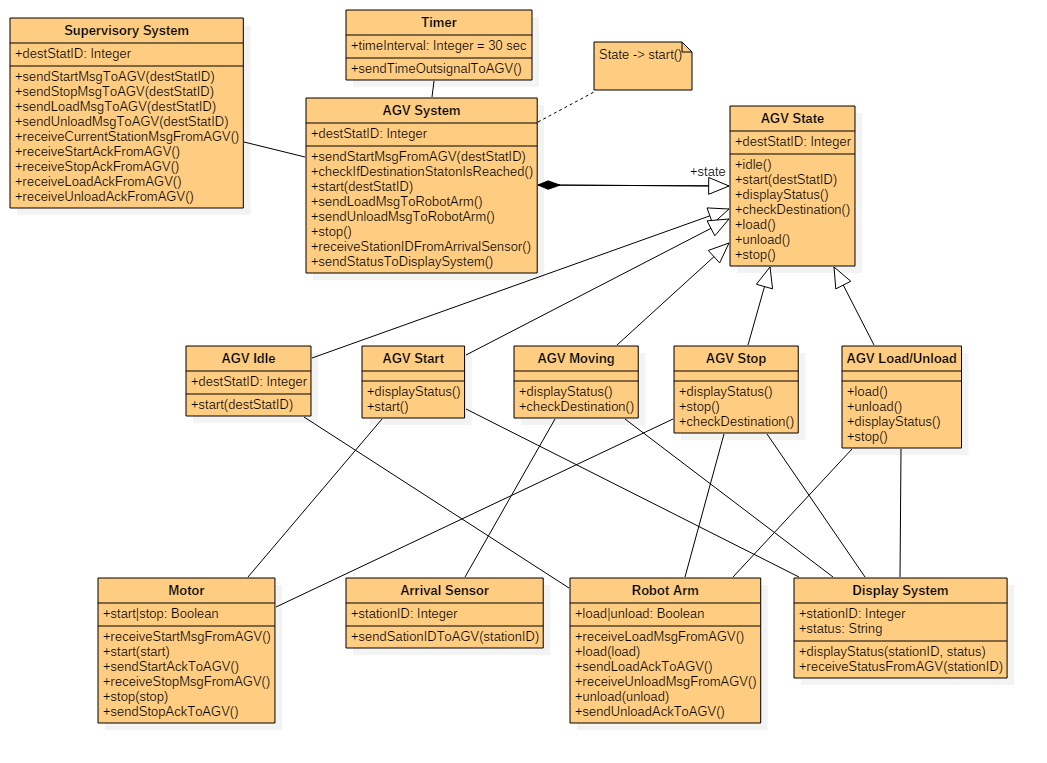


Figure 2 Refined class diagram after applying the state pattern

In the above diagram, the AGV System is the context class which provides a single interface to the other classes Supervisory System and Timer. AGV State is the state abstract base class which provides interface to its concrete state classes AGV Idle, AGV Start, AGV Moving, AGV Stop, and AGV Load/Unload.

It can also be noticed that a pointer is maintained to the current state, namely, ‘Start’ in the context class AGV. This current pointer has to be changed to another state to change the state of the AGV system.

## Observer Pattern

In observer pattern, multiple server objects are notified when change in state of subjects occur. The main intent of this design pattern is to define a one-to-many dependency between objects, so that, when one object changes its state, all its dependent are notified and updated automatically.

The design structure of an observer pattern consists of the following classes:

1. Subject – This is an abstract class defining the operations for attaching ad de-attaching the observer objects.
2. Observer – This is an abstract class which defines the operations used to notify the subject.
3. Concrete Subject – This is a class which maintains the object state and notifies all the attached observers whenever a change occurs.
4. Concrete Observer – This provides a reference to concrete subject objects to be notified of changes in the subject.

The object interaction happens as follows:

Whenever there is a change, Concrete Subject notifies its observers. After being notified, the Concrete Observer may query the Concrete Subject for information. Upon receiving this information, Concrete Observer then uses this to reconcile its state with that of the subject.

In the class diagram of the AGV system, all the objects behave independent of each other. In this section, we will apply an observer pattern to our refined class diagram, with the state pattern already applied, to further refine our class diagram, thereby, defining one-to-many dependencies between the objects. Having the dependencies defined, when an object changes its state, all its dependents are notified and updated accordingly.

In the below class diagram after applying the observer pattern and state pattern, we have the following classes as the concrete subjects:

* Supervisory System
* Arrival Sensor
* Timer

And the below classes as concrete observers:

* AGV System
* Display System
* Motor
* Robot Arm

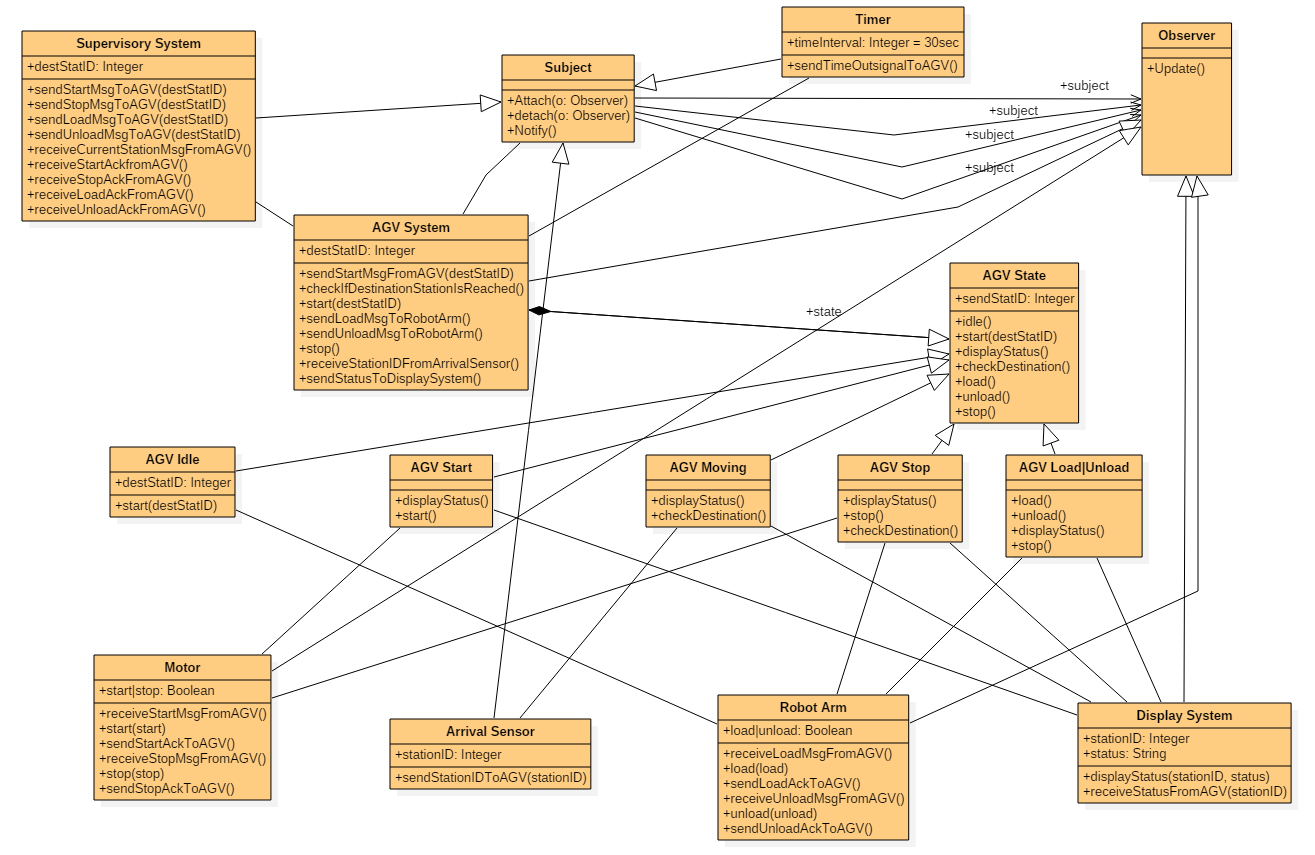


Figure 3 Refined Class diagram after applying observer pattern and state pattern

# **PART II**

## Basic Statechart Pattern

It is pattern language which uses an object oriented design and encapsulates the state of the entity into separate classes that correspond to the states defined in the specification. It distinguishes events, conditions, actions, entry and exit activities in each state class as methods and attributes of the state class.

## Hierarchial Statechart Pattern

It introduces superstates classes that are inherited from the abstract state class and uses the Composite pattern to allow the superstate to contain other states. Basic Statechart doesnot support hierarchy and thus the design is to be modified to allow enclosure of states inside superstates. To implement hierarchy in design, we distinguish different types of states as follows:

* SimpleState : a state that is not part of any superstate and doesn't contain any child state. (no parent and no children)
* Leaf State: a state that is child of a superstate but doesn't have any children (has a parent but has no children).
* Root SuperState: a state that encapsulates a group of other states (children) but has no parent.
* Intermediate SuperState: a state that encapsulates a group of other states (children) and has a parent state.

According to the Hierarchial StateChart given, we can deduce the following:

**Events**: This section contains the events to which the Entity responds to. Based on the entity’s current state and the type of event that is applied to it, the entity responds in a different way. The Base class implements these events that are common with respect to implementation for the other states of the entity. The Events in our Statechart pattern diagram are:

* InsertCard()
* AllowUserToUseATM()
* PINEntry()
* EnterChoiceOfTransaction()
* InvalidPINThrice()

**Actions**: These are static methods inherited by the state classes. The actions in the StateChart Pattern diagram are:

* CardValidation()
* UseATM()
* ValidatePIN()
* ProcessTransaction()
* ConfiscateCard()

**ATMState**: It is an abstract state class which groups both actions and events. When “set\_entity\_state” method is called, the entity’s state is changed. This has pointer to self “NextStates” which allows the state class to point to next possible states to be used in state transitions. It has an entity interface.

**ATMInterface**: It acts as an interface to the encapsulated logic in the pattern diagram. It has the current Entity’s state. It has an EventDispatcher method which receives events from the application environment and state implementation is called accordingly. When “UpdateState” is called by “set\_entity\_state” method, the entity’s state is updated.

**States**: The following are the different States in the State Chart Pattern

Simple States:

* Idle
* Closed Down

Leaf States:

* Waiting for PIN
* Validating PIN
* Waiting for Customer Choice
* Processing Transfer
* Processing Query
* Processing Withdrawal
* Confiscating
* Terminating
* Ejecting
* Printing
* Dispensing

Top Super States:

* Processing Customer Input
* Processing Transaction
* Terminating Transaction

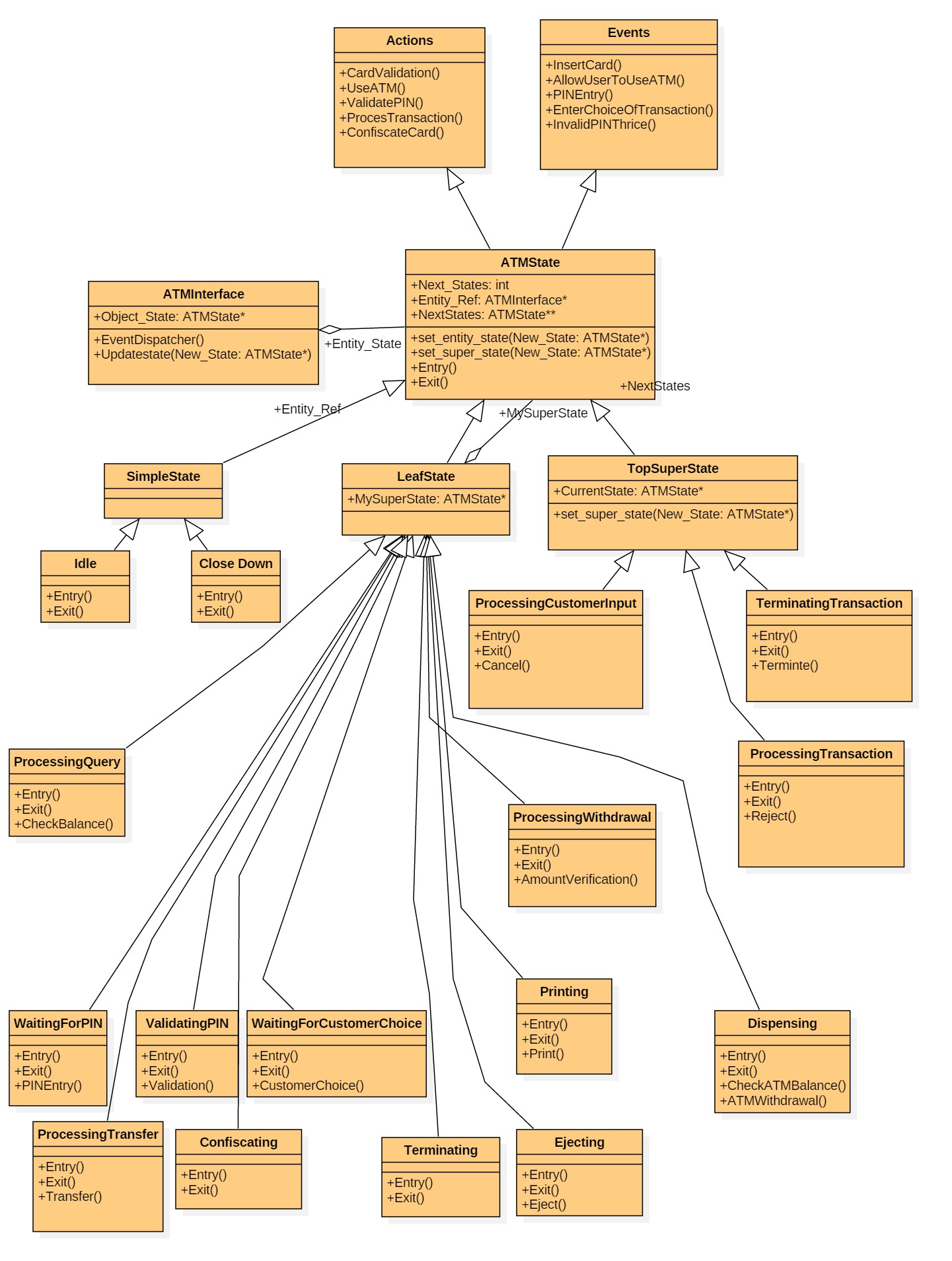


Figure 4 Refined Design Using State Chart Pattern Language

From the above state diagram we can explain the states as shown below.

1. Close down:

* Entry(): This is the default state for the ATM and displays welcome message to the users.
* Exit(): This state allows user to use the ATM.

1. Idle:

* Entry(): In this state, it displays message to the user to insert the ATM card.
* Exit() : In this state, the ATM card will get validated.

1. WaitingforPIN:

* Entry(): In this state, once user gets access to the ATM, it will display the message “Enter the PIN” to the user.
* PINEntry(): In this state, if user enters the PIN, the PIN value will be held by an attribute “PIN”.
* Exit(): In this state, PIN entered will be sent for the validation.

1. ValidatingPIN:

* Entry(): In this state, PIN held by attribute “Pin” will be processed.
* Validation(): In this case, validity of the PIN will be checked which leads to 3 scenarios.

1. If entered PIN is valid then it gives the user various chances to choose from ‘Transfer’, ‘Withdrawal’, ‘Balance Query’ etc.
2. If entered PIN is invalid it will ask the user to re-enter the PIN.
3. If user enters the invalid PIN thrice the card will get confiscated.

* Exit(): In this state, entered PIN will be cleared.

1. WaitingforCustomerChoice:

* Entry(): In this state, the ATM displays various options to users to select such as ‘Transfer’, ‘Withdrawal’, ‘Balance Query’.
* CustomerChoice(): In this state, user chooses one from all choices given.
* Exit(): In this state, action is performed based on the selected choice.

1. ProcessingTransfer:

* Entry(): In this state, user selects processing transfer option.
* Transfer(): In this state, user is asked to enter the account number and the amount to be transferred. The account number and the amount to be transferred will be held by attributes “AccountNumber” and “Amount” respectively. Also, balance is verified. If the current balance is greater than or equal to the amount requested, then transfer is done otherwise transfer is failed.
* Exit(): In this state, Account Balance will be updated and the details are sent for printing.

1. ProcessingQuery:

* Entry(): In this state, user selects ‘Balance Query’ option.
* CheckBalance(): In this method, the balance in the user account will be checked.
* Exit(): In this method, the balance details are sent for printing.

1. ProcessingWithdrawal:

* Entry(): In this method, user selects the “Withdrawal” option.
* AmountVerification(): In this method, user enters the amount to be withdrawn and the entered value will be held by an attribute “WIthdrawalAmount” and also the ATM checks the balance from the user account.
* Exit(): Withdrawal amount is valid.

1. Dispensing:

* Entry(): In this method, withdrawal process will be initiated.
* CheckATMBalance(): In this method, if there is sufficient balance in the account then AmountWithdrawal() method will be called, otherwise it gets closed down.
* AmountWithdrawal(): If there is sufficient balance, then withdrawal is processed and cash is dispensed.
* Exit(): The Account balance will be updated and details are sent for printing.

1. Printing:

* Entry(): In this method, details are processed for printing.
* Print(): Printing takes places.
* Exit(): The ATM card is ready to be ejected.

1. Ejecting:

* Entry(): In this method, the card is ready to be ejected.
* Eject(): Card is ejected.
* Exit(): The user takes the card.

1. Confiscated:

* Entry(): In this method, an invalid PIN is entered thrice.
* Exit(): In this method, card is confiscated.

1. Terminating:

* Entry(): In this method, all operations will be completed.
* Exit(): In this method, if closedown is not requested ATM moves to the idle state. Else, when closedown is requested, the ATM moves to the state where a new user can use it.

1. ProcessingCustomerInput:

* Cancel(): In this state, if user cancels the operation then the ATM card gets ejected.

1. ProcessingTransaction:

* Reject() : If transaction gets rejected for some reason, card gets rejected.

1. TerminatingTransaction:

* Terminate() : In this state, once the ATM serves one user then it gets ready to serve another.

# References

1. E. Gamma, R. Helm, R. Johnson and J. Vissides, “Design Patterns- Elements of Reusable Object-Oriented Software”, Addison-Wesly, 1995
2. J. Vlissides, "An Introduction to Design Patterns," 1994-1999.
3. J. Vlissides, "Desigining with Patterns," 1996-1999.