UML Class Diagrams :

Below are UML diagrams describing some of the major functionalities of the game.

There are several relationships exhibited by the different member classes that make up the components of the game. Some of these are :

1. Association : Shows a relationship between the two classes. One of the classes may have objects of another class being used within it. This is shown by a bold arrow line.

2. Dependency : In some cases it can also show a dependency between two or more classes. Any changes made to a class may cause changes in the class that is dependent on it. This is shown by a dotted arrow line.

3. Composition: This depicts the relationship between two classes where one class "is entirely made of another class" ie: One of the classes cannot exist if the parent/main class object doesn't exist. This is represented by an arrow with a darkened diamond at the end of the parent class.

4. Aggregation : This resembles the "part of" relationship between 2 classes. ie : One class is "a part" of another class. Here both the classes

5. Generalization : This is used one one class generalises the functionalities of all its subclasses. That is it is an umbrella class for other classes which inherit all properties from the parent as well have some other specifi properties unique to them.

6. Specialisation : This is the exact opposite of generalisation. A specialised class is a subclass which inherits all properties from its parent class and also adds certain specific properties that are unique to it. All specific classes will be under a particular main/parent class.

FUNCTIONALITY : CREATING A NEW PARTY WITH NEW PATRONS

- The classes involved in this are :

    - ControlDesk
    - ControlDeskObserver (Interface)
    - ControlDeskEvent
    - ControlDeskView
    - Bowler
    - Party
    - NewPatronView
    - AddPartyView

- As we can observe from the diagram, the `ControlDeskEvent` class is the class that manages the interclass function calls and data that is to be passed from one class to another.

- The cardinalities - *number of participating objects in any association between two classes* - are also mentioned on the relationship arrows of the classes.

- To further indicate the creation of classes, it has also been specified as to which class is create from which parent class. The `create` written above the arrows indicates the parent-child relationship.

- The `ControlDeskObserver` class is an interface class. Essentially serves as an interface between two or more classes that may not be able to interact otherwise. For eg: the `ControlDeskView` and `ControlDeskEvent` classes are interfaced to be able to share infromation and perform functions. This is also a demonstration of the *Adapter Design Pattern*

- All the methods and attributes associated with each class are shown in the UML class diagram as well.

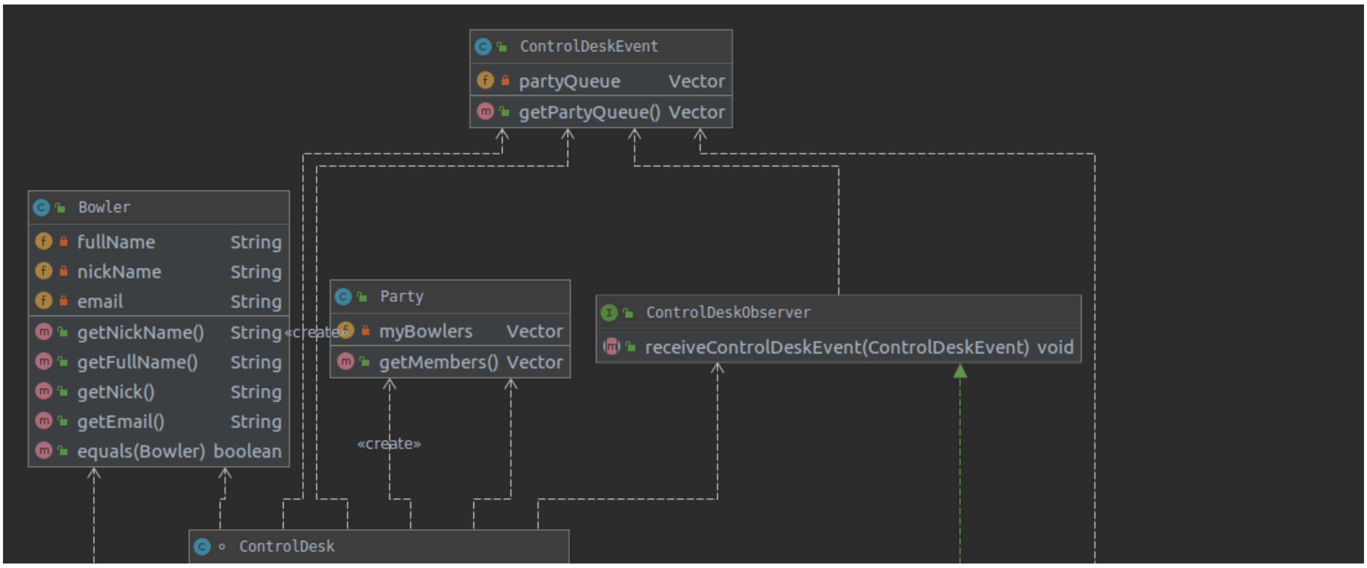
The several arrows in the diagram represent different relationships between the multiple classes.

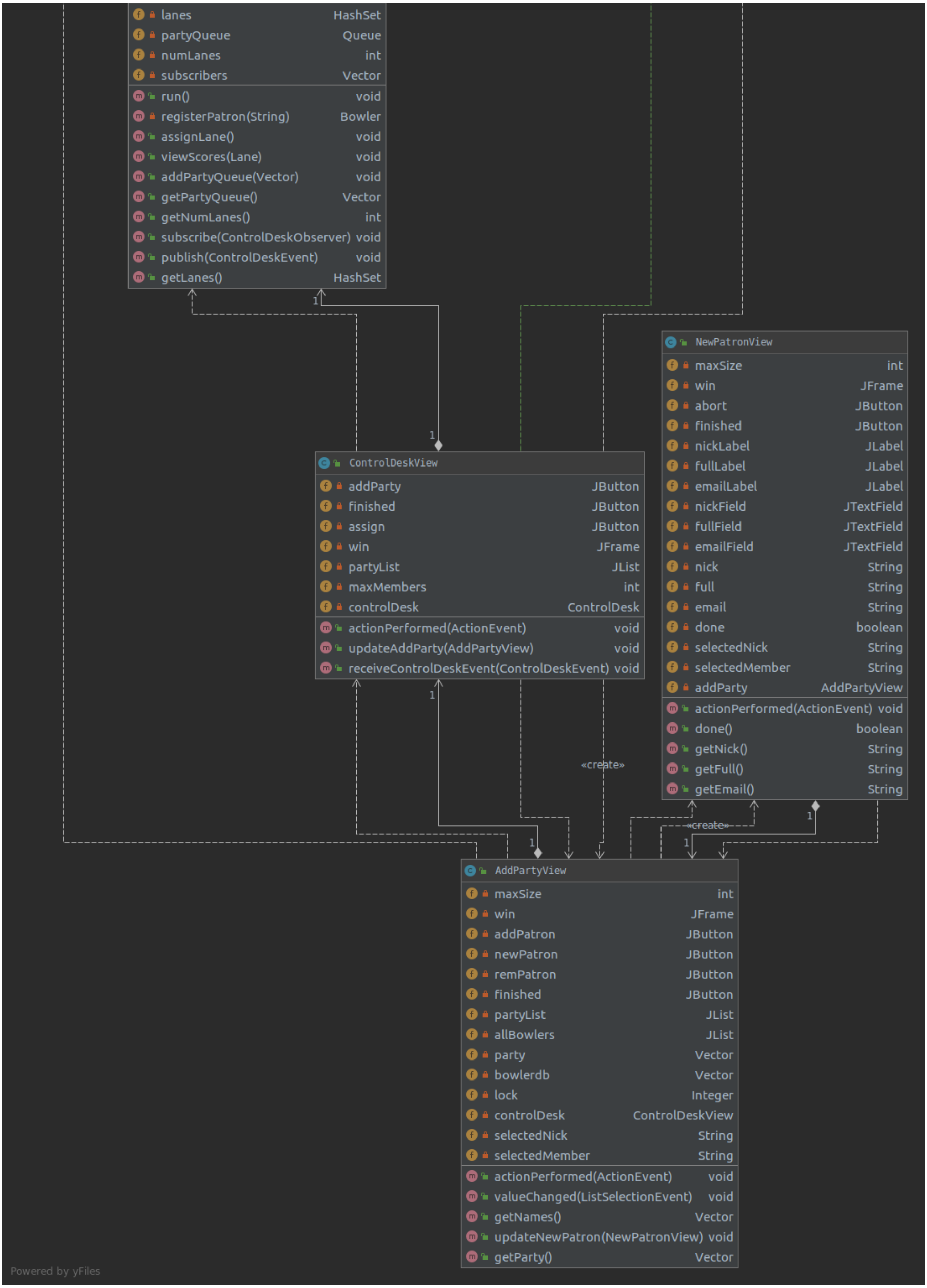
- **Associations and Dependencies** :

    - Every class is dependent to the ControlDesk class. The ControlDesk class spawns all the other classes.
    - `AddPartyView` and Bowler class are also related via association. The `AddPartyView` class has a list of all the Bowlers and hence depends on the Bowler class in order to obtain the Names and Nicknames of the bowlers.
    - The `ControlDesk` and `ControlDeskView` classes are associated as the `ControlDeskView` class uses an object of the `ControlDesk` class.
    - `AddPartyView` and `NewPatronView` are also associated in a similar manner
    - `ControlDeskView` and `AddPartyView` also have an association relation

- **Compositions**:

    - The `NewPatronView` class composes the `AddPartyView` class since a major functionality supported by the `AddPartyView` class is the add a patron to the list of already existing bolwers.
    - The `AddPartyView` class composes the `ControlDeskView` since the main purpose or functionality of the `ControlDesk`is to enable the players to add/create a party with a set of members/patrons and play the game. This is done via the `AddPartyView` class's members and methods.
    - The `ControlDeskiew` class inturn composes the control desk class since without the View provided by the GUI there would be no interface to support the `ControlDesk`class.





FUNCTIONALITY : SIMULATING A BALLTHROW

- The classes involved in this are :
  - Pinsetter
  - PinsetterObserver (interface)
  - PinsetterEvent
  - PinSetterView
  - Party
  - Bowler
  - Lane

- The cardinalities - *number of participating objects in any association between two classes* - are mentioned on the relationship arrows of the classes.

- To further indicate the creation of classes, it has also been specified as to which class is create from which parent class. The `create` written above the arrows indicates the parent-child relationship.

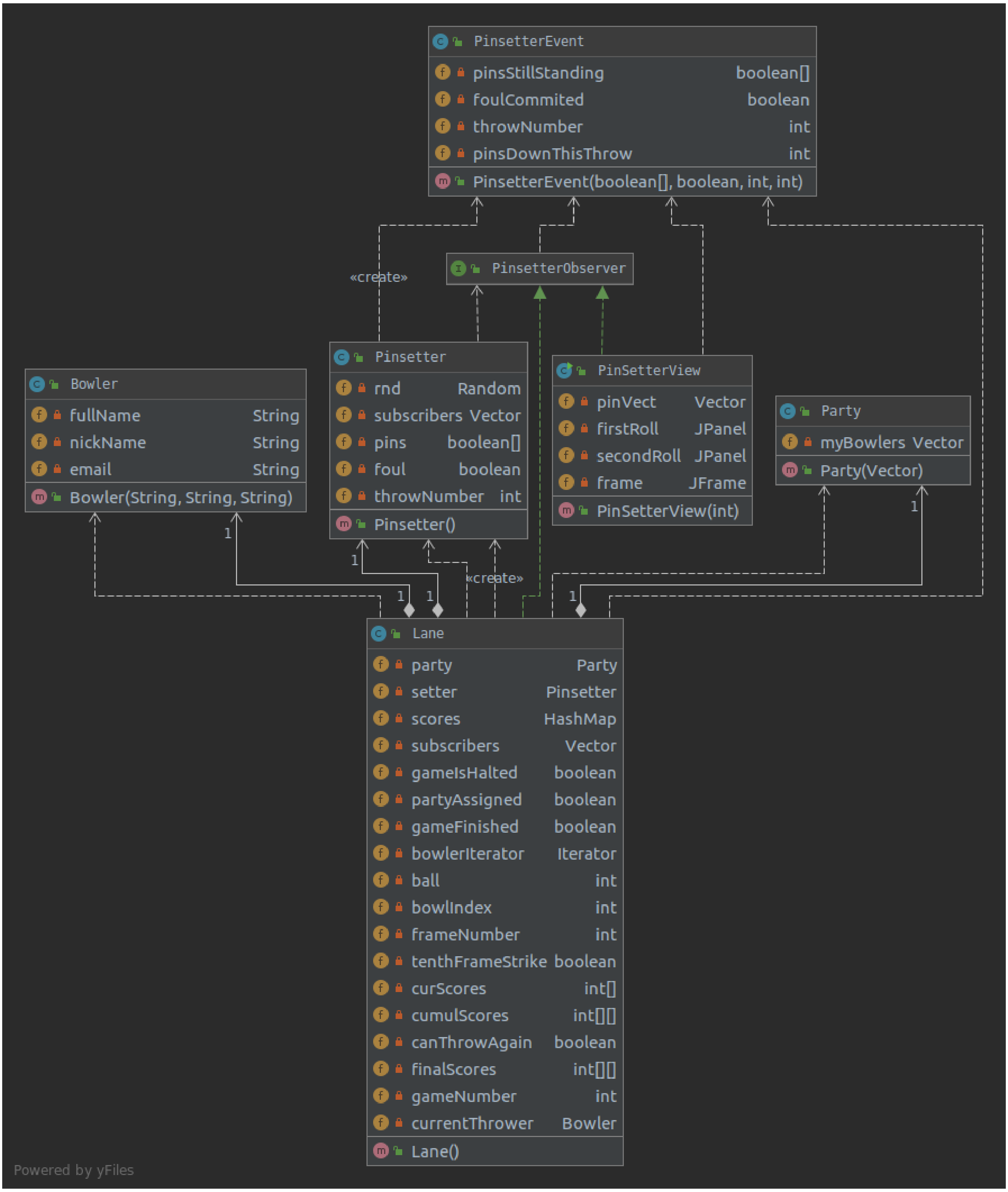
- The `PinsetterObserver` class is an interface class. Essentially serves as an interface between two or more classes that may not be able to interact otherwise. For eg: the `PinSetterView` and `Pinsetter` classes are interfaced to be able to share infromation and perform functions. This is also a demonstration of the *Adapter Design Pattern*

- All the methods and attributes associated with each class are shown in the UML class diagram as well.

The several arrows in the diagram represent different relationships between the multiple classes.

- **Associations and Dependencies** :
  - `Bowler` & `Lane` and `Party` & `Lane` classes are related via an association indicated by a solid arrow that shows that they have objects of these respective classes shared between them.
  - Dependencies are indicated by dotted arrows and show that a change in the class to which the arrow head points will and can cause a change to the dependent class.

- **Compositions**:
  - The `Bowler`, `Pinsetter` and `Party` classes exist only if the `Lane` class object exists. That is the `Lane` class composes the remaining classes. This is because, if there is a bowler then he must belogn to a lane, likewise if there's a party it must belong to a lane, if there's a pinsetter it must belong to a lane.


PinsetterEvent
pinsStillStanding boolean[]
foulCommited boolean
throwNumber int
pinsDownThisThrow int
PinsetterEvent(boolean[], boolean, int, int)
«create»
PinsetterObserver
Pinsetter
rnd Random
subscribers Vector
pins boolean[]
foul boolean
throwNumber int
Pinsetter()
PinSetterView
pinVect Vector
firstRoll JPanel
secondRoll JPanel
frame JFrame
PinSetterView(int)
Bowler
fullName String
nickName String
email String
Bowler(String, String, String)
Party
myBowlers Vector
Party(Vector)
1
1
1
1
1
1
«create»
Lane
party Party
setter Pinsetter
scores HashMap
subscribers Vector
gameIsHalted boolean
partyAssigned boolean
gameFinished boolean
bowlerIterator Iterator
ball int
bowlIndex int
frameNumber int
tenthFrameStrike boolean
curScores int[]
cumulScores int[][]
canThrowAgain boolean
finalScores int[][]
gameNumber int
currentThrower Bowler
Lane()
Powered by yFiles