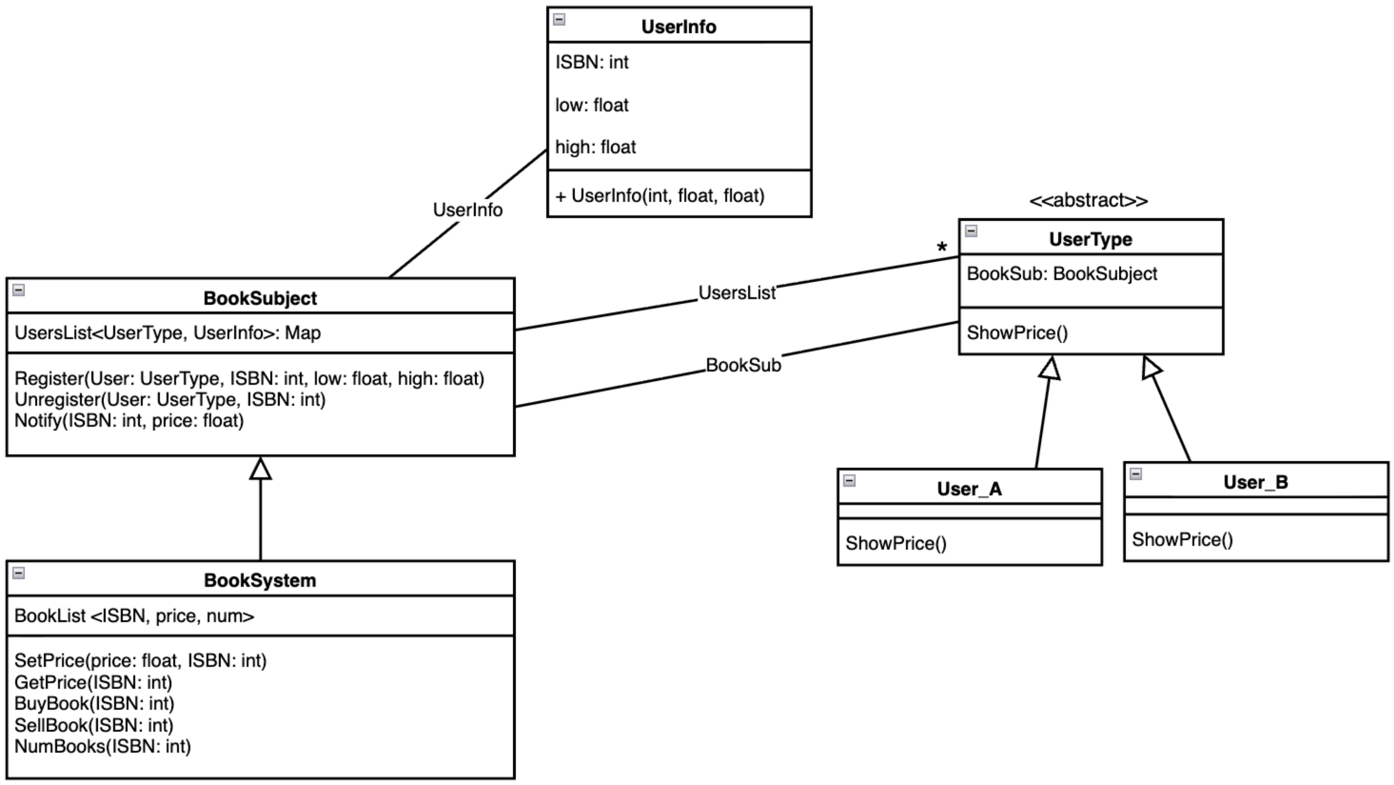
Problem #1:

(a) Class Diagram:



UserInfo
ISBN: int
low: float
high: float
+ UserInfo(int, float, float)
UserInfo
<<abstract>>
UserType
BookSub: BookSubject
ShowPrice()
*
UsersList
BookSub
BookSubject
UsersList<UserType, UserInfo>: Map
Register(User: UserType, ISBN: int, low: float, high: float)
Unregister(User: UserType, ISBN: int)
Notify(ISBN: int, price: float)
User_A
ShowPrice()
User_B
ShowPrice()
BookSystem
BookList <ISBN, price, num>
SetPrice(price: float, ISBN: int)
GetPrice(ISBN: int)
BuyBook(ISBN: int)
SellBook(ISBN: int)
NumBooks(ISBN: int)

Pseudocode:

```
Class UserInfo:
   attributes:
      ISBN: int
      low: float
      high: float
   Constructor
       UserInfo(ID, l, h):
       ISBN = ID
       low= l
       high=h
```

Class BookSubject:

attributes:

```
    UsersList: Map<UserType, UserInfo>  // Maps users to their Book interests
```

methods:

```
    +Register(User: UserType ISBN: int, low: float, high: float):
        // Register a user for notifications on a specific Book
        userInfo = new UserInfo(ISBN, low, high)
        UsersList.add(User, userInfo)

    +Unregister(User: UserType, ISBN: int):
        // Unregister a user from notifications for a specific Book
        for each (user, info) in UsersList:
            if info.ISBN == ISBN: & thisUser is the User
                UsersList.remove(user)

    +Notify(ISBN: int, price: float):
        // Notify users when the Book price changes
        for each (user, info) in UsersList:
            if info.ISBN == ISBN and (price < info.low or price > info.high):
                user.ShowPrice(ISBN)
```

```
class BookSystem:
  attributes:
   // List to store Book information: ISBN, Price, and Quantity
  // Note:BookList[0] = ISBN, BookList[1]= Price, BookList[2]= N
   BookList<Tuple<int, float, int> : List

  methods:
   // Method to set the price of a Book
   + SetPrice(price: float, ISBN: int):
      for each Book in BookList:
         if Book[0] == ISBN:
            stock[1] = price
         Notify(ISBN, price);// Potential Notification here! Go check ranges
            return
      // If ISBN not found, add new Book
      BookList.append((ISBN, price, 1))

   // Method to read the price of a Book
   + GetPrice(ISBN: int) -> float:
      for each Book in BookList:
         if Book[0] == ISBN:
            return Book[1]
      return 0.0  // Return 0.0 if ISBN not found


  // Method to buy a certain quantity of a stock
   + BuyBook(ISBN: int):
      for each Book in BookList:
         if Book[0] == ISBN:
            Book[2] += 1
            return
      // If ISBN not found, add new Book with quantity 0
      Book.append((ISBN, 0.0, 1))
```

```
 // Method to sell a Book
   + SellBook(ISBN: int):
      for each Book in BookList:
         if Book[0] == ISBN:
            if Book[2] >= 1:
               Book[2] -= 1
            else:
               print("Not enough Book to sell")
            return
      print("ISBN not found")
```

abstract class UserType:

attribute

```
   // Reference to the BookSubject (observer pattern)
   BookSub: BookSubject

   // Abstract method to display the Book price
   abstract DisplayPrice()
```

class User_A extends UserType:

```
   // Constructor to initialize User_A
   method __init__(BookSub: BookSubject):
      this.BookSub = BookSub
```

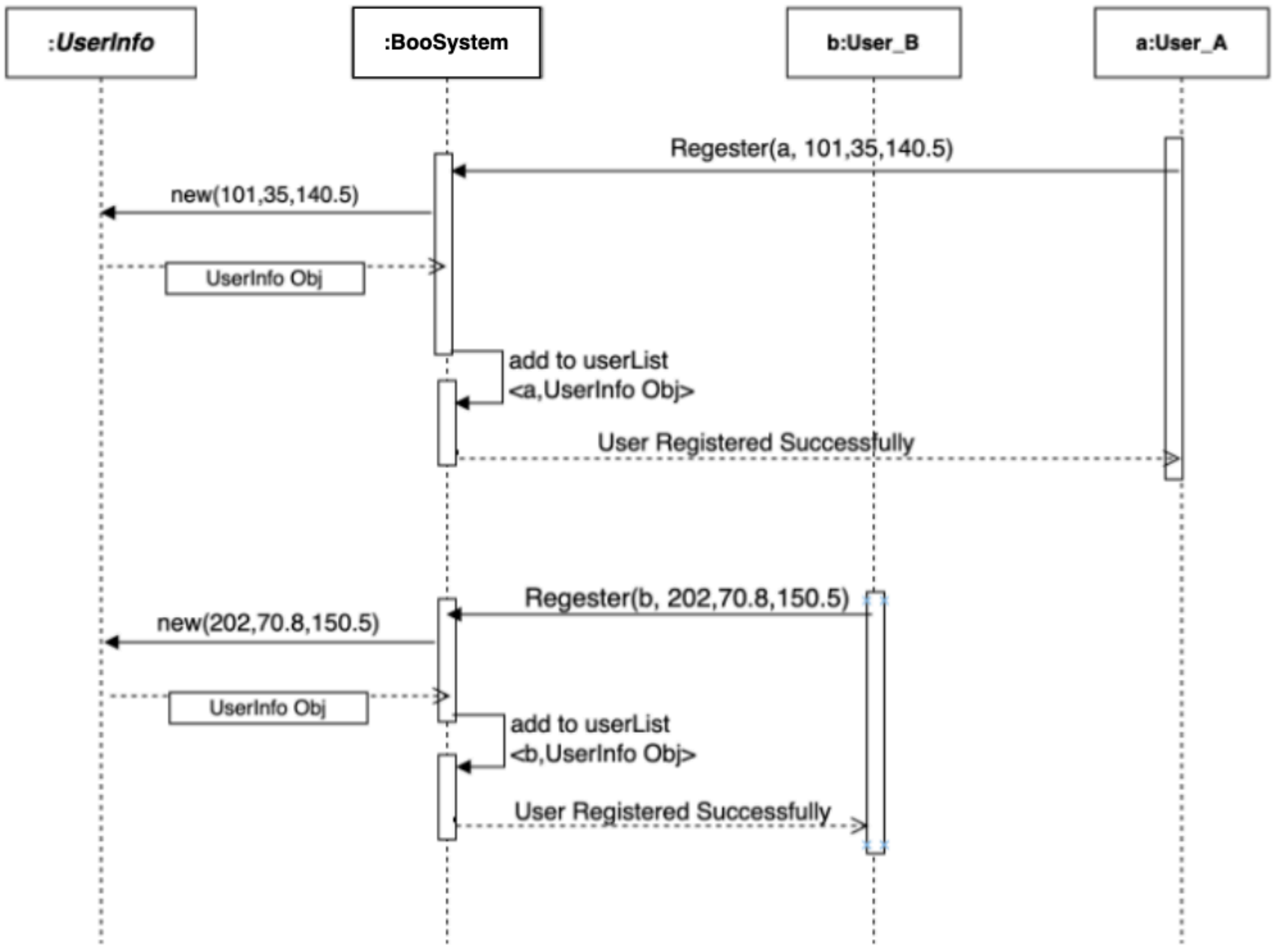
methods:

```
   // Implementation of DisplayPrice for User_A
   +DisplayPrice(ISBN):
      // Example: Display the price in a specific format for User_A
         price = BookSub.GetPrice(ISBN)
         print(f"User_A: Book ISBN {ISBN} has price {price}")
```

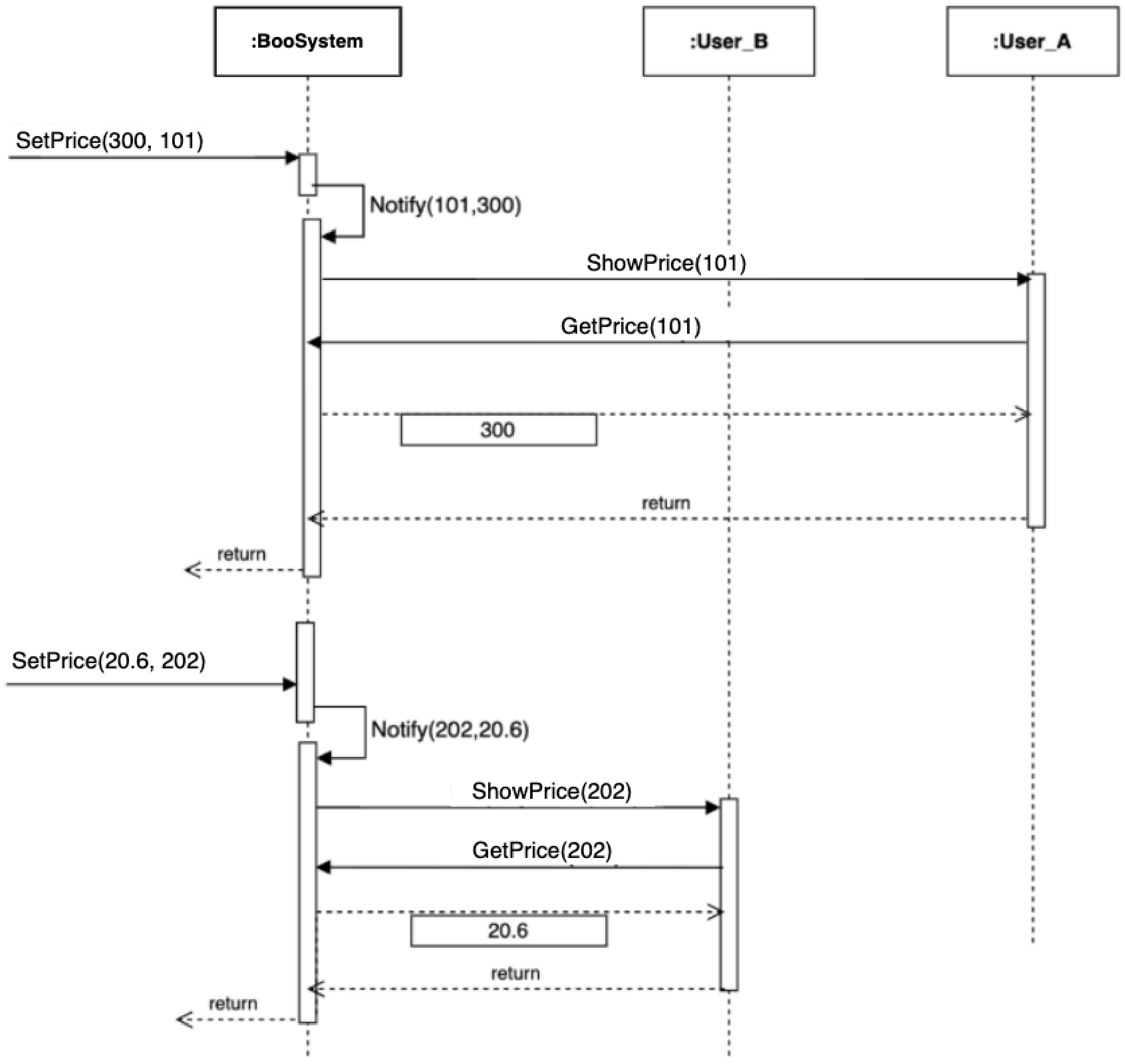
```
class User_B extends UserType:
    // Constructor to initialize User_B
    method __init__(BookSub: BookSubject):
        this.BookSub = BookSub

    methods:
    // Implementation of DisplayPrice for User_B
    method DisplayPrice(ISBN):
        // Example: Display the price in a different format for User_B
            price = BookSub.GetPrice(ISBN)
            print(f"User_B: ISBN {ISBN} is currently priced at {price}")
```

(b) Sequence Diagram(1)

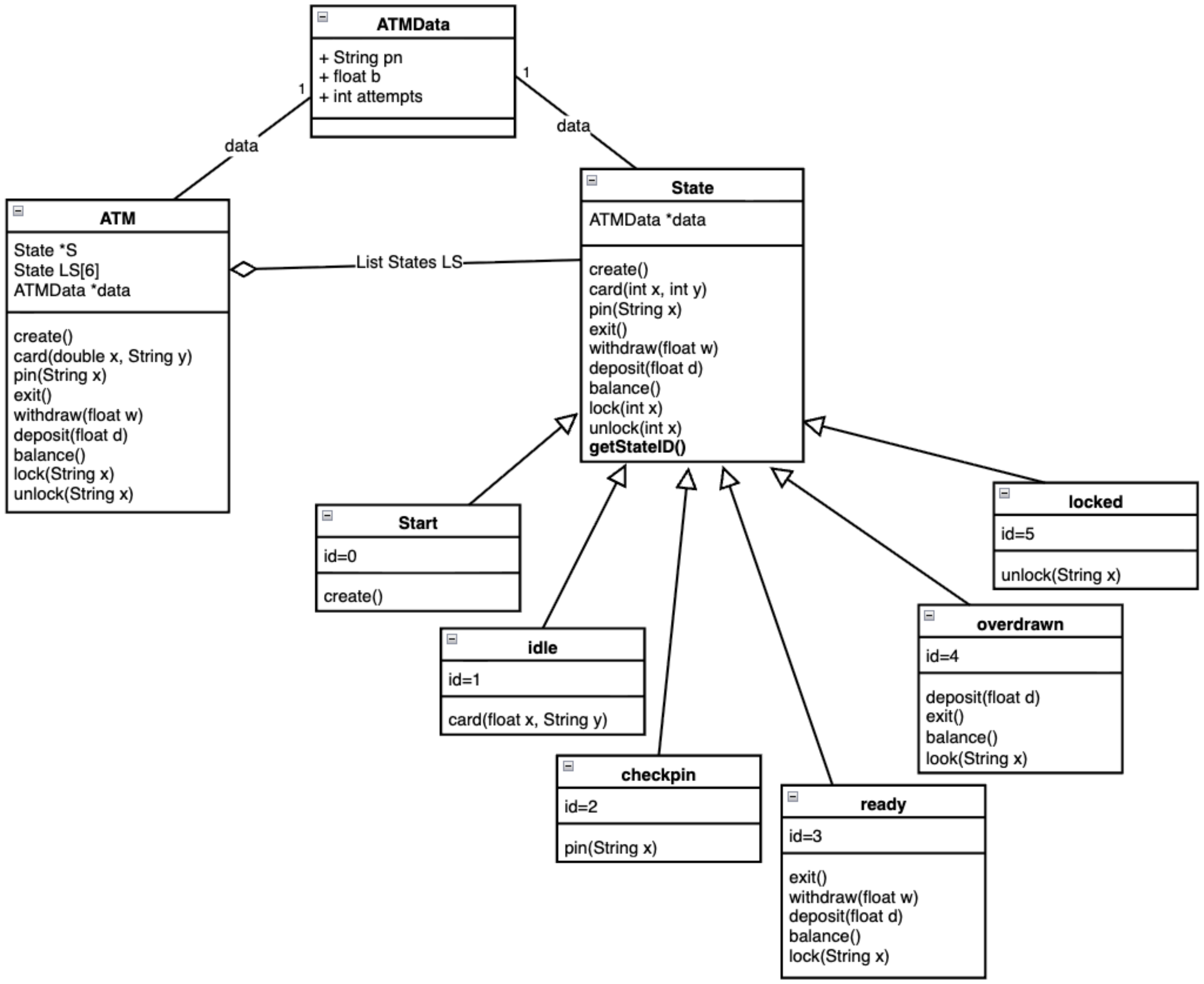


(b) Sequence Diagram(2)



Problem 2:

Centralized Version:



Pseudocode: Centralized Version

Class ATM:

```
S // points to the current obj
LS[0] // points to start obj
LS[1] // points to idle obj
LS[2] // points to checkpin obj
LS[3] // points to ready obj
LS[4] // points to overdrawn obj
LS[5]// points to locked obj
S = LS[0] // initialize state object to "start"
ATMData* data // points to objects of the ATM vars
```

Operations:

```
create(){
S-> create()
if( S-> getStateID() == 0 )
        S = LS[1]
}

card( double x, String y){
S-> card(x,y)
if( S-> getStateID() == 1 )
        S = LS[2]
}

pin(String x){
int temp_attempts = data.attempts // store the value of attempts before modification
S-> pin(x)
if(S->getStateID() == 2){
  if((x == data.pn) and (data.b >= 1000))
      S= LS[3]
  else if((x==data.pn) and (data.b < 1000))
        S=LS[4]
  else if(( x != data.pn) and (temp_attempts == 3))
        S= LS[1]
  else if(( x != data.pn) and (temp_attempts < 3))
        // Don't change the state
}  }
```

```
exit(){
S-> exit()
If ((S->getStateID() == 3) OR ( (S->getStateID() == 4))
S= LS[1]
}

withdraw(float w){
float temp_b = data.b // store the value of b before modification
S-> withdraw(w)
if(S-> getStateID() == 3){
  if((temp_b - w) < 1000) and ((temp_b - w) > 0)
     S= LS[4]
  else if((temp_b - w) >= 1000)
     // No state change  } }

deposit(float d){
float temp_b = data.b
S-> deposit(d)
If(S-> getStateID() == 4)
   if((temp_b + d) >= 1000)
      S= LS[3]  }

balance(){
S-> balance() }

lock(String x){
S-> lock(x)
If ((S-> getStateID() == 3) OR (S-> getStateID() == 4))
    If (x == data.pn)
         S=LS[5]
 }

unlock( String x){
S->unlock(x)
If (S->getStateID() == 5)
  if((x== data.pn) and (data.b >= 1000))
         S=LS[3]
 else if((x==data.pn) and (data.b < 1000)
        S=LS[4]
 }
```

Class ATMData
String pn
float b
int attempts
// All public access

Class State
  ATMData* data
   int id

***Operations*:**

All abstract except getStateID
create()
card(float x, String y)
pin(String x)
exit()
withdraw(float w)
deposit(float d)
balance()
lock(String x)
unlock(String x)
getStateID(){return id}

Class Start
id =0

Operations:

creat(){ }

Class idle
id =1

Operations:
card(float x, String y){
data.b= x
data.pn= y
data.attempts=0
}

Class checkpin
id= 2

Operations:

```
pin(String x){
  if((x == data.pn) and (data.b >= 1000))
      Display menu
  else if((x==data.pn) and (data.b < 1000))
         Display menu
  else if(( x != data.pn) and (data.attempts == 3))
         Eject card
  else if(( x != data.pn) and (data.attempts < 3))
         data.attempts = data.attempts +1
}
```

Class ready
id= 3

Operations:

```
exit(){
Eject card
}

withdraw(float w){
  if((data.b - w) < 1000) and ((data.b - w) > 0)
     data.b= data.b - w - 10
  else if((data.b - w) >= 1000)
      data.b= data.b - w
}

deposit(float d){
data.b= data.b + d
}

balance(){
Display balance b
}

lock(String x){  }        // No action in it
unlock(String x) { }    // No action in it
```

Class overdrawn
id=4

Operations:

```
deposit(float d){
if(data.b+d < 1000)
    data.b= data.b+ d -10
else if(data.b+d >= 1000)
       data.b= data.b+ d
}

balance(){
Display balance b
}

exit(){
Eject card
}

lock(String x){  }       // No action in it
```

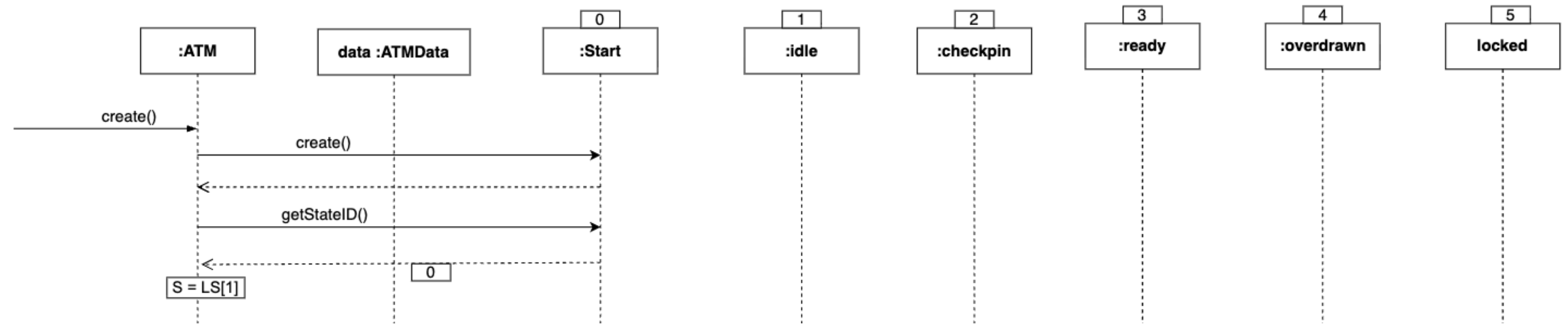
Class locked
id= 5

Operations:

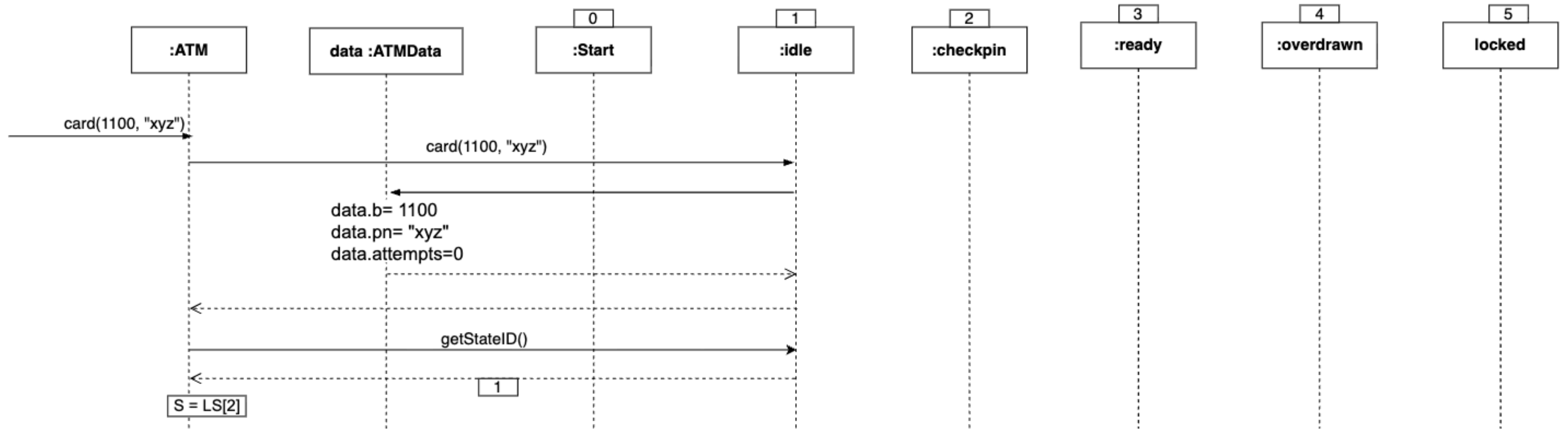
```
unlock(String x){  } // No action in it
```

Sequance Diagram: Centralized Version

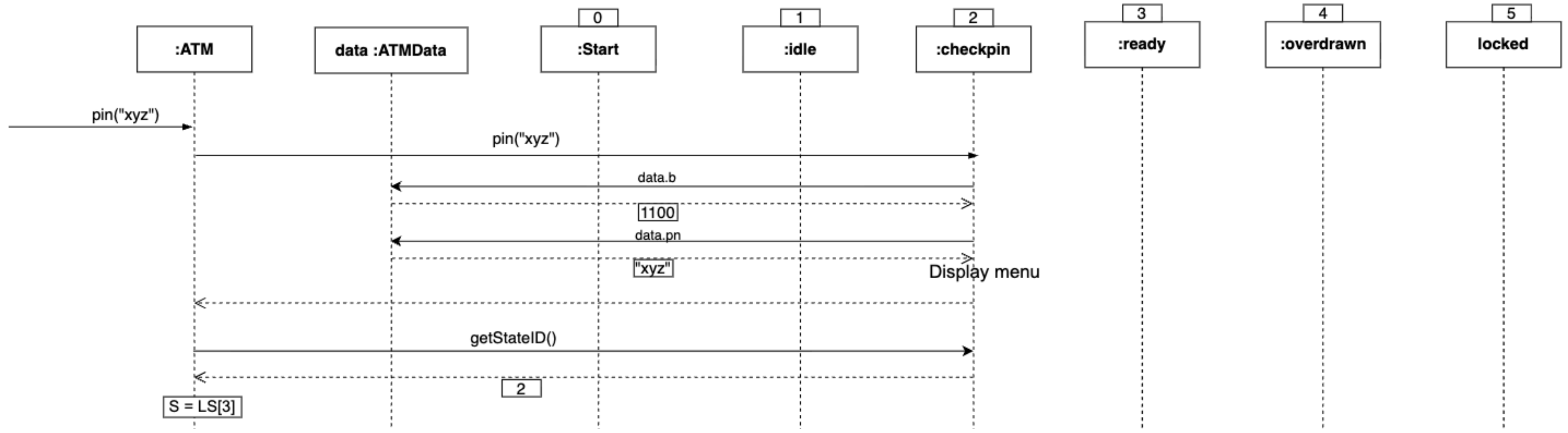
Event: creat()



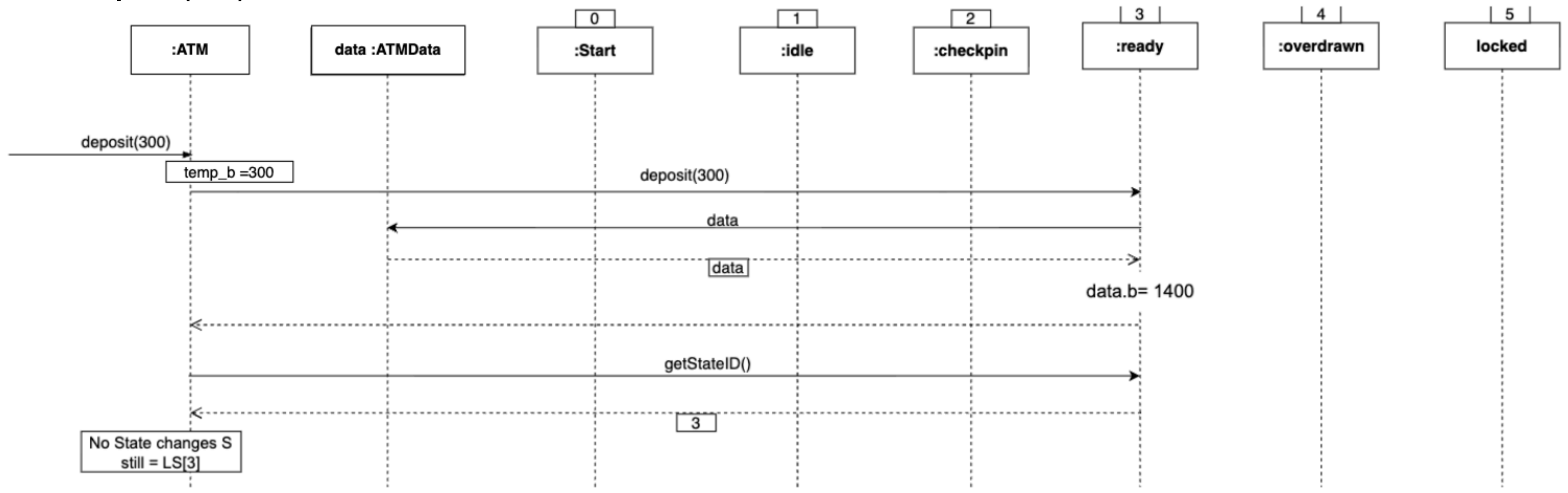
Event: card(1100, “xyz”)



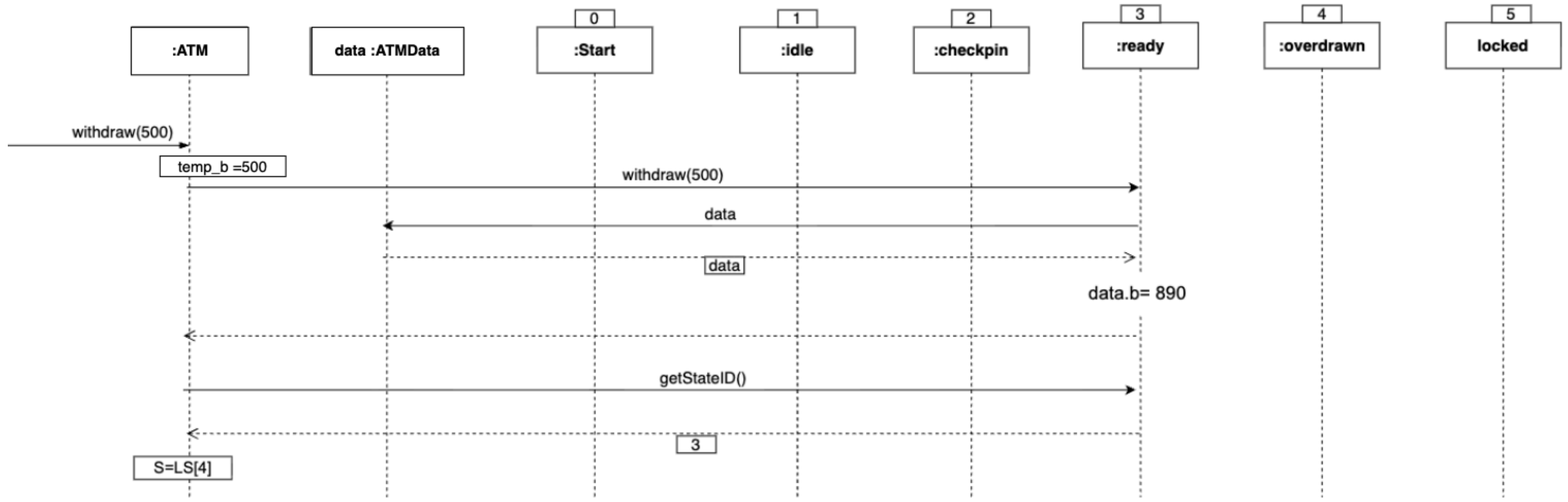
Event: pin(“xyz”)



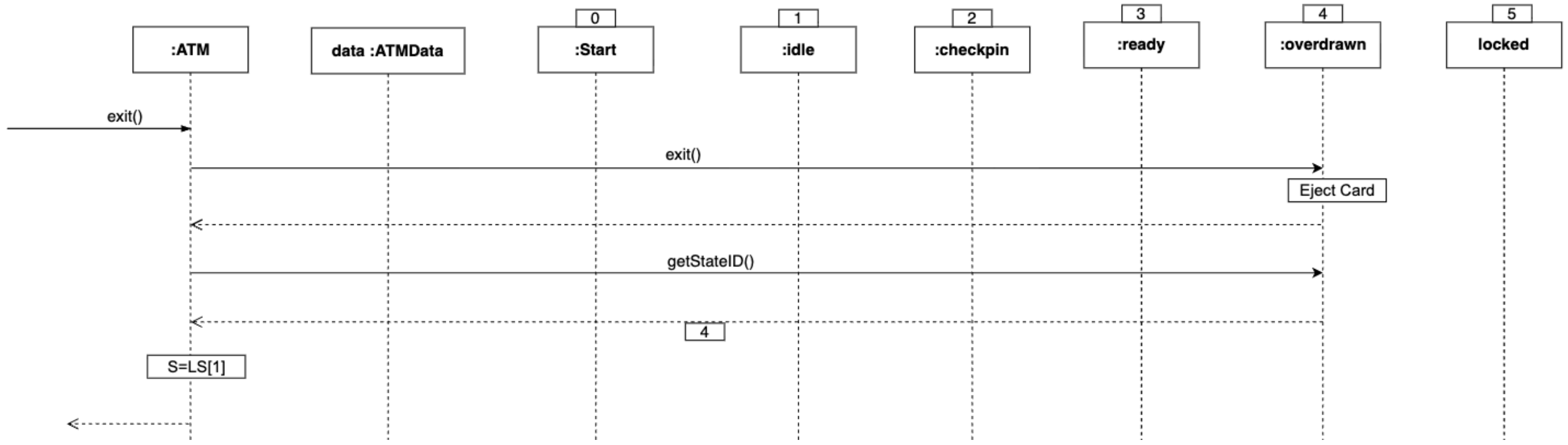
Event: deposit(300)



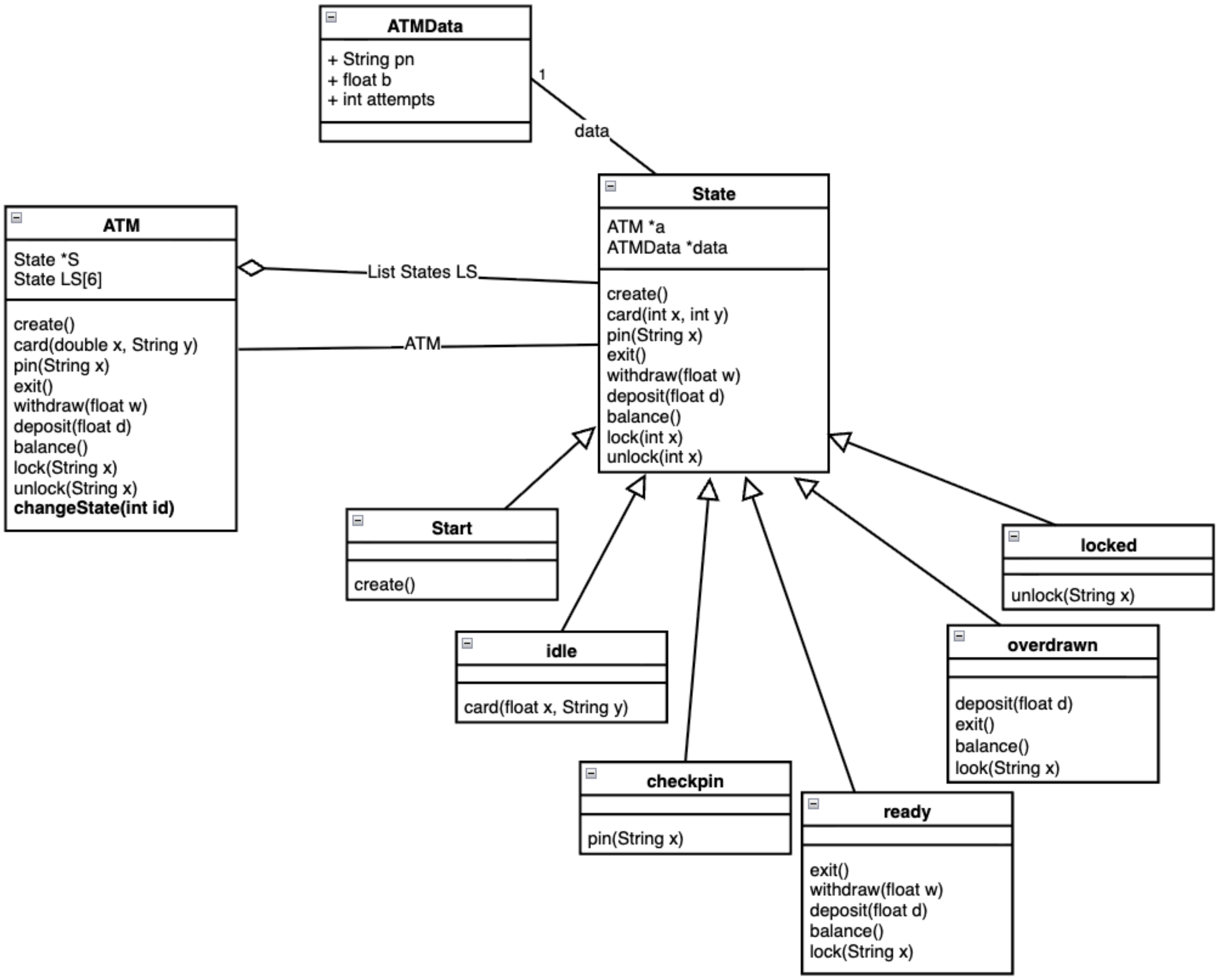
Event: withdraw(500)



Event: exit()



Problem 2:
Decentralized Version:



Class ATM:

```
S // points to the current obj
LS[0] // points to start obj
LS[1] // points to idle obj
LS[2] // points to checkpin obj
LS[3] // points to ready obj
LS[4] // points to overdrawn obj
LS[5]// points to locked obj
S = LS[0] // initialize state object to "start"
```

Operations:

```
create(){
S-> create() }

card( double x, String y){
S-> card(x,y)  }

pin(String x){
S-> pin(x)  }

exit(){
S-> exit() }

withdraw(float w){
S-> withdraw(w)  }

deposit(float d){
S-> deposit(d) }

balance(){
S-> balance() }

lock(String x){
S-> lock(x) }

unlock( String x){
S->unlock(x) }

changeState(int id){
 S= LS[id] }
```

Class ATMData

```
String pn
float b
int attempts
// All public access
```

Class State

```
  ATM* a
  ATMData* data
```

Operations:

All operations are abstract

```
create()
card(float x, String y)
pin(String x)
exit()
withdraw(float w)
deposit(float d)
balance()
lock(String x)
unlock(String x)
```

Class Start

Operations:

```
creat(){
   a->changeState(1)
}
```

Class idle

Operations:

```
card(float x, String y){
    data.b= x
    data.pn= y
   data.attempts=0
d-> changeState(2)
}
```

Class checkpin

Operations:

```
pin(String x){

  if((x == data.pn) and (data.b >= 1000)){
      Display menu
      a->changeState(3)
}
  else if((x==data.pn) and (data.b < 1000)){
          Display menu
           a->changeState(4)
}

  else if(( x != data.pn) and (data.attempts == 3)){
          Eject card
           a->changeState(1)
}
  else if(( x != data.pn) and (data.attempts < 3))
          data.attempts = data.attempts +1
}
```

Class ready

Operations:

```
exit(){
a->changeState(1)
Eject card
}

withdraw(float w){
  if((data.b - w) < 1000) and ((data.b - w) > 0) {
     data.b= data.b - w - 10
      a->changeState(4)
}
  else if((data.b - w) >= 1000)
      data.b= data.b - w
}
```

```
deposit(float d){
  data.b= data.b + d
 }


balance(){
  Display balance b
}


lock(String x){
    If (x == data.pn)
         a-> changeState(5)
 }
```

Class overdrawn

Operations:

```
deposit(float d){
   if((data.b + d) >= 1000){
       data.b= data.b+ d
      a->changeState(3)
  }
else if(data.b+d < 1000)
     data.b= data.b+ d -10
}

balance(){
Display balance b
}

exit(){
Eject card
a->changeState(1)
 }

lock(String x){
  if( x == data.pn)
      a->changeState(5)   }
```

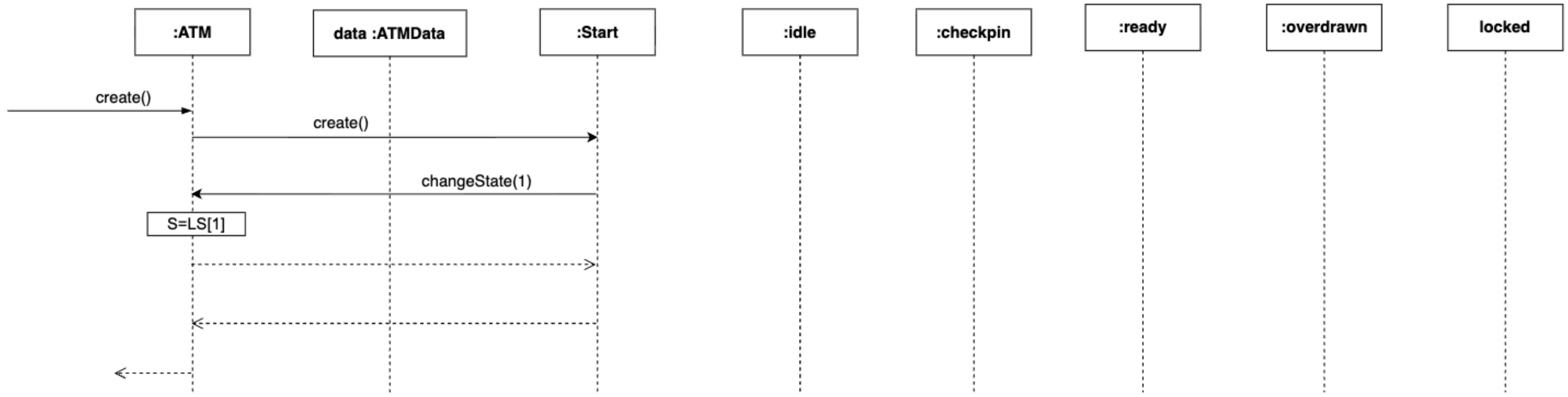
Class locked

Operations:

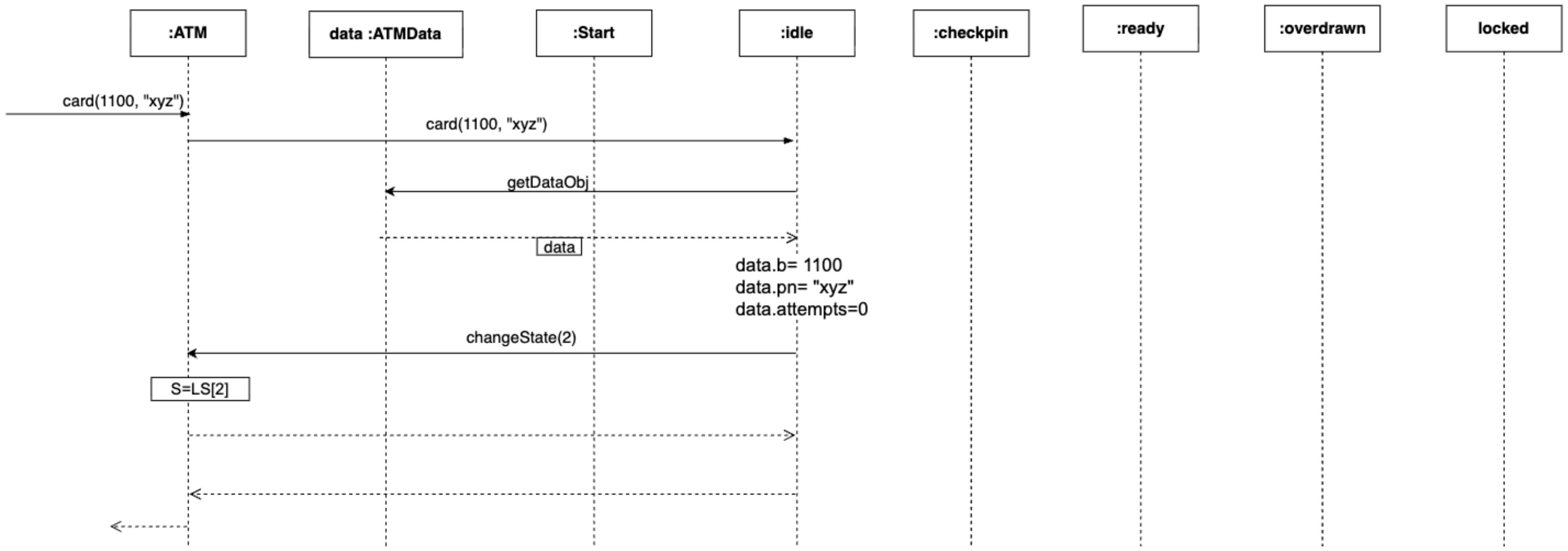
```
unlock( String x) {
  if((x== data.pn) and (data.b >= 1000)) {
         a->changeState(3)
}
 else if((x==data.pn) and (data.b < 1000){
        a->changeState(4)
    }
 }
```

Sequance Diagram: Decentralized Version

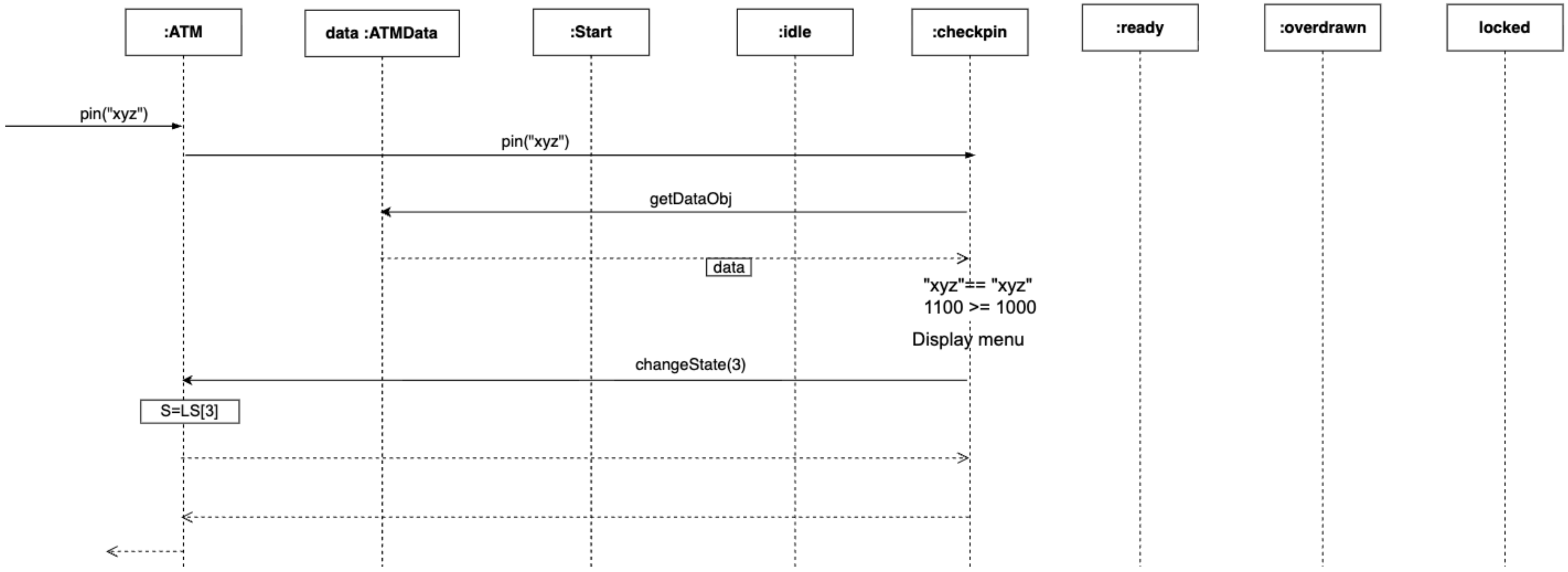
Event: creat()



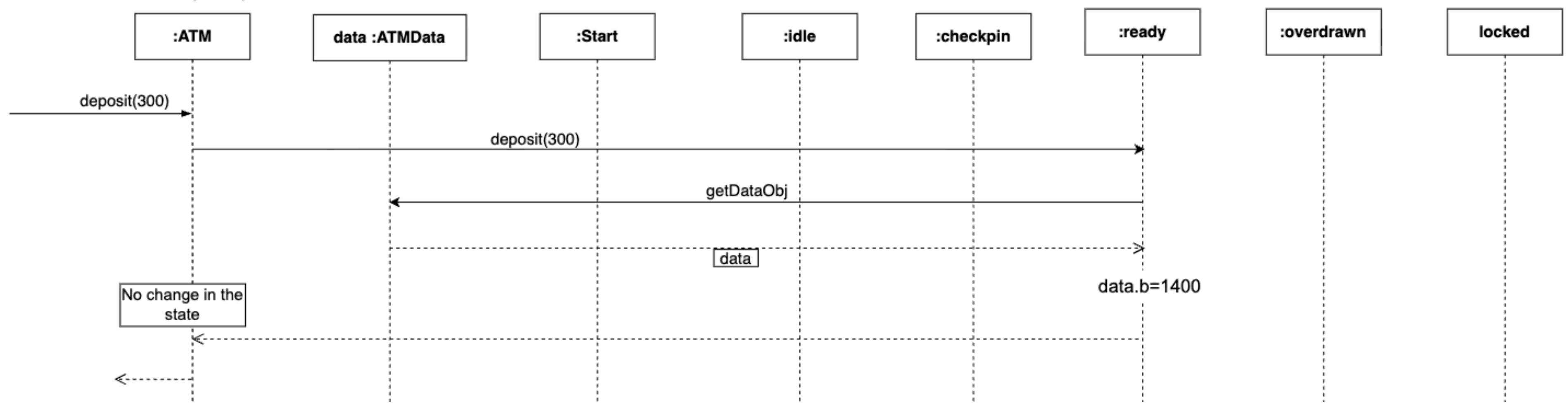
Event: card(1100,”xyz”)



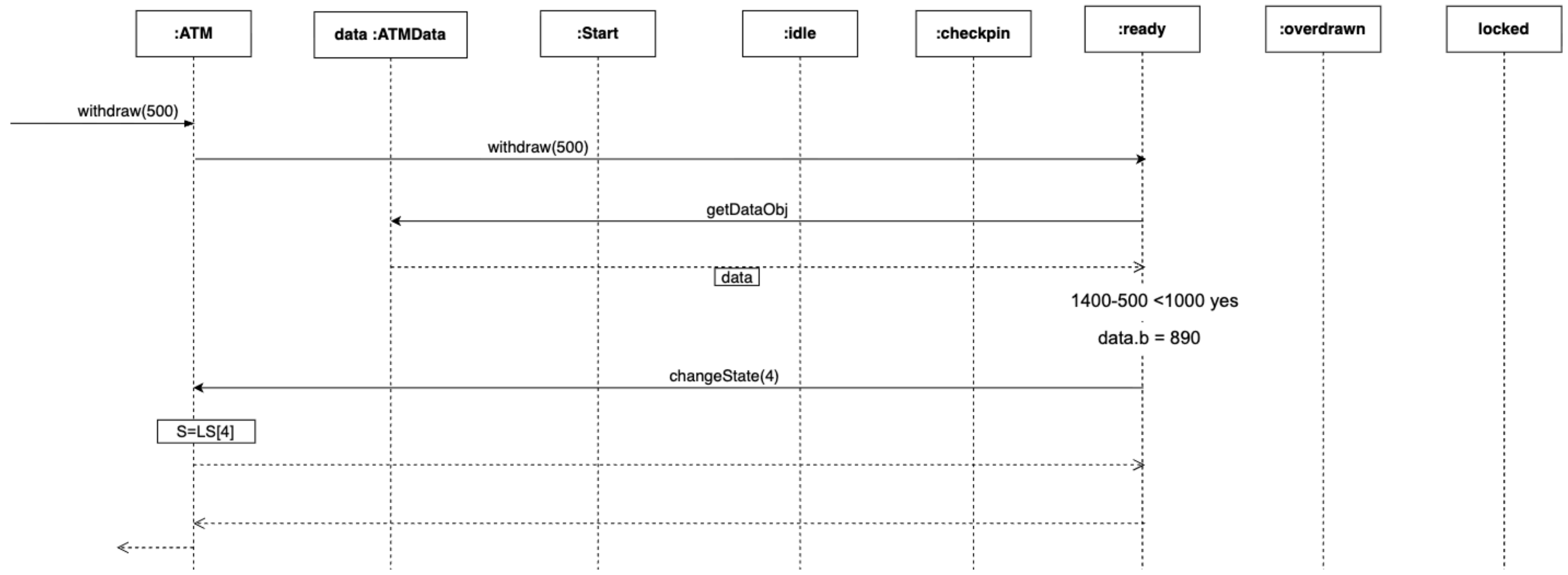
Event: pin(”xyz”)



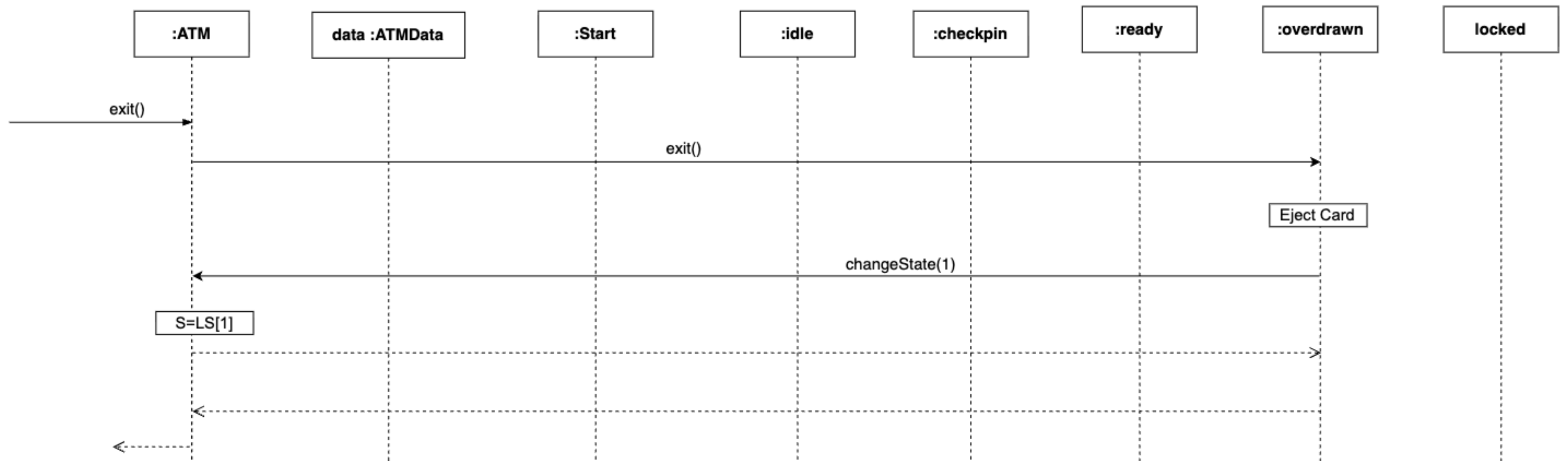
Event: deposit(300):



Event: withdraw(500):



Event: exit():



End Of File