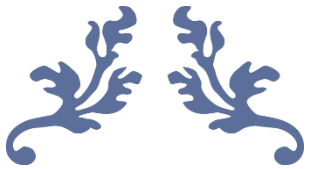


TU856-2
SOFTWARE ENGINEERING 2

EXTENDING AND TESTING A COMPREHENSIVE USE MODEL FOR THE LIBRARY SYSTEM IN USE ASSIGNMENT

COMPLETED BY: PAULINA CZARNOTA C21365726

DUE DATE: 05/05/2023

OVERVIEW

For this assignment I chose the “Extend and test a more comprehensive USE model for the Library system in USE” option for several reasons. Firstly, I have always been interested in the inner workings of a library system, and the Library option provides an excellent opportunity to apply the theoretical knowledge I have gained in my Software Engineering 2 course to a practical problem. Additionally, the Library option aligns with my interests, provides a practical application of the theoretical knowledge gained in class, and presents an exciting challenge.

Moreover, I have prior experience working on Library code in USE during the labs. This experience has given me confidence in my ability to work on the Library option for this assignment, and it has provided me with a solid foundation to build on and expand my understanding of the Library system's functionality. Working on the Library option for this assignment will allow me to build on my existing knowledge and skills and apply the techniques I learned during the labs to extend and test a more comprehensive USE model for the Library system.

The Library option requires the extension and testing of a more comprehensive USE model for the Library system, and this presents an exciting challenge for me. I am eager to explore the extra use case scenarios that can be included in the model, such as reserving a book, changing a book's status, and generating reports. These scenarios will allow me to gain a deeper understanding of the Library system's functionality and provide a more comprehensive model.

Furthermore, the Library option requires the creation of various types of diagrams, including class, sequence, state machine, and object, which will help me visualize the system's components and interactions. I am also excited about the opportunity to test the constraints with objects and use SOIL implementations for easier testing.

IMPROVEMENTS MADE TO LIBRARY SYSTEM USE CODE

The Library System code is a program that helps to keep track of books, copies, and members in a library system. The starter code for this program was a lab work, which I have improved by adding new classes, operations, enums, statemachines, associations, pre, and post-conditions.

To begin with, I added two new enums, BorrowStatus, and ReserveStatus, to the Book class. These enums help to keep track of the status of a book, whether it is borrowed or reserved. The enums have two states, Borrowed and NotBorrowed, and Reserved and NotReserved, respectively. This addition helps to ensure that the library system can accurately track the status of each book and its availability.

Next, I added a new operation called createCopy() to the Book class. This operation automatically creates new copies of a book and sets the borrowed and reserved statuses of each copy to NotBorrowed and NotReserved, respectively. It also adds each copy to the OfType association with the corresponding book. This addition makes it easier to add new books to the library system and ensures that all copies of a book have the same status.

To handle the borrowing and returning of books, I added the borrow() and return() operations to the Book class. These operations increment and decrement the available attribute, respectively. I also added a post-condition to the return() operation to ensure that the available attribute is updated correctly. This addition ensures that the number of available copies of a book is accurately reflected in the library system.

To handle the borrowing and returning of copies of books, I added the borrow() and return() operations to the Copy class. These operations update the borrowed and onLoan attributes of the copy and book objects, respectively. I also added a post-condition to the return() operation to ensure that the onLoan attribute of the book is updated correctly. This addition ensures that the library system can accurately track which copies of a book are borrowed and which are available.

To handle the reservation of copies of books, I added the reserve() and removeReservation() operations to the Copy class. These operations update the reserved attribute of the copy and insert or delete the copy and person objects into or from the HasReserved association, respectively. This addition ensures that the library system can accurately track which copies of a book are reserved and who has reserved them.

To handle the borrowing and returning of copies of books by a member, I added the borrow() and return() operations to the Member class. These operations use the okToBorrow() operation to check whether the member can borrow more copies of books. If the member can borrow, the operations insert or delete the copy and member objects into or from the HasBorrowed association, respectively. This addition ensures that the library system can accurately track which members have borrowed which copies of books.

To handle the viewing of borrowed copies of books by a member, I added the viewBorrowed() operation to the Member class. This operation uses a loop to iterate through the copies of books borrowed by the member and prints out the titles of each book. This

addition makes it easier for members to keep track of the books they have borrowed and when they are due to be returned.

To handle the payment of fines by a member, I added the payFine() operation to the Person class. This operation updates the fine attribute of the person object to reflect the payment made. This addition ensures that the library system can accurately track fines paid by members.

To handle the application of fines by an employee, I added the applyFine() operation to the Employee class. This operation updates the fine attribute of the person object to reflect the fine applied. This addition ensures that the library system can accurately track fines applied to members.

To ensure the correct functioning of the code, I added several pre and post-conditions and constraints to various operations and associations. These conditions and constraints ensure that the various operations are only executed under specific conditions and that the associations are updated correctly. This addition helps to ensure the accuracy and reliability of the library system.

Finally, I used state machines to model the different states that a book can be in, such as available, unavailable, and reserved. This helps to simplify the code and ensure that the correct operations are executed based on the current state of the book. I also modified the Book class's state machine to reflect the changes made to the borrow() and return() operations. The state machine now has three states, newTitle, available, and unavailable, and four transitions. The newTitle state is the initial state, while the available and unavailable states represent the states of a book with available copies and no available copies, respectively. This addition helps to ensure that the library system can accurately track the status of each book.

In conclusion, the improvements made to the Library System code have added new classes, operations, enums, state machines, associations, pre, and post-conditions to make the program more efficient and effective. These modifications have made the program more robust and capable of handling a more complex library system. The added classes, operations, enums, state machines, associations, pre, and post-conditions all work together to ensure that the library system is comprehensive and efficient. Overall, the improvements made to the provided starter code have created a more comprehensive and efficient library system that is better suited to handle various scenarios that can arise in a real-world library.

USE CODE

Library.use

```
model Library

enum BorrowStatus {Borrowed, NotBorrowed}
enum ReserveStatus {Reserved, NotReserved}



class Book
	attributes
		title: String
		author: String
		amount: Integer init = 2
		available: Integer init = 2

	operations
		createCopy()
		begin
			declare c: Copy;
			for i in Sequence{1..self.amount} do
				self.available := self.amount;
				c := new Copy;
				c.borrowed := #NotBorrowed;
				c.book := self;
				c.reserved := #NotReserved;
				insert(self, c) into OfType;
			end
		end
```

```
        borrow()
        begin
            self.available := self.available - 1;
        end

        return()
        begin
            self.available := self.available + 1;
        end

        statemachines
            psm States
            states
                newTitle: initial
                available[available > 0]
                unavailable[available = 0]
            transitions
                newTitle -> available { create }
                available -> unavailable { [available = 1] borrow() }
                available -> available { [available > 1] borrow() }
                available -> available { return() }
                unavailable -> available { return() }
        end
end


class Copy
    attributes
        book: Book
        borrowed: BorrowStatus init = #NotBorrowed
```

```
        reserved: ReserveStatus init = #NotReserved
        onLoan: Boolean

operations
        borrow(p: Person)
        begin
                for p1 in self.reservation do
                        if p = p1 then
                                self.reserved := #NotReserved;
                                delete(self, p) from HasReserved;
                        end
                end;

                if self.reserved = #NotReserved then
                        insert(p, self) into HasBorrowed;
                        self.borrowed := #Borrowed;
                        self.book.borrow();
                        p.amountBorrowed := p.amountBorrowed + 1;
                end
        end

        return(p: Person)
        begin
                delete(p, self) from HasBorrowed;
                self.borrowed := #NotBorrowed;
                self.book.return();
                p.amountBorrowed := p.amountBorrowed - 1;
        end

        reserve(p: Person)
```

```
            begin
                self.reserved := #Reserved;
                insert(self, p) into HasReserved;
                WriteLine('This copy has been reserved for you');
            end

            removeReservation(p: Person)
            begin
                if self.reserved = #NotReserved then
                    WriteLine('This Copy does not have a reservation to remove');
                else
                    self.reserved := #NotReserved;
                    delete(self, p) from HasReserved;
                end
            end
end


class Person
    attributes
        name: String
        address: String
        amountBorrowed: Integer init = 0
        no_onloan: Integer init = 0
        limit: Integer init = 6
        fine: Integer init = 0
        status: String

    operations
        borrow(c: Copy)
```

```
begin
        declare ok: Boolean;
        ok := self.okToBorrow();
        c.borrow(self);
end

okToBorrow(): Boolean
begin
        if self.no_onloan < 2 then
                result := true
        else
                result := false
        end
end

return(c: Copy)
begin
        delete(self, c) from HasBorrowed;
        self.no_onloan := self.no_onloan - 1;
        c.return(self)
end

viewBorrowed()
begin
        for c in self.borrowed do
                WriteLine(c.book.title);
        end;
end

payFine(amount: Integer)
```

```
        reserve(c: Copy)
        begin
            c.reserve(self)
        end

        removeReservation(c: Copy)
        begin
            c.removeReservation(self)
        end
end


class Employee < Person
    attributes
        employeeID: Integer
        role: String

    operations
        applyFine(p: Person, amount: Integer)
        begin
            if p.fine + amount <= 50 then
                p.fine := p.fine + amount
            else
                WriteLine('Fine amount exceeds limit of 50');
            end
        end
end


class Member < Person
```

```
        attributes
                memberID: Integer
end


association OfType between
        Book[1] role book
        Copy[0..*] role type
end

association HasBorrowed between
        Person[0..1] role borrower
        Copy[0..*] role borrowed
end

association HasReserved between
        Copy[0..1] role copy
        Person[0..*] role reservation
end


constraints

context Person::borrow(c: Copy)
        pre amountBorrowedLimit: self.amountBorrowed < self.limit
        pre notAlreadyBorrowed: self.borrowed -> excludes(c)
        pre notSameBook: self.borrowed.book -> excludes(c.book)
        pre noOnLoanLimit: self.no_onloan < 2

context Copy::borrow(p: Person)
```

```
        pre notAlreadyBorrowed: self.borrowed = #NotBorrowed

context Book::borrow()
        post sufficientAvailable: self.available >= 0

context Person::return(c: Copy)
        pre borrowedCopy: self.borrowed -> includes(c)
        post notBorrowedCopy: self.borrowed -> excludes(c)

context Person::payFine(amount: Integer)
        pre hasFine: self.fine > 0
        post fineNotNegative: self.fine >= 0

context Person::reserve(c: Copy)
        pre noReservation: c.reservation -> isEmpty()

context Copy::reserve(p: Person)
        pre notReserved: self.reserved = #NotReserved
        pre notAlreadyBorrowed: self.borrowed = #NotBorrowed

context Person::removeReservation(c: Copy)
        pre hasReservation: c.reservation -> includes(self)
        post noReservation: c.reservation -> isEmpty()

context Employee::applyFine(p: Person, amount: Integer)
        pre fineNotExceedLimit: p.fine < 50
        post fineNotExceedLimit2: p.fine < 50
```

EXPLANATION OF USE CODE

The provided code represents a model for a library system. It consists of several classes, including Book, Copy, Person, Employee, and Member, along with several associations and constraints.

The Book class represents a book with attributes such as title, author, amount, and available. The amount attribute represents the total number of copies of the book, and the available attribute represents the number of copies currently available for borrowing. The Book class also defines operations such as createCopy(), borrow(), and return(). The createCopy() operation creates new copies of the book and sets their initial statuses to NotBorrowed and NotReserved. The borrow() and return() operations update the available count of the book when a copy is borrowed or returned.

The Copy class represents a physical copy of a book with attributes such as book, borrowed, reserved, and onLoan. The book attribute is a reference to the Book class, and the borrowed and reserved attributes are enums representing the statuses of the copy. The Copy class also defines operations such as borrow(), return(), reserve(), and removeReservation(). The borrow() operation borrows the copy to a Person and updates the statuses of the copy and the borrower. The return() operation returns the copy and updates the statuses of the copy and the borrower. The reserve() operation reserves the copy for a Person, and the removeReservation() operation removes the reservation of the copy for a Person.

The Person class represents a person who can borrow books with attributes such as name, address, amountBorrowed, no_onloan, limit, and fine. The amountBorrowed attribute represents the number of copies currently borrowed by the person, and the no_onloan attribute represents the number of copies currently on loan by the person. The Person class also defines operations such as borrow(), return(), viewBorrowed(), payFine(), reserve(), and removeReservation(). The borrow() operation borrows a copy for the person and updates the statuses of the copy and the person. The return() operation returns a copy and updates the statuses of the copy and the person. The viewBorrowed() operation displays the titles of books currently borrowed by the person. The payFine() operation allows the person to pay fines incurred for late returns. The reserve() and removeReservation() operations allow the person to reserve and remove reservations for copies.

The Employee class extends the Person class and represents an employee of the library with attributes such as employeeID and role. The Employee class also defines an operation applyFine() that allows the employee to apply fines to a person's account.

The Member class extends the Person class and represents a member of the library with an additional attribute memberID.

The associations between the classes are defined using the association keyword. The OfType association between Book and Copy represents the relationship between a book and its copies. The HasBorrowed association between Person and Copy represents the relationship between a person and the copies they have borrowed. The HasReserved association between Copy and Person represents the relationship between a copy and the persons who have reserved it.

The code also defines constraints using the context keyword. The constraints ensure that the operations are performed only under certain conditions, such as checking if a person has already borrowed a copy before borrowing it again or checking if a copy is available for borrowing before reserving it.

SOIL CODE

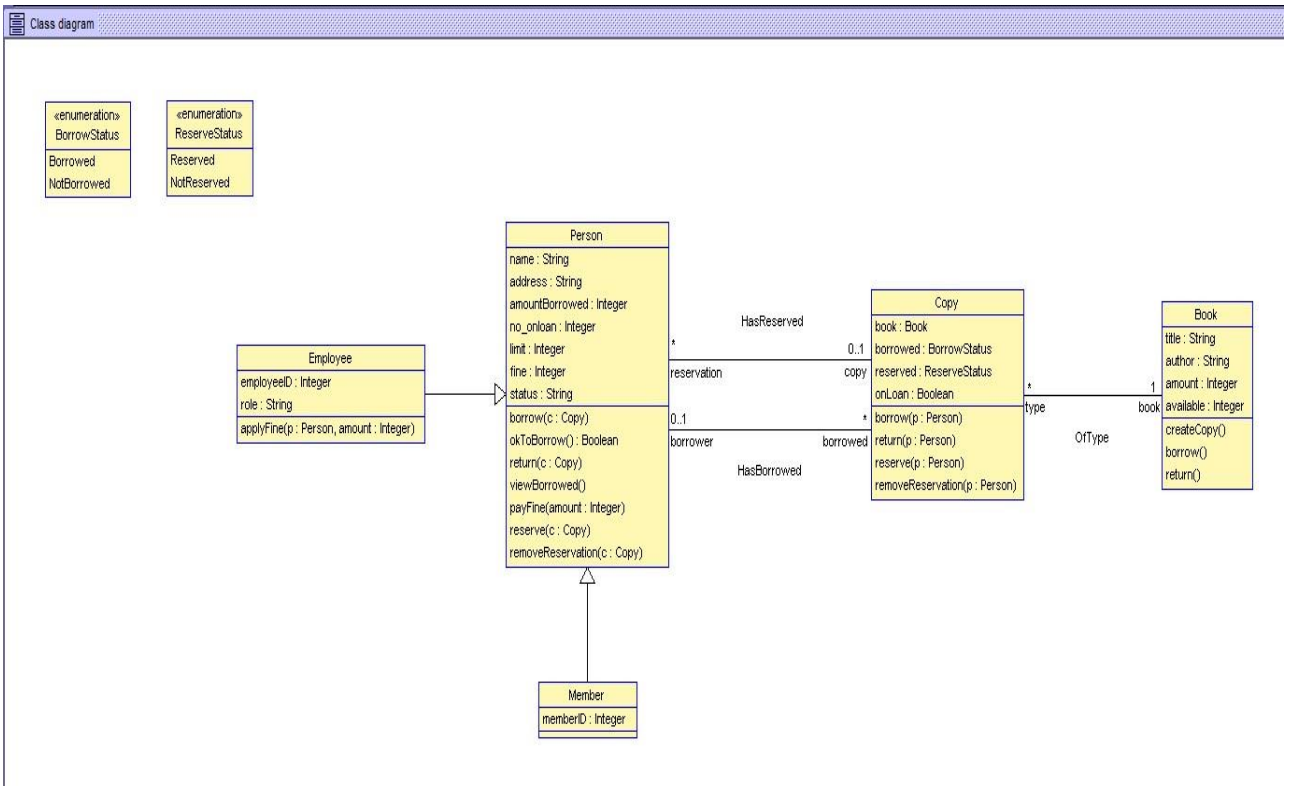
Library.soil

```
-- Script generated by USE 6.0.0

!new Member('Danny')
!Danny.name := 'Daniel Rodriguez'
!Danny.address := 'Apt 10, Manor Street, Dublin 7'
!Danny.amountBorrowed := 3
!Danny.no_onloan := 0
!Danny.limit := 6
!Danny.fine := 0
!Danny.status := 'Borrowed'
!Danny.memberID := 122112

!new Member('Bella')
!Bella.name := 'Isabella Soprano'
!Bella.address := 'North Circular Road, Dublin'
!Bella.amountBorrowed := 1
!Bella.no_onloan := 1
!Bella.limit := 6
!Bella.fine := 0
!Bella.status := 'Borrowed'
!Bella.memberID := 433443
```

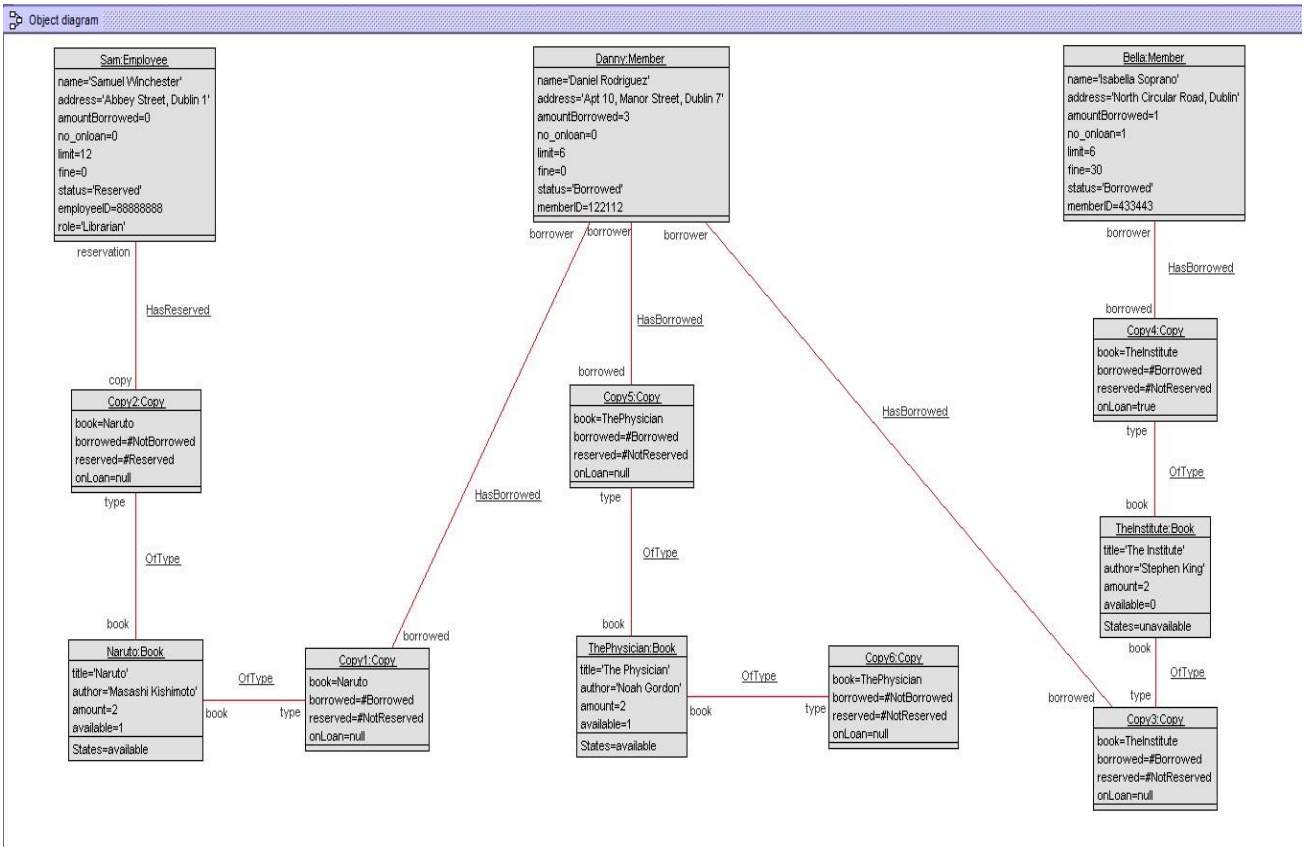
```
!new Employee('Sam')
!Sam.name := 'Samuel Winchester'
!Sam.address := 'Abbey Street, Dublin 1'
!Sam.amountBorrowed := 0
!Sam.no_onloan := 0
!Sam.limit := 12
!Sam.fine := 0
!Sam.status := 'Reserved'
!Sam.employeeID := 88888888
!Sam.role := 'Librarian'

!new Book('Naruto')
!Naruto.title := 'Naruto'
!Naruto.author := 'Masashi Kishimoto'
!Naruto.amount := 2
!Naruto.available := 0
!Naruto.createCopy()

!new Book('TheInstitute')
!TheInstitute.title := 'The Institute'
!TheInstitute.author := 'Stephen King'
!TheInstitute.amount := 2
!TheInstitute.available := 1
!TheInstitute.createCopy()

!new Book('ThePhysician')
!ThePhysician.title := 'The Physician'
!ThePhysician.author := 'Noah Gordon'
!ThePhysician.amount := 2
!ThePhysician.available := 1
```

```
!ThePhysician.createCopy()

!Danny.borrow(Copy1)
!Danny.borrow(Copy5)

!Bella.reserve(Copy4)
!Bella.removeReservation(Copy4)
!Bella.borrow(Copy4)

!Sam.reserve(Copy2)

!Copy4.onLoan := true

!Sam.applyFine(Bella, 30)
!openter Bella payFine(40)
!Bella.fine := (Bella.fine - 40)
!opexit

!Sam.applyFine(Bella, 100)

!Danny.borrow(Copy3)
```

CLASS DIAGRAM

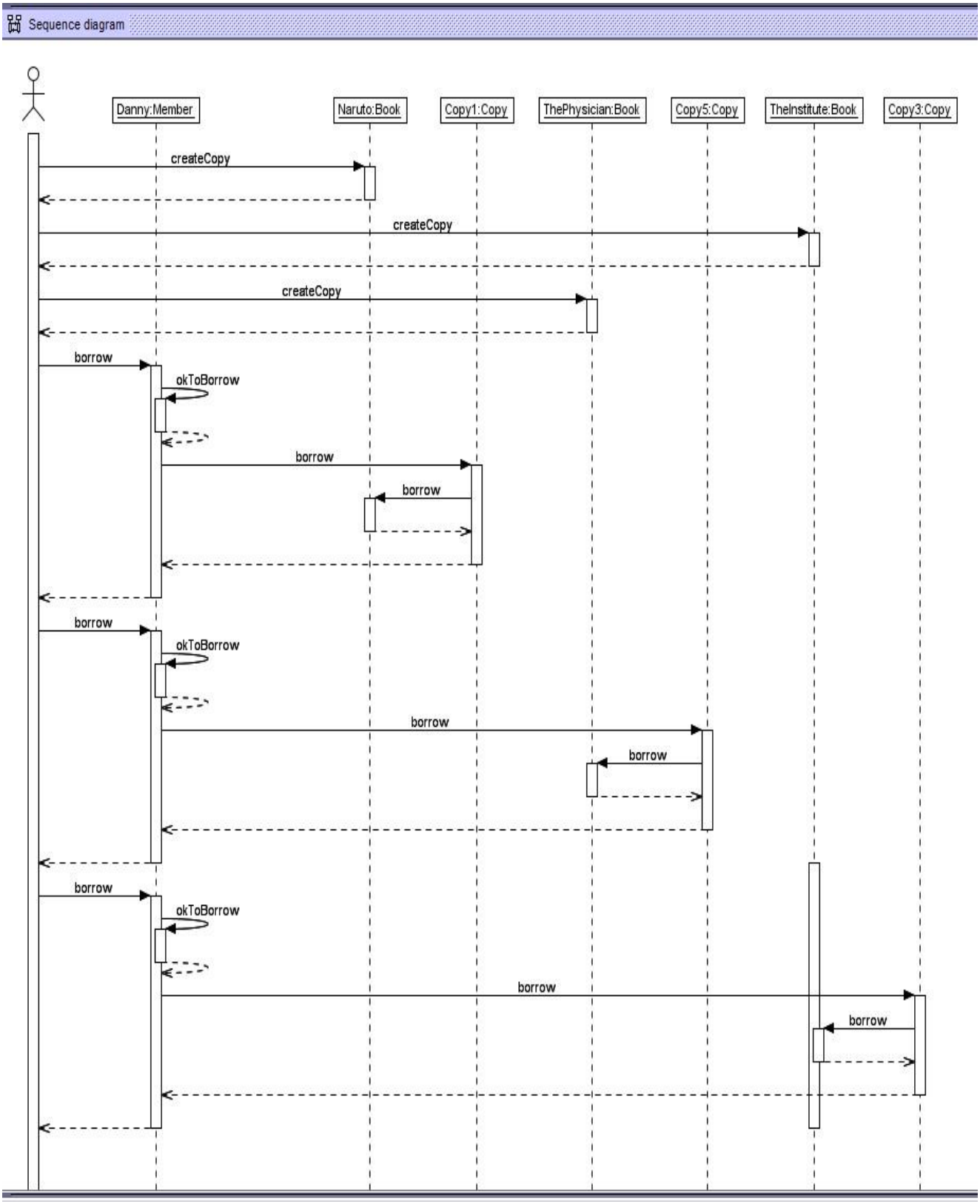


OBJECT DIAGRAM

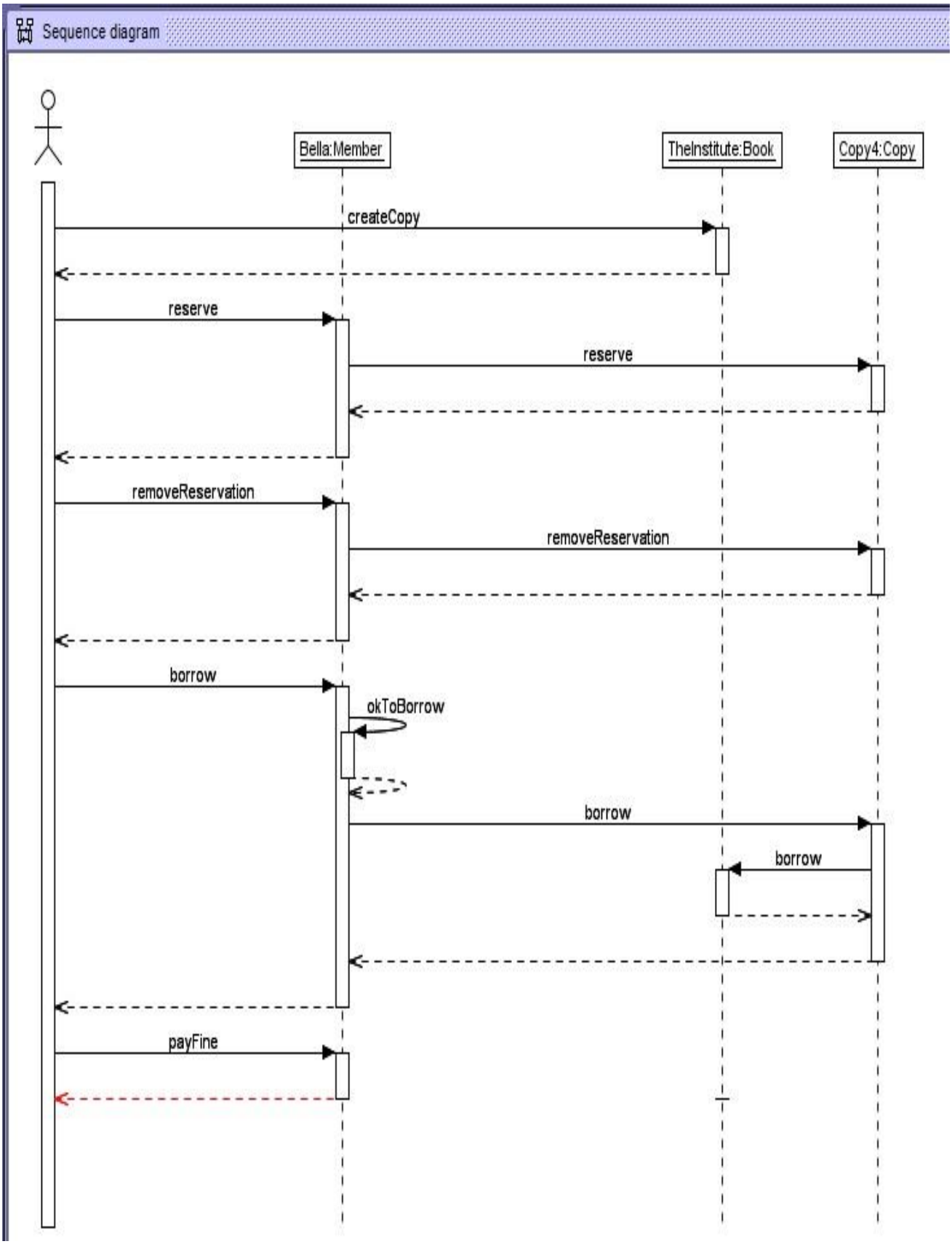


SEQUENCE DIAGRAMS

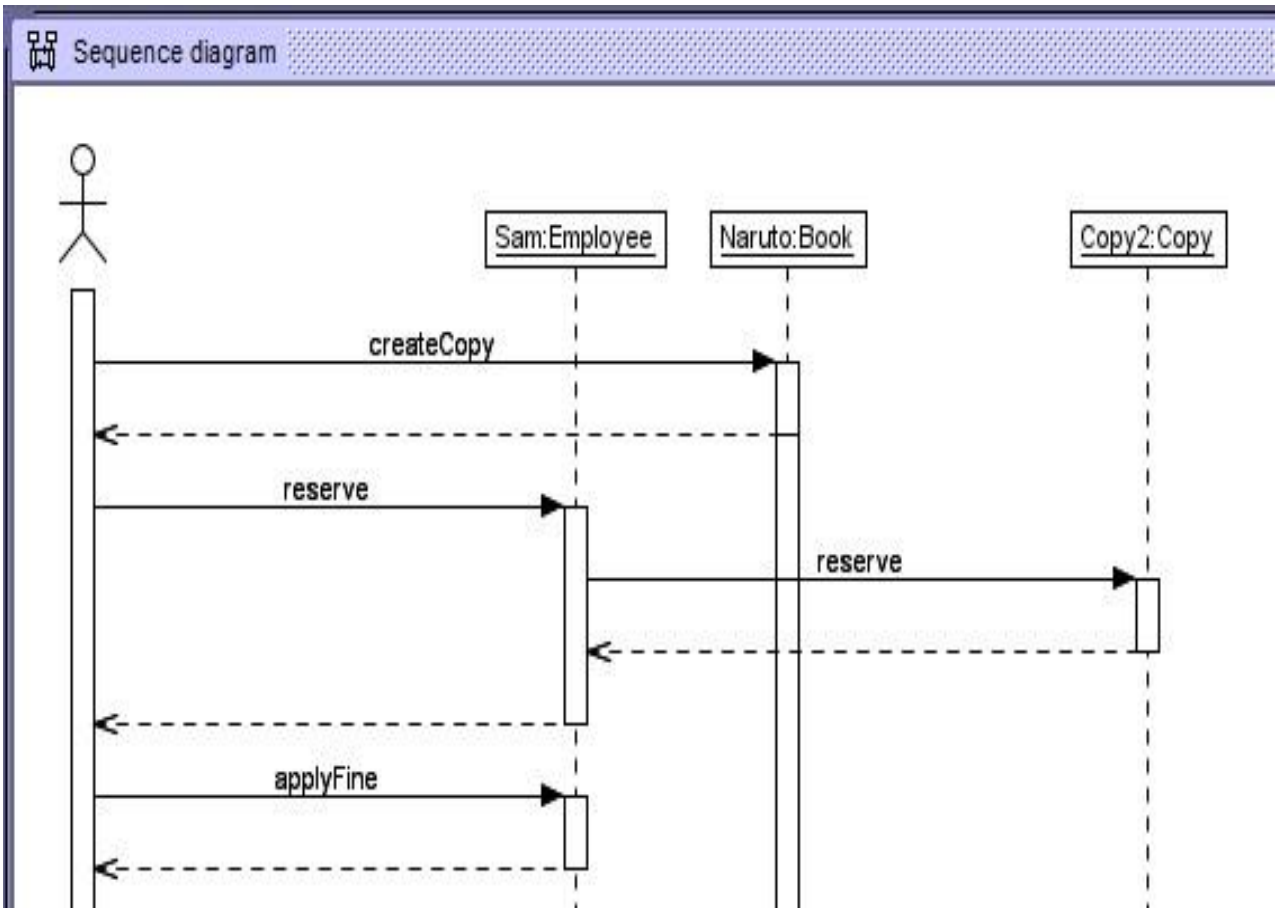
Danny Member For Borrowing



Bella Member For Reserving, Removing Reservation, Borrowing & Paying Fine

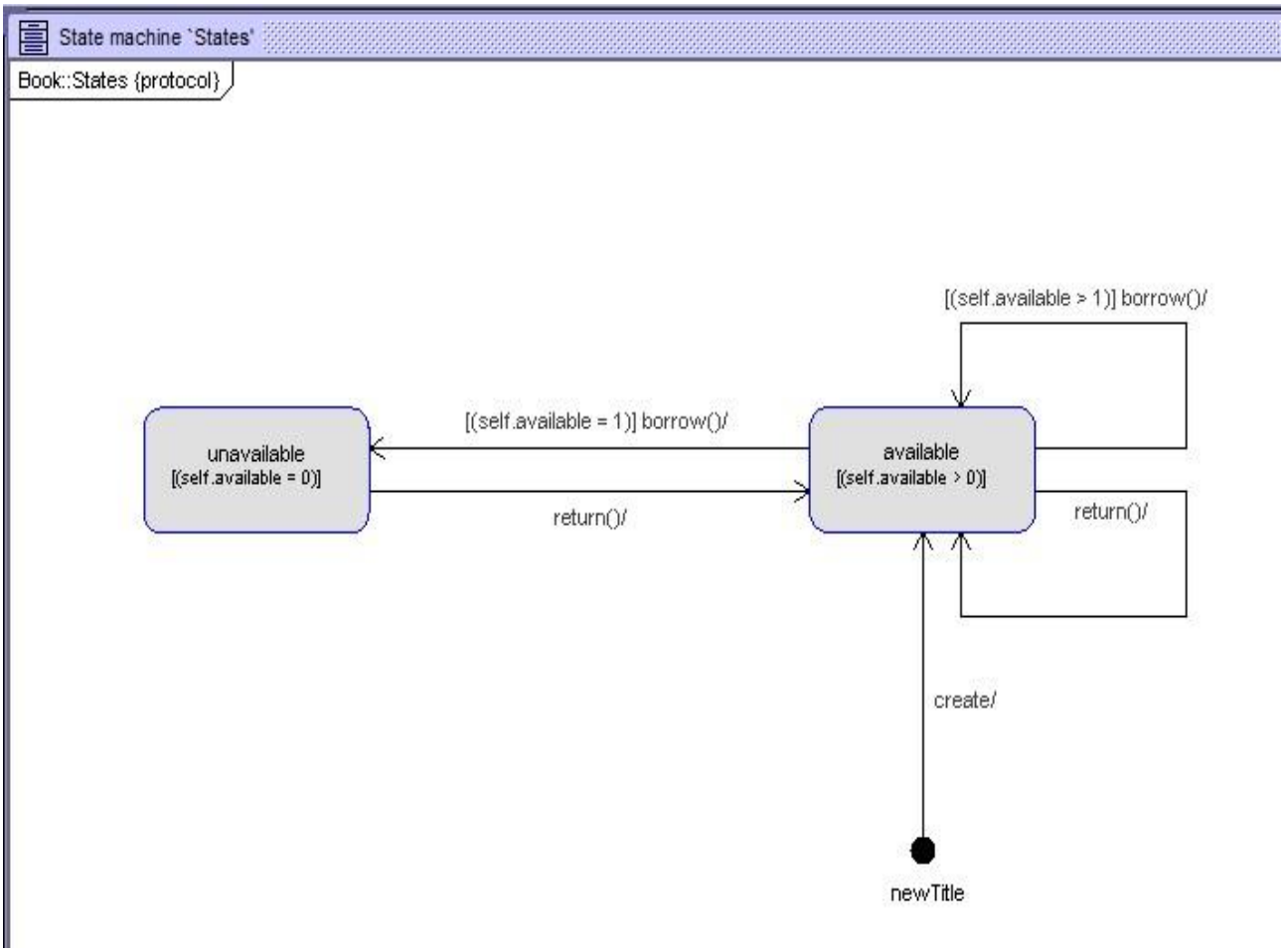


Sam Employee For Reserving & Applying Fine



STATE MACHINE

Book Available And Book Unavailable



USE Code For State Machine

```
        statemachines
            psm States
            states
                newTitle: initial
                available[available > 0]
                unavailable[available = 0]
            transitions
                newTitle -> available { create }
                available -> unavailable { [available = 1] borrow() }
                available -> available { [available > 1] borrow() }
                available -> available { return() }
                unavailable -> available { return() }
        end
end
```

CONSTRAINTS

```
constraints

context Person::borrow(c: Copy)
    pre amountBorrowedLimit: self.amountBorrowed < self.limit
    pre notAlreadyBorrowed: self.borrowed -> excludes(c)
    pre notSameBook: self.borrowed.book -> excludes(c.book)
    pre noOnLoanLimit: self.no_onloan < 2

context Copy::borrow(p: Person)
    pre notAlreadyBorrowed: self.borrowed = #NotBorrowed

context Book::borrow()
    post sufficientAvailable: self.available >= 0

context Person::return(c: Copy)
    pre borrowedCopy: self.borrowed -> includes(c)
    post notBorrowedCopy: self.borrowed -> excludes(c)

context Person::payFine(amount: Integer)
    pre hasFine: self.fine > 0
    post fineNotNegative: self.fine >= 0

context Person::reserve(c: Copy)
    pre noReservation: c.reservation -> isEmpty()

context Copy::reserve(p: Person)
    pre notReserved: self.reserved = #NotReserved
    pre notAlreadyBorrowed: self.borrowed = #NotBorrowed

context Person::removeReservation(c: Copy)
    pre hasReservation: c.reservation -> includes(self)
    post noReservation: c.reservation -> isEmpty()

context Employee::applyFine(p: Person, amount: Integer)
    pre fineNotExceedLimit: p.fine < 50
    post fineNotExceedLimit2: p.fine < 50
```

CONSTRAINTS TESTING

1. The Book's Copy Has Already Been Reserved

```
use> !Sam.reserve(Copy2)
This copy has been reserved for you
use> !Sam.reserve(Copy2)
[Error] 1 precondition in operation call `Person::reserve(self:Sam, c:Copy2)' does not hold:
  noReservation: c.reservation->isEmpty
    c : Copy = Copy2
    c.reservation : Set(Person) = Set{Sam}
    c.reservation->isEmpty : Boolean = false

  call stack at the time of evaluation:
    1. Person::reserve(self:Sam, c:Copy2) [caller: Sam.reserve(Copy2)@<input>:1:0]

+-----------------------------------------------------------------+
| Evaluation is paused. You may inspect, but not modify the state. |
+-----------------------------------------------------------------+

Currently only commands starting with `?', `:', `help' or `info' are allowed.
`c' continues the evaluation (i.e. unwinds the stack).

>
```

2. There Is No Reservation To Be Removed

```
C:\WINDOWS\system32\cmd.exe
use> !Bella.removeReservation(Copy4)
[Error] 1 precondition in operation call `Person::removeReservation(self:Bella, c:Copy4)' does not hold:
  HasReservation: c.reservation->includes(self)
    c : Copy = Copy4
    c.reservation : Set(Person) = Set{}
    self : Member = Bella
    c.reservation->includes(self) : Boolean = false

  call stack at the time of evaluation:
    1. Person::removeReservation(self:Bella, c:Copy4) [caller: Bella.removeReservation(Copy4)@<input>:1:0]

+-----------------------------------------------------------------+
| Evaluation is paused. You may inspect, but not modify the state. |
+-----------------------------------------------------------------+

Currently only commands starting with `?', `:', `help' or `info' are allowed.
`c' continues the evaluation (i.e. unwinds the stack).

>
```

3. It Is Not Possible To Return A Book That Was Never Borrowed

```
use> !Danny.return(Copy6)
[Error] 1 precondition in operation call `Person::return(self:Danny, c:Copy6)' does not hold:
  borrowedCopy: self.borrowed->includes(c)
    self : Member = Danny
    self.borrowed : Set(Copy) = Set{Copy1,Copy5}
    c : Copy = Copy6
    self.borrowed->includes(c) : Boolean = false

  call stack at the time of evaluation:
    1. Person::return(self:Danny, c:Copy6) [caller: Danny.return(Copy6)@<input>:1:0]

+-----------------------------------------------------------------+
| Evaluation is paused. You may inspect, but not modify the state. |
+-----------------------------------------------------------------+

Currently only commands starting with `?', `:', `help' or `info' are allowed.
`c' continues the evaluation (i.e. unwinds the stack).

>
```

4. The Copy Of The Book Has Already Been Borrowed

```
use> !Danny.borrow(Copy1)
[Error] 2 preconditions in operation call `Person::borrow(self:Danny, c:Copy1)' do not hold:
  notAlreadyBorrowed: not self.borrowed->includes(c)
    self : Member = Danny
    self.borrowed : Set(Copy) = Set{Copy1,Copy5}
    c : Copy = Copy1
    self.borrowed->includes(c) : Boolean = true
    not self.borrowed->includes(c) : Boolean = false

  notSameBook: self.borrowed->collect($e : Copy | $e.book)->excludes(c.book)
    self : Member = Danny
    self.borrowed : Set(Copy) = Set{Copy1,Copy5}
    $e : Copy = Copy5
    $e.book : Book = ThePhysician
    $e : Copy = Copy1
    $e.book : Book = Naruto
    self.borrowed->collect($e : Copy | $e.book) : Bag(Book) = Bag{Naruto,ThePhysician}
    c : Copy = Copy1
    c.book : Book = Naruto
    self.borrowed->collect($e : Copy | $e.book)->excludes(c.book) : Boolean = false

  call stack at the time of evaluation:
    1. Person::borrow(self:Danny, c:Copy1) [caller: Danny.borrow(Copy1)@<input>:1:0]

+-----------------------------------------------------------------+
| Evaluation is paused. You may inspect, but not modify the state. |
+-----------------------------------------------------------------+

Currently only commands starting with `?', `:', `help' or `info' are allowed.
`c' continues the evaluation (i.e. unwinds the stack).

>
```

5. Can't Overpay A Fine

```
use> !Sam.applyFine(Bella,30)
use> !openter Bella payFine(40)
precondition `hasFine' is true
use> !Bella.fine := Bella.fine -40
use> !opexit
postcondition `fineNotNegative' is false
  self : Member = Bella
  self.fine : Integer = -10
  0 : Integer = 0
  (self.fine >= 0) : Boolean = false
Error: postcondition false in operation call `Person::payFine(self:Bella, amount:40)'.
use>
```

6. Can't Apply A Fine That Exceeds The Limit

```
use> !Sam.applyFine(Bella,100)
Fine amount exceeds limit of 50
use>
```

```
use> !Sam.applyFine(Bella,200)
[Error] 1 postcondition in operation call `Employee::applyFine(self:Sam, p:Bella, amount:200)' does not hold:
  fineNotExceedLimit2: (p.fine < 50)
    p : Member = Bella
    p.fine : Integer = 200
    50 : Integer = 50
    (p.fine < 50) : Boolean = false

  call stack at the time of evaluation:
    1. Employee::applyFine(self:Sam, p:Bella, amount:200) [caller: Sam.applyFine(Bella, 200)@<input>:1:0]

+-------------------------------------------------------------+
| Evaluation is paused. You may inspect, but not modify the state. |
+-------------------------------------------------------------+

Currently only commands starting with `?', `:', `help' or `info' are allowed.
`c' continues the evaluation (i.e. unwinds the stack).

>
```