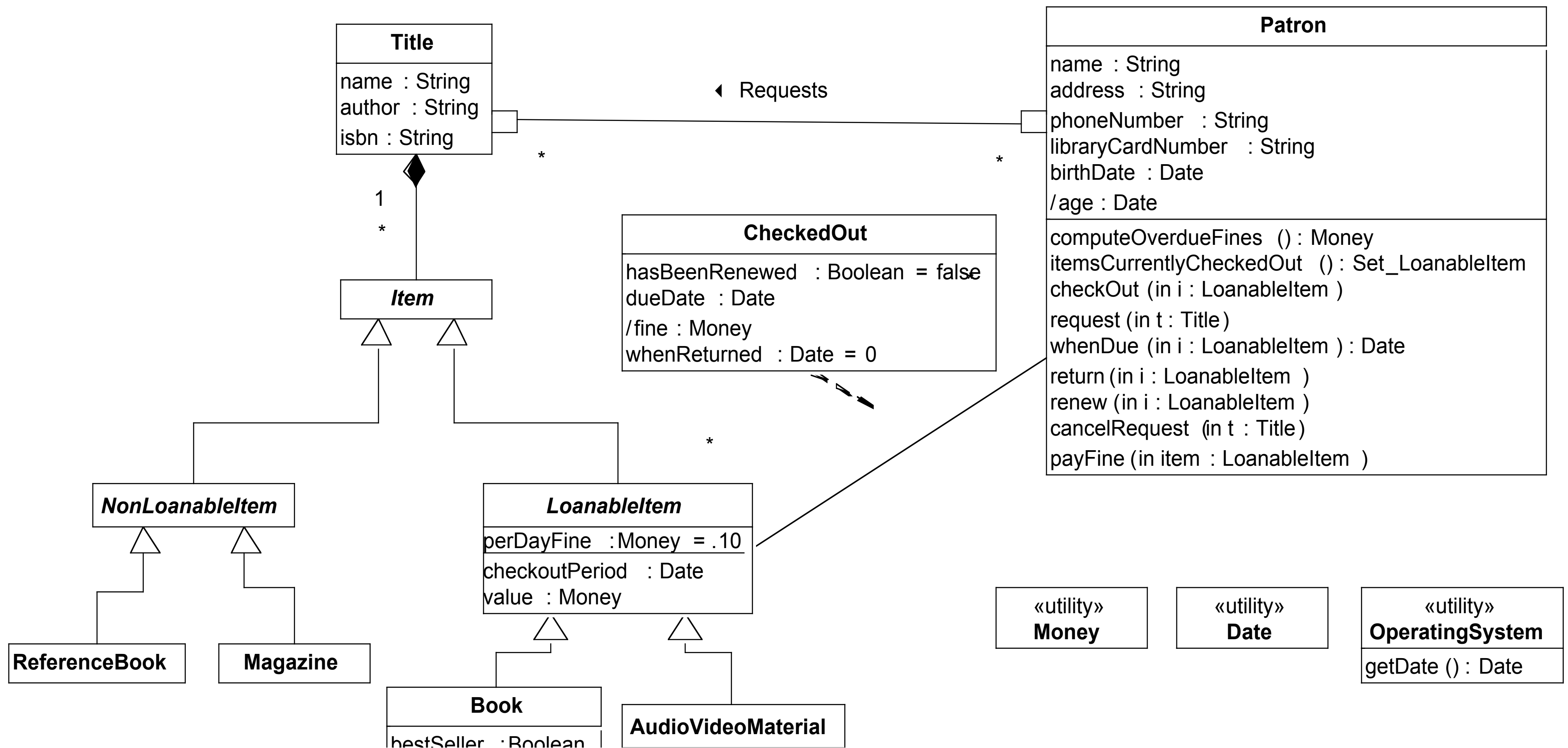
Motivation

- With the Library Problem, we saw that we could use a UML class model diagram to conceptualize the main participants in a problem and their relationships
- The modeling process actually helped us improve the requirements by raising various questions
- But there were significant limits on what could be expressed using only the diagram
- We will now see how OCL annotations can be used to address these limitations

Class Model Diagram



Library Problem Requirements

1. Each patron has one unique library card for as long as they are in the system.
2. The library needs to know at least the name, address, phone number, and library card number for each patron.
3. In addition, at any particular point in time, the library may need to know or to calculate the items a patron has checked out, when they are due, and any outstanding overdue fines.
4. Children (age 12 and under) have a special restriction–they can only check out five items at a time.
5. A patron can check out books or audio/video materials.
6. Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks.
7. A/V materials may be checked out for two weeks.
8. The overdue fine is ten cents per item per day, but cannot it go higher than the value of the overdue item.
9. The library also has reference books and magazines, which cannot be checked out
10. A patron can request a book or A/V item that is not currently in.
11. A patron can renew an item exactly once–unless there is a outstanding request for the item, in which case the patron must return it.

Diagram

- Which of these requirements did the diagram adequately address?

Requirements that are Satisfied by the Diagram itself

- The diagram was able to adequately express some requirements for example, numbers 2 and 9

2. *The library needs to know at least the name, address, phone number, and library card number for each patron.*
9. *The library also has reference books and magazines, which cannot be checked out*

Limitations

- For all of the other requirements, however, either the specification as given by the diagram must be supplemented or an entirely new specification must be introduced
- We will begin by looking at Requirement 6, which can be expressed by an invariant constraint local to the `Book` class

Requirement 6

Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks

- How would we specify this requirement using OCL?

Requirement 6

Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks

- How would we specify this requirement using OCL?
- First ask: Where should the constraint go?

Requirement 6

Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks

- How would we specify this requirement using OCL?
- First ask: Where should the constraint go?
  - Answer: Class `Book`

Requirement 6

Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks

- How would we specify this requirement using OCL?
- First ask: Where should the constraint go?
  - Answer: Class `Book`
- Then ask: Should the constraint be an invariant on the class or a pre/post condition pair on an operation?

Requirement 6

Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks

- How would we specify this requirement using OCL?
- First ask: Where should the constraint go?
  - Answer: Class `Book`
- Then ask: Should the constraint be an invariant on the class or a pre/post condition pair on an operation?
  - The most direct solution is to use a class invariant
  - We can also solve the problem with pre and post condition constraints on the operation `checkout`
  - The former is more powerful because it holds over all possible operations in the model, not just `checkout`

Requirement 6

Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks

- How would we specify this requirement using OCL?
- First ask: Where should the constraint go?
  - Answer: Class `Book`
- Then ask: Should the constraint be an invariant on the class or a pre/post condition pair on an operation?
  - The most direct solution is to use a class invariant
  - We can also solve the problem with pre and post condition constraints on the operation `checkout`
  - The former is more powerful because it holds over all possible operations in the model, not just `checkout`
- Now we can actually express the invariant

Requirement 6

Books are checked out for three weeks, unless they are current best sellers, in which case the limit is two weeks

```
context Book inv:
  if bestSeller then
    checkoutPeriod = 2 -- weeks
  else
    checkoutPeriod = 3
  endif
```

Requirement 7

A/V materials may be checked out for two weeks

- This requirement is handled similarly to Requirement 6

```
context ?? inv:
  ??
```

Requirement 7

A/V materials may be checked out for two weeks

- This requirement is handled similarly to Requirement 6

```
context AudioVideoMaterial inv:
  checkoutPeriod = 2
```

Operations

- Now let's look at how to specify a constraint on an operation
- We will first look at a query operation that has no side effects, that is, it only returns the value of an attribute (or possibly some combination of such values)
- Part of Requirement 3 states: *"... the library may need to know or to calculate the items a patron has checked out ... "*
- The word "calculate" tips us off that an operation is involved

`itemsCurrentlyCheckedOut`

- We first ask: What is the context?

`itemsCurrentlyCheckedOut`

- We first ask: What is the context?
  - For operations, the context is always the operation itself, with the operation's class prepended, the argument names and their types and the type of the return value, if any

itemsCurrentlyCheckedOut

- We first ask what is the context
  - For operations, the context is always the operation itself, with the operation's class prepended, the argument names and their types specified and the type of the return value
  - That is, the operation's signature

```
context
  Patron::itemsCurrentlyCheckedOut() :
      Set(LoanableItem)
```

`itemsCurrentlyCheckedOut`

- For operations, the next question typically is: What are the preconditions, the circumstances under which it is meaningful for the operation to execute?

`itemsCurrentlyCheckedOut`

- For operations, the next question typically is: What are the preconditions, the circumstances under which it is meaningful for the operation to execute?
  - In the case of `itemsCurrentlyCheckedOut`, as with most operations that provide a value without effecting a state change, there are no preconditions
  - We can either have a precondition whose expression is **`true`**, or we can omit the precondition entirely

`itemsCurrentlyCheckedOut`

- Finally, we need to specify what value is returned
- To compute the `Items` that are currently checked out, we merely navigate along the `checkedOut` association to the corresponding set of `LoanableItems`

```
context
  Patron::itemsCurrentlyCheckedOut() :
      Set(LoanableItem)
post:
  result = checkedOut.LoanableItem
```

Requirement 4

Children (age 12 and under) have a special restriction–they can only check out five items at a time.

- Here's a slightly more complex situation, having to do with an attempt to check out an `Item`. How would we make sure that the limit for children is adhered to?
- Note that this is not a complete specification of `checkOut`, only the part having to do with children
  - Other pre and post conditions will have to be conjoined

Requirement 4

Children (age 12 and under) have a special restriction–they can only check out five items at a time.

- Here's a slightly more complex situation, having to do with an attempt to check out an `Item`. How would we make sure that the limit for children is adhered to?
- Note that this is not a complete specification of `checkOut`, only the part having to do with children
  - Other pre and post conditions will have to be conjoined

```
context Patron::checkOut(i : LoanableItem)
pre: age <= 12 implies
   itemsCurrentlyCheckedOut()->size() < 5
```

Side Effects

- Now let's try specify an even more complex situation, one where an operation actually results in a change of state
- We choose to model the actual process of checking out a `LoanableItem`

Requirement 5

A patron can check out books or audio/video materials.

- The context is obviously the `checkOut` operation in `Patron`
- The argument is a `LoanableItem`
- And there is no value returned

```
context Patron::checkOut(i : LoanableItem)
```

Requirement 5

A patron can check out books or audio/video materials.

- But the preconditions are not so obvious; how many you can name?

Requirement 5

A patron can check out books or audio/video materials.

- But the preconditions are not so obvious; how many you can name?
  - The `Item` must be available
  - The `Patron` must not be a child who already has five `Items` checked out
  - The `Item` must not have been requested by someone other than the `Patron`
  - [The `Patron` has no outstanding fines]

Requirement 5

A patron can check out books or audio/video materials.

```
context Patron::checkOut(i : LoanableItem)
pre:
  i.loanable() and
  (age <= 12 implies
      itemsCurrentlyCheckedOut()->size() < 5) and
  (i.Title.Requests->size() = 0 or
      (i.Title.Requests->size() = 1 and
          i.Title.Requests.Patron = self)) and
  forAll(c : CheckedOut | c.fine = 0)
```

Requirement 5

A patron can check out books or audio/video materials.

- Now we have to come up with the postconditions (effects) of checking out an `Item`
- See how many you can state

Requirement 5

A patron can check out books or audio/video materials.

- Now we have to come up with the postconditions (effects) of checking out an `Item`
- See how many you can state
  - The `Item` is no longer available
  - There is a new `CheckedOut` link
    - With the appropriate `dueDate` and other attributes
  - If the user was a requester of that `Item`, then the `Request` link must be removed

Requirement 5

A patron can check out books or audio/video materials.

```
post:
  exists(c : CheckedOut | c.loanableItem = i and
    c.dueDate = operatingSystem.getDate() +
      c.loanableItem.checkoutPeriod and
    checkedOut =
      checkedOut@pre->including(c))
```

Requirement 5

A patron can check out books or audio/video materials.

```
context Patron::checkOut(i : LoanableItem)
post:
  let t : Title = i.title in
    if t.requests->includes(self)
    then
      requests =
        requests@pre->reject(title = t)
    else
      true
    endif
```

Requirement 1

Each patron has one unique library card for as long as they are in the system.

- There are built-in features of OCL that support this kind of situation
  - `allInstances` return the set of instances of a class
  - `isUnique` checks the values of a specified attribute

```
context Patron inv:
   Patron.allInstances()->isUnique(libraryCardNumber)
```

Requirement 10

A patron can request a book or A/V item that is not currently in.

- This specification makes use of a (not yet defined) utility operation called `loanable` in class `LoanableItem` to represent the concept "currently in the library and available to be checked out"

```
context Patron::request(t : Title)
pre:
  t.LoanableItem->size() > 0 and
  t.LoanableItem->forAll(i | ! i.loanable()) and
  ! self.Requests->exists(r | r.Title = t)
post:
  exists(r : Request |
    r.patron = self and r.title = t and
    requests = requests@pre->including(r))
```

Requirement 3

In addition, at any particular point in time, the library may need to know or to calculate the items a patron has checked out, when they are due, and any outstanding overdue fines.

```
-- dueDate for checked out Items not yet returned
context Patron::whenDue(i : LoanableItem) : Date
post:
  result = CheckedOut->select(c | c.LoanableItem = i and
  c.whenReturned <> 0).dueDate
```

```
context Patron::computeOverDueFines() : Money
post: CheckedOut->iterate(c; s : Integer = 0
      | s + c.fine)
```

Requirement 11

A patron can renew an item exactly once–unless there is a outstanding request for the item, in which case the patron must return it.

```
context Patron::renew(i : LoanableItem)
pre:
  i.Title.Requests->isEmpty() and
  checkedOut.loanableItem->includes(i) and
  CheckedOut->exists(c | c.LoanableItem = i
    c.whenReturned <> 0 and not c.hasBeenRenewed)
post:
  CheckedOut->exists(c | c.LoanableItem = i and
    and c.whenReturned = 0 and
     c.dueDate = i.dueDate + i.CheckOutPeriod and
     c.hasBeenRenewed)
```

Derived Data

- We have seen how to specify invariants and operations
- There is another obligation that the specifier must discharge: specify any derived data

Requirement 8

The overdue fine is ten cents per item per day, but cannot it go higher than the value of the overdue item.

```
context CheckedOut
derive:
  let daysOverDue : Date =
    if whenReturned = 0 then
      OperatingSystem.getDate() - dueDate
    else
      whenReturned - dueDate
    endif
  in
    let nominalCharge : Money =
      daysOverDue + LoanableItem.perDayFine
    in
      fine = if daysOverDue > 0 then
                nominalCharge.min(LoanableItem.value)
             else 0 endif
```

`age`

- `age` is a derived attribute in `Patron`. Its value must be expressed in terms of other values using a constraint

```
context Patron inv:
  age = OperatingSystem.getDate() - birthDate
```

- This assumes that the subtraction operation has been defined for `Dates`

Auxiliary Operation

- UML allows the definition of auxiliary functions that are not directly mentioned in the requirements
  - Such functions may be useful in avoiding tedious duplication
  - Auxiliary operations may query values but must not cause any side effects
- The specifier must provide a definition for each auxiliary operation

Returning an `Item`

- Returning is actually somewhat subtle
- First off, a `Patron` can only return an `Item` if that `Patron` has checked out that `Item`
- If a fine is due, then the link in `CheckedOut` cannot be removed until the fine is paid

```
context Patron::return(i : LoanableItem)
pre:
  self.CheckedOut.LoanableItem->includes(i)
post:
  let l : CheckedOut = self.CheckedOut->
      select(Item = i and whenReturned = 0)
  in
      l.whenReturned = OperatingSystem.getDate() and
      l.fine = 0 =>
        CheckedOut = CheckedOut@pre->excluding(l)
```

Paying Fines

- A `Patron` can pay a fine only if the `Item` has been returned
- The `Patron` can only pay `fines` on `Items` they have checked out
- We also require that a `fine` must be paid in full

```
context Patron::payFine(i : LoanableItem, m : Money)
  pre:
    LoanableItem->includes(i) and
    let
      r = CheckedOut->any(LoanableItem = i)
    in r.whenReturned > 0 and r.fine = m
  post:
    let
      r = CheckedOut->any(LoanableItem = i)
    in
      checkedOut = checkedOut@pre->excluding(r)
```

Canceling a `Request`

- This might happen because a patron decided he/ she no longer wanted to borrow the item, or because of a length limit on outstanding requests or because the request is being satisfied
- Note that the precondition below is not really needed

```
context Patron::cancelRequest(t : Title)
pre:
  self.Requests->exists(r | r.Title = t)
post:
  ! self.Requests->exists(r | r.Title = t)
```

loanable

- This utility operation indicates whether or not an `Item` may be checked out
- There are two possibilities
  - There is no `CheckedOut` link existing for this `Item`
  - There is such a link, but the `Item` has been returned and is waiting for its fine to be paid

```
context LoanableItem::loanable() : Boolean
pre: true
post: result = self.CheckedOut->isEmpty() or
       (self.CheckedOut->notEmpty() and
        self.CheckedOut->exists(whenReturned = 0))
```

Best Sellers

- If a `Book` is a best seller, then all copies of the `Book` must be best sellers

```
context b1 : Book inv:
  forall(b2 : Book |
    (b1.bestSeller and
    b1.title = b2.title)
    implies b2.bestSeller)
```

Conclusions

- Notice that the process of modeling caused us to realize that there were possible situations that we had not considered
  - E.g. an overdue Book returned without the fine being paid
- Many new operations were suggested
- There may be more than one correct answer

Unresolved Issues

- The preceding constraints on operations were actually incomplete
  - They indicated what you should expect from an operation, but they didn't include constraints saying that nothing else should have changed!
  - These kinds of constraints, which would have to be added, are called *frame* conditions
- The class model diagram and the OCL did not include constraints specifying operations in the utility classes `Money` and `Date`
- Neither did they indicate how the classes of `Item` and `Patron` have instances added to them

Outstanding Problems

- `{frozen}` should be used for `perDayFine`
- Currently, a request for a `Book` can be satisfied with `AudioVideoMaterial`. This might be handled by adding an extra enumeration argument to `Patron::request()` that gets stored as an attribute of the `Request` association class
- Zero has meaning as a `dueDate` but is not a meaningful `Date`
- No `Title` has both a `Checkoutable` and a `Non-Checkoutable Items`