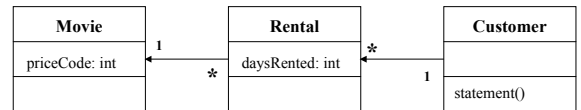


Case Study

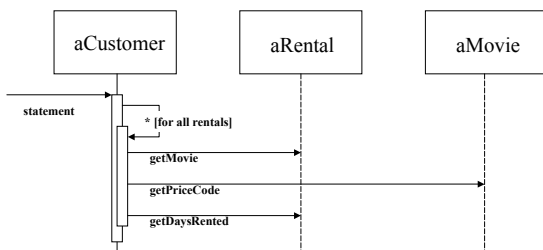
- Software to support a video store
- Track rentals; print reports
- Refactor to improve design

- (code handouts given in class)

Class Diagram



Interaction Diagram



Design Goals

- Abstraction (information hiding)
- Flexibility (not the same as generality)
- Clarity
- Irredundancy ("thrifty" code)

Refactoring 1

Decompose and Redistribute Statement Method

- Bad smell: long method
- Refactoring: extract method
 - Handle local variables
 - Unmodified (`each`) passed in as a parameter
 - Modified (`thisAmount`) returned as result

Refactoring 2

Rename Variables

- Provide more descriptive names
- Avoid name conflicts to assure correctness

Guidelines

- Refactor while reading to gain understanding
- Try to eliminate the need for comments
- Try to eliminate the need for local variables in methods

Refactoring 3

Move Method

- `amountFor` is in `Customer`, but it doesn't use any `Customer` data
 - Move it to `Rental` (3a)
 - Give it a new name
 - Remove references to `aRental` from message invocations
 - Change method invocations to use new name (3b)
- Test after each small change

Refactoring 4

Replace Temp with Query

- Local variables that are only set once in a method act like constants
- Replace them with calls to the defining method
- This assumes the method call has no side effect
- May reduce performance
- Improves understandability
- Makes other refactorings easier

Refactoring 5

Extract FrequentRenterPoints Computation

- Further simplify `statement` computation
- New method (`getFrequentRenterPoints`) in `Rental`
- Some algebraic simplification as well
 - Removes side effects
 - Removes a local variable

Refactoring and Enhancements

- Refactorings leave functionality unchanged
 - This makes testing easier
- But sometimes refactorings can be used to prepare for an enhancement
 - For example, introduce an abstract class that will be subclassed as part of a new feature

Possible Enhancements to the Video Store Program

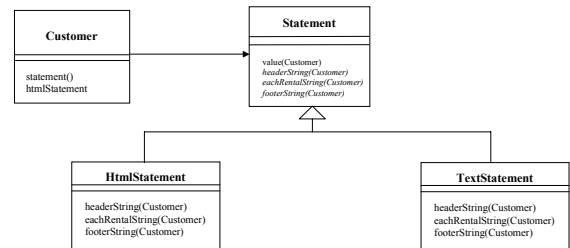
- Altered charging rules
- New classes of movies
- HTML version of statement

Interlude 6

Prepare to add `htmlStatement`

- One way in which this program might be extended is to add a second output format
- `htmlStatement` is a new method for formatting output in HTML instead of text
- It should not be concerned with the rules for computing charges, just with how to format
- Consequently, statements for computing charges should be factored into methods, even if this means duplicating loop code
- These methods will be callable from both `statement` and `htmlStatement`

Class Diagram



Refactoring 7: Steps

- Define a new `Statement` class with `TextStatement` and `HtmlStatement` subclasses
- Rename `statement` method to `value` within the `Statement` classes and pass a `Customer` as argument
- Add `getRentals` method to return an Enumeration of current `Rentals`
- Relax visibility of `getTotalCharge` and `getTotalFrequentRenterPoints`

Refactoring 7: Steps - 2

- Prepare methods for all format-specific functionality and replace existing code with calls to the new methods
- Remaining calling method can be pulled up into `Statement` class
- Declare abstract classes for format-specific methods

Refactoring 8:

Replace Conditional Logic on `priceCode` with Polymorphism

- `switch` statements (or `else ifs`) are strong indicators of poor object-oriented coding practices
- They should be replaced by subclasses and polymorphic methods

Refactoring 8: Steps

- First, note that the `switch` statement is discriminating on the type of the movie, so it should really be in the `Movie` class
- Do the same with the `getFrequentRenterPoints` method
- Now the way is prepared for subclassing `Movie` objects into `Regular`, `Childrens`, and `NewRelease` movies

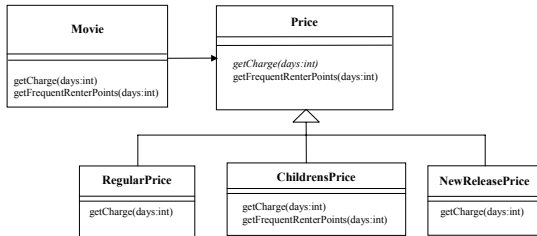
Refactoring 8: Steps - 2

- Unfortunately, this doesn't work. Can you see why?
- Movies in video stores can change their categories dynamically as, for example, when a `New Release` moves into the `Regular` category
- But an object can't change its class dynamically

Refactoring 8: Steps - 3

- There is a trick (called the *state* pattern) for overcoming this problem. It involves introducing a class solely for holding type information
- We will use the `Price` class with subclasses for `ChildrensPrice`, `NewReleasePrice`, and `RegularPrice`
- These classes provide methods for `getCharge` and `getFrequentRenterPoints`

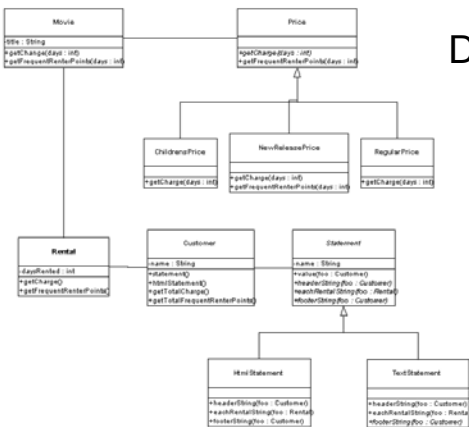
Class Diagram



Refactoring 8: Steps - 4

- Move type code behavior into `Price` class
- Move `switch` statement into `Price` class
- Replace `switch` statement with polymorphism
- Update `Movie` class to reflect changes

Class Diagram



Observations

- Ended up with many small methods
- Largely self documenting
- Classes correspond to concepts
- Changes made incrementally with frequent testing