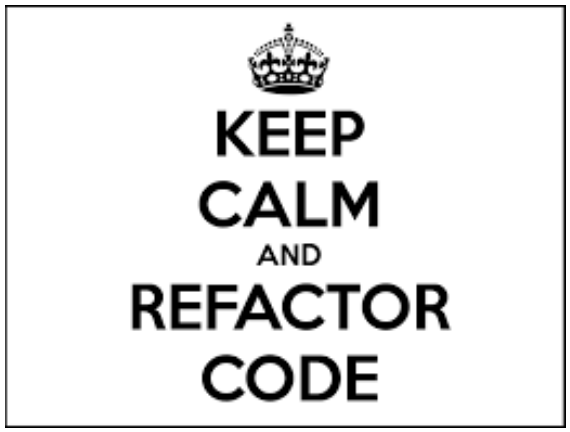


Refactoring

Many thanks to Profs. Milanova, Goldschmidt, and Michael Ernst, U. Wash.)

Refactoring

- Introduction to Refactoring
  - Programming is just refactoring a blank screen
    - ~ unknown source

- Refactorings
  - Extract method, Move method
  - Replace temp with query
  - Replace type code with State/Strategy
  - Replace conditional with polymorphism
  - Form Template Method
  - Replace magic number with symbolic constant

We'll learn more about the State, Strategy and Template Method design patterns

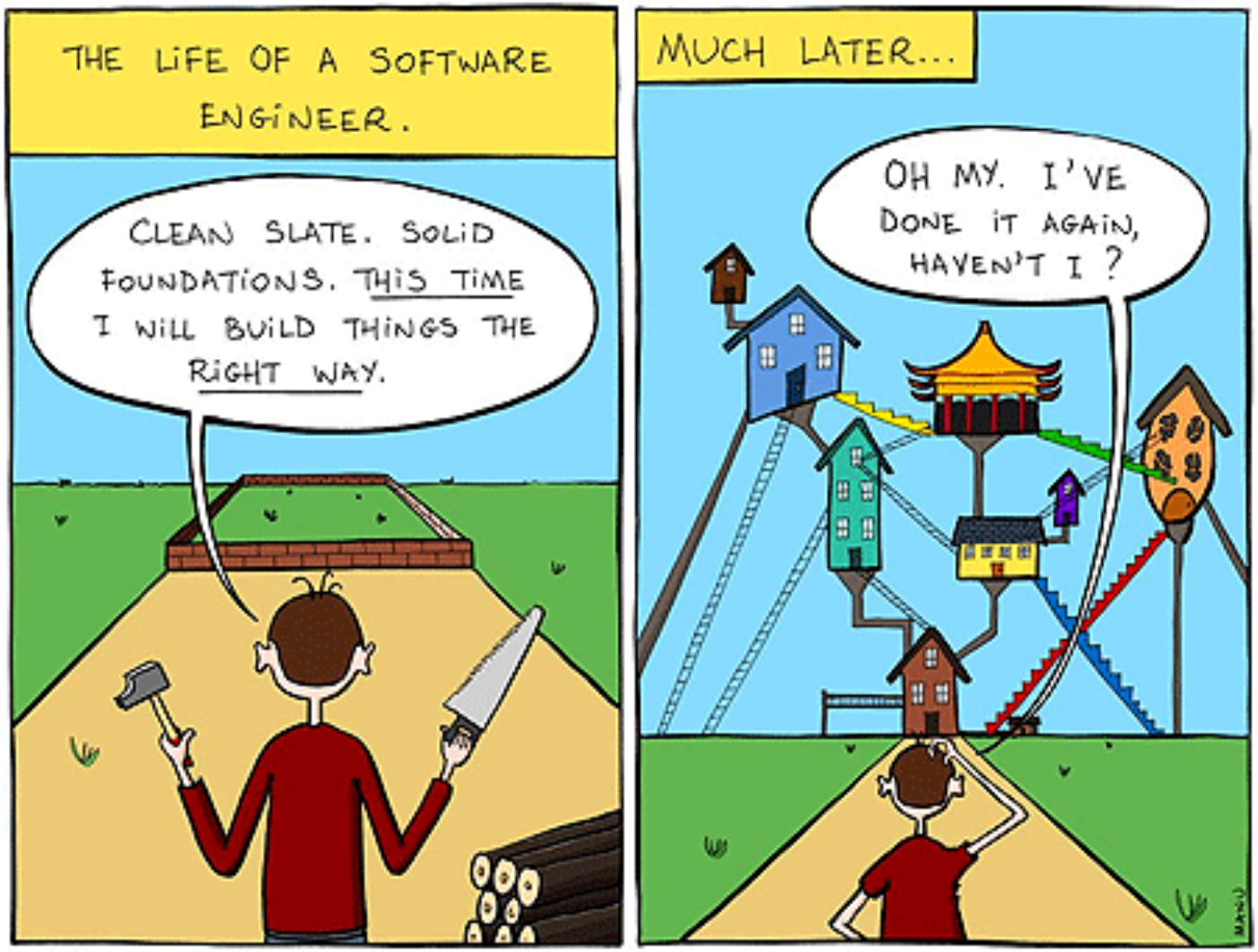
So Far

- We studied techniques for writing correct and maintainable code
  - Correctness: careful planning, specifications, reasoning about code, testing
  - Understandability and maintainability: Design patterns promote low coupling and “open/closed” designs
    - designs that are “open for extension but closed to modification”

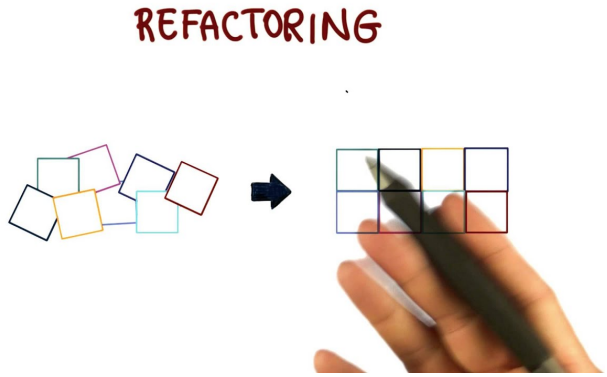
So Far

- How to design your code
  - The hard way: Start hacking. Hack some more…
    - Seems like the easy way
  - The easier way: Plan carefully
- How to verify your code
  - The hard way: Make up some inputs…
  - An easier way: systematic testing and reasoning
- The hard way leads down the dark path

- But we do get down the dark path



https://usercontent2.hubstatic.com/8530569.gif



Refactoring

- Premise: we have written complex, possibly ugly, code, but it works! Can we simplify this code?
  - Two opposite tendencies
    - "Don't fix it, if it's not broken."
    - When it breaks, it can be a real mess
- Refactoring: disciplined rewrite of code
  - Small-step behavior-preserving transformations
  - Followed by execution of test cases
    - Depends on having a good suite of tests
- Continuous refactoring combined with testing is an essential software development practice

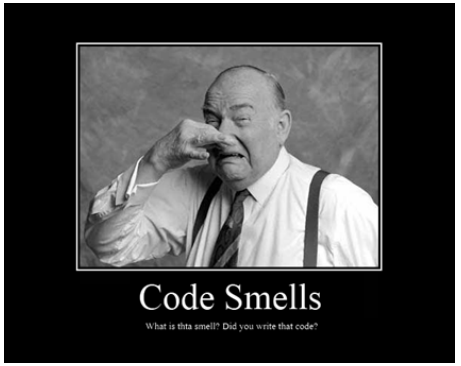
Technical Debt

- **Technical debt** reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution.
- Refactoring helps prevent accumulation of technical debt
- Caused by
  - Business pressures
  - Lack of understanding
  - Tightly coupled modules
  - Not enough/improper testing
  - Lack of documentation
  - Lack of collaboration
  - Etc.
- AntiPatterns are a source of technical debt

Refactoring

- Refactorings attack code smells
- **Code smells/AntiPatterns –** bad coding practices leading to technical debt
- Code Smells
  - any characteristic in the source code of a program that possibly indicates a deeper problem.
  - E.g., big method
  - An oversized “God” class
  - Similar methods, classes or subclasses
  - Little or no use of subtype polymorphism
  - High coupling between objects,
  - Etc.

Code Smells



Smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality.

Code smells are not necessarily bugs

- not necessarily technically incorrect
- indicate possible weakness in code
- indicator of possible technical debt
- similar to but not always antipatterns

Examples

Duplicated code: identical or very similar code exists in more than one location.

Contrived complexity: forced usage of overcomplicated design patterns.

Shotgun surgery : a single change needs to be applied to multiple classes at the same time.

More Code Smells

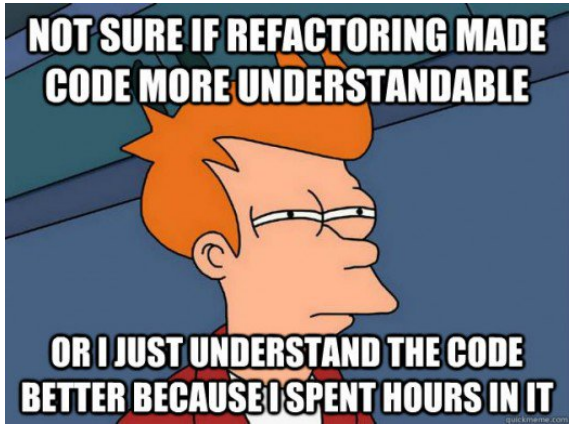
Large class: a class that has grown too large. God object.
Feature envy: a class that uses methods of another class excessively.
Inappropriate intimacy: a class that has dependencies on implementation details of another class.
Refused bequest: a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class. See Liskov substitution principle.
Lazy class / freeloader: a class that does too little.
Excessive use of literals: these should be coded as named constants
*Cyclomatic comple*xity: too many branches or loops
Downcasting: a type cast which breaks the abstraction model
Orphan variable or constant class: a class that typically has a collection of constants which belong elsewhere
Data clump: Occurs when a group of variables are passed around together in various parts of the program. This suggests that it would be more appropriate to formally group the different variables together into a single object.

Refactoring Activities

- Make long methods shorter
- Remove duplicate code
- Introduce design patterns
- Remove the use of hard-coded constants
- Etc…

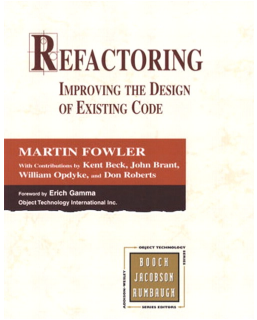
- Goal: achieve code that is short, tight, clear, and without duplication

Refactoring Activities



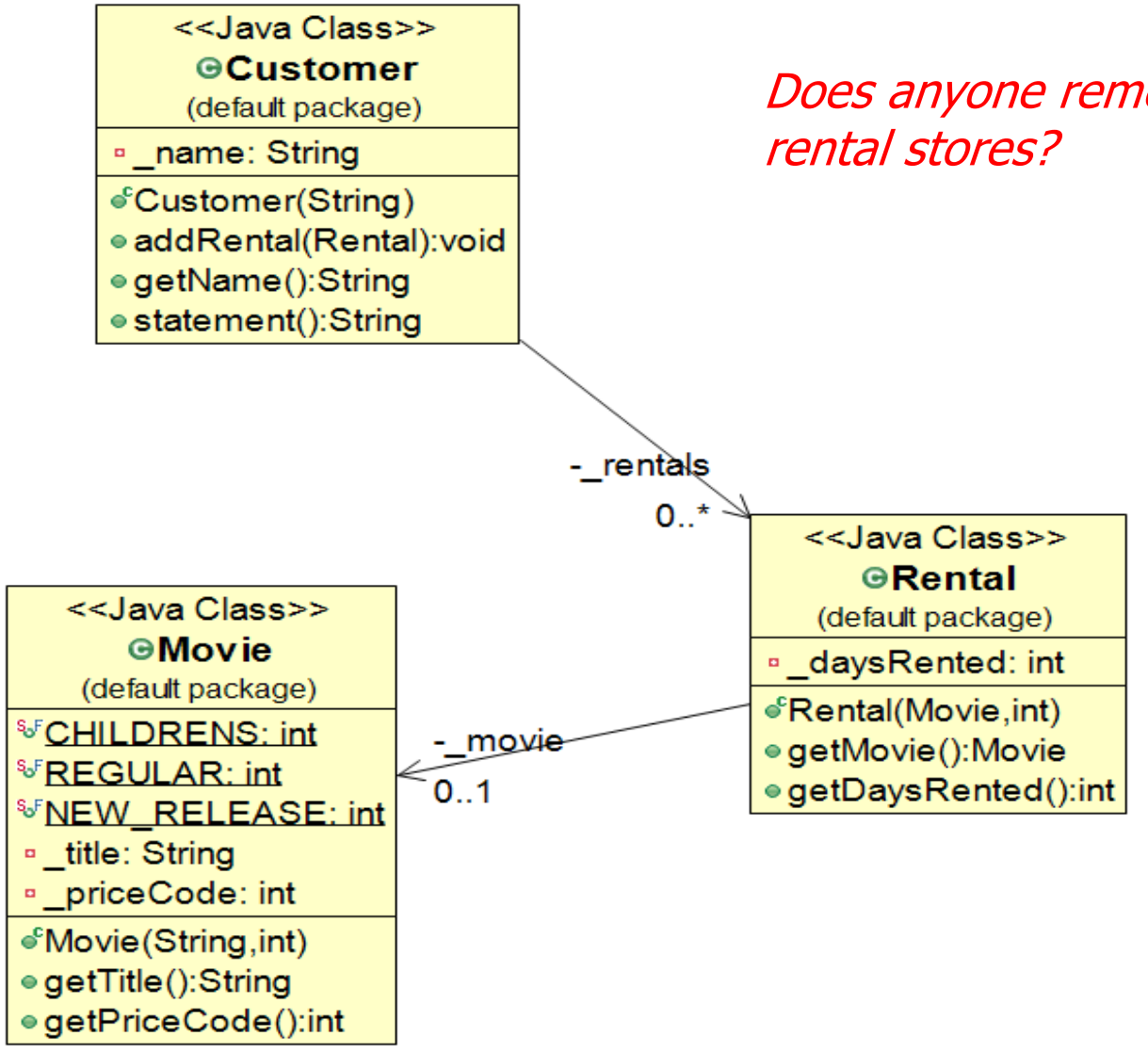
- Rule:
  - Make small disciplined changes
  - Test after each change

A Refactoring (noun)



- A refactoring is a named, documented algorithm that attacks a specific bad coding practice
  - E.g., Extract method, Move method, Replace constructor call with Factory method
  - Relatively well-defined mechanics, can be automated
  - Eclipse can automate some of these
- Canonical Reference:
- *Refactoring, Improving the Design of Existing Code* by Martin Fowler
  - Initial catalog of 72 refactorings, 1999
  - Currently, more than 100 documented refactorings

Movie Rentals (Fowler)



Does anyone remember movie rental stores?

https://martinfowler.com/articles/refactoring-video-store-js/#DecomposingIntoSeveralFunctions

Movie Rentals

```
public class Movie {
    // immutable class!
    public static final int CHILDRENS = 2;
    public static final int REGULAR = 0;
    public static final int NEW_RELEASE = 1;

    private String _title;
    private int _priceCode;

    public Movie(String title, int priceCode) {
            _title = title;
            _priceCode = priceCode;
    }

    public String getTitle() {
            return _title;
    }

    public int getPriceCode() {
            return _priceCode;
    }
}
```

Movie Rentals (Fowler, 1999)

```
public class  Rental {
    // an immutable class
    private Movie _movie;
    private int _daysRented;

    public Rental(Movie movie, int daysRented) {
            _movie = movie;
            _daysRented = daysRented;
    }

    public Movie getMovie() {
            return _movie;
    }

    public int getDaysRented() {
            return _daysRented;
    }
}
```

Class Customer

```
public class Customer {
    // mutable class
    private String _name;
    private List<Rental> _rentals;

    public Customer (String name) {
            _name = name;
            _rentals =  new ArrayList<Rental>();
      }

    public void addRental(Rental arg) {
            _rentals.add(arg);

     … // other methods
    }
```

Method statement() in Customer, composes a Customer statement

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Iterator<Rental> rentals = _rentals.iterators();
    String result =
       "Rental Record for " + getName()+"\n";
    while (rentals.hasNext()) {
       // process current rental
       double thisAmount = 0; // amount for this rental
       Rental each = rentals.next();

       // Next, compute amount due for current rental
(each)
```

Method statement()

```
switch (each.getMovie().getPriceCode()) {
  case Movie.REGULAR:
     thisAmount += 2;
     if (each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented()-2)*1.5;
     break;
  case Movie.NEW_RELEASE:
     thisAmount += each.getDaysRented()*3;
     break;

  case Movie.CHILDRENS:
     thisAmount += 1.5;
     if (each.getDaysRented() > 3)
       thisAmount += (each.getDaysRented()-3)*1.5;
      break;
} // end of switch statement
```

Method statement()

```
// add frequent renter points contributed by current rental
      frequentRenterPoints++;
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
          && each.getDaysRented() > 1)
        frequentRenterPoints++;
      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
   } // end of the while loop over the _rentals
     // add totalAmount and frequentRenterPoints to result
   return result; // Finally DONE!
 } // end of method statement()
} // end of Customer class
```

Discussion

- What code smells can you find?
- There are many…

- Code smells:
  - big method
  - strong coupling between objects
    - Uses _rentals, Movie.getMovie()
  - little or no use of subtype polymorphism
  - and so on

The Extract Method Refactoring

- Problem: Method **Customer.statement()** is too big, difficult to understand and maintain
- Solution: Find a logical chunk of code to be extracted out of **statement()**, and perform the **Extract** Method refactoring
- Key point: safety – refactoring preserves behavior. Test after every refactoring!

What part of statement() would you extract?

Extract Method, Mechanics

- Create a new method, name it appropriately
- Copy the extracted code in the new method
- Scan the extracted code for references to local variables
- If local variables are used only in extracted code, declare them in the new method
- See if any local variables are modified by the extracted code – tricky part…
  - Are they used in previous method?
- Pass as parameters local variables that are not modified but are only used by the extracted code
- Compile
- Replace the extracted code with a method call
- Compile and test!!!

Extract Method, extracted code

```
double thisAmount = 0; // thisAmount is local to while loop
switch (each.getMovie().getPriceCode()) {
  // each is local to while loop too
 case Movie.REGULAR:
      thisAmount +=2;
      if (each.getDaysRented() > 2)
        thisAmount += (each.getDaysRented()-2)*1.5;
      break;
 case Movie.NEW_RELEASE:
      thisAmount += each.getDaysRented()*3;
      break;

 case Movie.CHILDRENS:
      thisAmount += 1.5;
      if (each.getDaysRented()>3)
        thisAmount += (each.getDaysRented()-3)*1.5;
         break;
}
totalAmount += thisAmount;
```

Extract Method. The new method

```
private double amountFor(Rental each) {
    double thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
    case Movie.REGULAR:
          thisAmount +=2;
          if (each.getDaysRented() > 2)
              thisAmount += (each.getDaysRented()-2)*1.5;
          break;

    case Movie.NEW_RELEASE:
          thisAmount += each.getDaysRented()*3;
          break;

    case Movie.CHILDRENS:
          thisAmount += 1.5;
          if (each.getDaysRented()>3)
              thisAmount += (each.getDaysRented()-3)*1.5;
          break;
    }
    return thisAmount;
}
```

For readability, rename thisAmount to result

Extract Method. The new method, still in class Customer

```
private double amountFor(Rental each) {
  double result = 0;
  switch (each.getMovie().getPriceCode()) {
     case Movie.REGULAR:
        result +=2;
        if (each.getDaysRented() > 2)
          result += (each.getDaysRented()-2)*1.5;
        break;


     case Movie.NEW_RELEASE:
        result += each.getDaysRented()*3;
        break;

     case Movie.CHILDRENS:
        result += 1.5;
        if (each.getDaysRented()>3)
          result += (each.getDaysRented()-3)*1.5;
        break;
  }
  return result;
}
```

What's still "wrong" here?

The Move Method Refactoring

- Problem: Unnecessary coupling from **Customer** to **Rental** through **getDaysRented()**.
- Unnecessary coupling to **Movie** too.
  - **Customer** does not have the "information" to compute the rental amount
- Solution: Move **amountFor(Rental)** to the class that is the logical "information expert"
- Key point: safety. **Test after refactoring**!

What class has the information to compute the charge amount?

Move Method, Mechanics

- Examine all features used by the source method that are defined on the source class. Consider if they should be moved also.
- Check the sub- and superclasses for other declarations of the method.
- Declare the method in the target class.
- Appropriately copy the code from source to target.
- Compile the target class.
- Reference the correct target object from the source.
- Turn the source method into a delegating method.
- Compile and test.

Move Method. getCharge(), now in class Rental

```
double getCharge() { // now in Rental! Name change
  double result = 0;
  // No reference to each!
  // No each, Rental object has info

  switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
                result +=2;
                if (getDaysRented() > 2)
                  result += (getDaysRented()-2)*1.5;
                break;


        case Movie.NEW_RELEASE:
                result += getDaysRented()*3;
                break;


        case Movie.CHILDRENS:
                result += 1.5;
                if (getDaysRented()>3)
                   result += (getDaysRented()-3)*1.5;
                break;
   }
  return result;
}
```

Move Method

Initially, replace body of old method with delegation:

```
class Customer {
  private double amountFor(Rental aRental) {
        return aRental.getCharge();
    }
}
```

Compile and test to see if it works.

Next, find each reference to **amountFor** and replace with call to the new method:

thisAmount = amountFor(each); becomes

thisAmount = each.getCharge();

Compile and test!

New and improved statement()

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPionts = 0;
  Iterator<Rental> rentals = _rentals.iterator();

  String result = "Record for " + getName() + "\n";
  while (rentals.hasNext()) {
        double thisAmount = 0;
        Rental each = rentals.next();
        thisAmount = each.getCharge(); // BIG CHANGE!

        // …code for frequent renter points


        result += "\t" + each.getMovie().getTitle() +  "\t" +
                String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

  // code to add totalAmount and
  // frequentRenterPoints to result string, return result
  // and DONE!
}
```

Is **thisAmount** necessary?

The Replace Temp with Query Refactoring

- Problem: Temporary variable **thisAmount** is meaningless, hinders readability
- Solution: Replace **thisAmount** with "query method" **each.getCharge()**
  - Claim: A "query method" is more informative
- Aside: "query methods" must be free of side effects (i.e., they modify nothing)
- Key point: safety. Test after refactoring!

Replace Temp With Query, Mechanics

- Look for a temporary variable that is assigned to only once. Why?
- Declare the temp as final
- Compile (makes sure temp is assigned once!)
- Extract the right-hand side of the assignment into a "query"; replace all occurrences of temp with query
  - Method computing the value of temp should be a "query" method, i.e., it should be free of side effects! Why?
- Compile and test

Replace Temp with Query

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Iterator<Rental> rentals = _rentals.iterator();
  String result = "Record for " + getName() + "\n";
  while (rentals.hasNext()) {
        Rental each = rentals.next();
        double thisAmount = each.getCharge();
        // …code for frequent renter points
        result += "\t" + each.getMovie().getTitle() + "\t" +
                              String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
  }
// code to add totalAmount and
// frequentRenterPoints to result string, return result and DONE!
}
```

We got rid of **thisAmount** temp, replaced it with **each.getCharge()**. Do you see issues with this refactoring?

What else can we do?

```
public String statement() {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Iterator<Rental> rentals = _rentals.iterator();
  String result = "Record for " + getName() + "\n";
  while (rentals.hasNext()) {
      Rental each = rentals.next();
      frequentRenterPoints++;
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE)
              && each.getDaysRented() > 1)
         frequentRenterPoints++;

       result += "\t" + each.getMovie().getTitle() + "\t" +
                 String.valueOf(each.getCharge()) + "\n";
       totalAmount += each.getCharge();
    } // end while
    // code to add totalAmount and
    // frequentRenterPoints to result string, return result, and DONE!
}
```

Extract Method + Move Method

```
public String statement() {
  …
  while (… ) { …
    // add frequent renter and other bonus points:
    frequentRenterPoints++;
    if ((each.getMovie().getPriceCode() ==Movie.NEW_RELEASE)
        && each.getDaysRented() > 1)
        frequentRenterPoints++;
     …
    }
    // code to add totalAmount and frequentRenterPoints to result
}
```

After Extract Method & Move Method we have:

```
frequentRenterPoints += each.getFrequentRenterPoints();
```

Replace Temp with Query, again

```
public String statement() {
      double totalAmount = 0;
      int frequentRenterPoints = 0;
      Iterator<Rental> rentals = _rentals.iterator();
      String result = "Rental Record for " + getName() + "\n";
      while (rentals.hasNext()) {
           Rental each = rentals.next();
           frequentRenterPoints += each.getFrequentRenterPoints();
           result += …+String.valueOf(each.getCharge())+…+"\n";
           totalAmount += each.getCharge();
      }
      result +=… totalAmount…+…frequentRenterPoints+…
      return result;
}
```

Can we replace these last two temps?

Replace Temp with Query

- Extract computation for **totalAmount** in a separate method in **Customer**. Is it a "query"?

```
private double getTotalCharge() {
    double result = 0;
    Iterator<Rental>rentals =_rentals.iterator();
    while (rentals.hasNext()) {
      Rental each = rentals.next();
      result += each.getCharge();
    }
    return result;
  }
```

Replace Temp with Query

- Similarly, extract computation for **frequentRenterPoints**

```
private double getTotalFrequentRenterPoints() {
    double result = 0;
    Iterator<Rental> rentals = _rentals.iterator();
    while (rentals.hasNext()) {
        Rental each = rentals.next();
        result += each.getFrequentRenterPoints();
    }
    return result;
}
```

First, take totalAmount out

```
public String statement() {
    int frequentRenterPoints = 0; // first, take totalAmount out
    Iterator<Rental> rentals = _rentals.iterator();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasNext()) {
          Rental each = rentals.next();
          frequentRenterPoints+=each.getFrequentRenterPoints();
          result += …+String.valueOf(each.getCharge())+…+"\n";
    }
    result += …+getTotalCharge()+…+frequentRenterPoints
    return result;
}
```

The key point: small steps, preserving behavior!

Next, take frequentRenterPoints out

```
public String statement() {
    Iterator<Rental> rentals = _rentals.iterator();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasNext()) {
        Rental each = rentals.next();
            result +=
                    …+String.valueOf(each.getCharge())+…+"\n";
    }
    result += …+getTotalCharge() + … +
                            getTotalFrequentRenterPoints();
    return result;
}
```

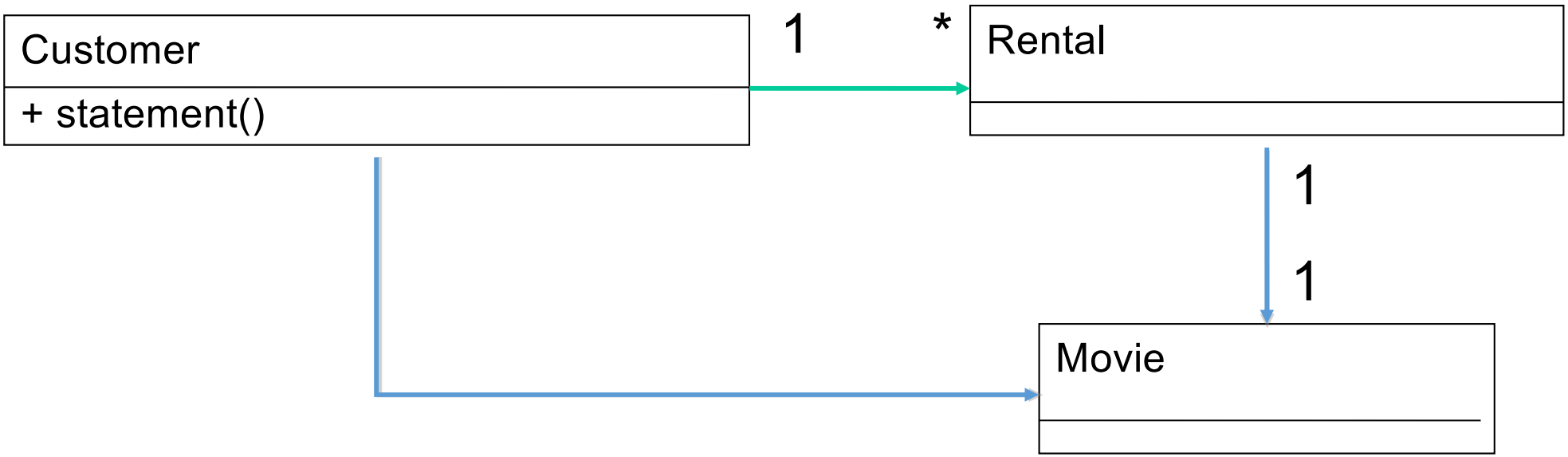
- Methods **getTotalCharge()** and **getFrequentRenterPoints()**, two private methods in Customer. Both iterate over the rentals. Issues?

Refactoring So Far

- Small-step, behavior-preserving transformations. Continuously test.
- Goal: achieve code that is short, tight, clear and without duplication. Eliminate **code smells**
- Refactorings
  - Extract method
  - Move method
  - Replace temp with query… More

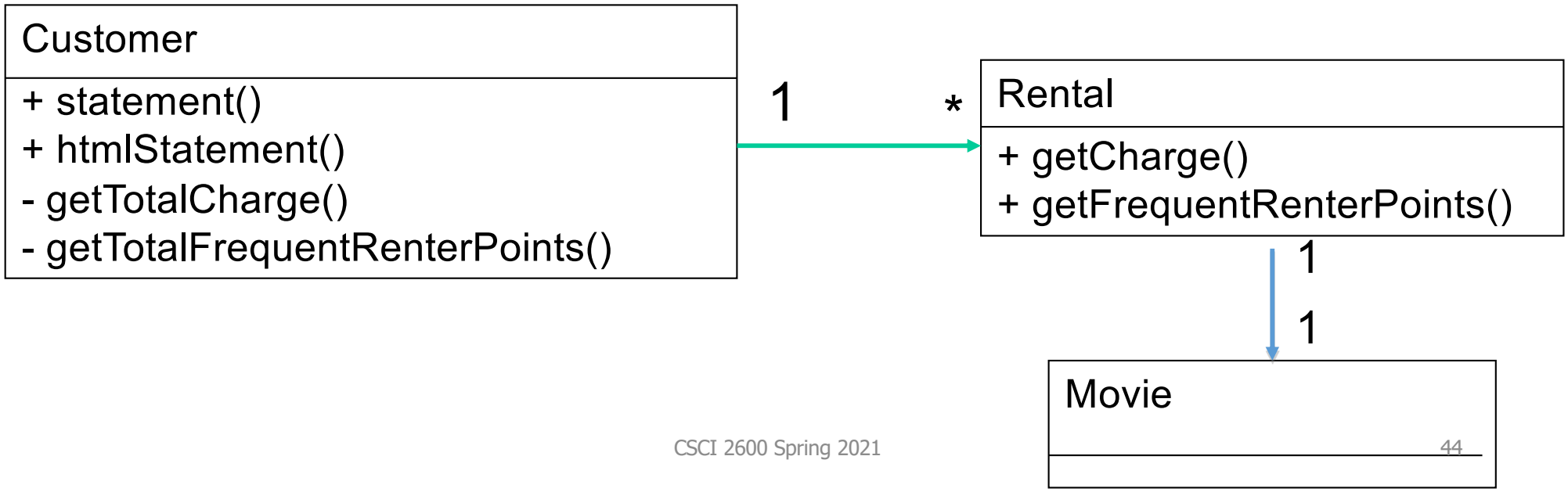
Before…

- Code smells: all code in long method statement(), unnecessary coupling between Customer and Rental and Customer and Movie



After…

- Shortened statement() with Extract Method, eliminated unnecessary coupling between Customer and Rental, and Customer and Movie with Move Method, improved readability with Replace Temp with Query



Still refactoring… Back to getCharge

```
double getCharge() { // now in Rental
 double result = 0;
 switch (getMovie().getPriceCode()) {
   case Movie.REGULAR:
     result +=2;
     if (getDaysRented()>2)  result += (getDaysRented()-2)*1.5;
     break;
   case Movie.NEW_RELEASE:
     result += getDaysRented()*3;
     break;

   case Movie.CHILDRENS:
     result += 1.5;
     if (getDaysRented()>3) result += (getDaysRented()-3)*1.5;
         break;
 }
 return result;
}
```

What's wrong here?

Replacing Conditional Logic

- Problem: A switch statement can be bad.
  - Hard to understand
  - Hard to change
- First step towards solution: move **getCharge** and **getFrequentRenterPoints** from **Rental** to **Movie**:

```
class Rental { // replace with delegation
        double getCharge() {
                return _movie.getCharge(_daysRented);
        }
…
```

Move Method

```
double getCharge(int daysRented) { // now in Movie
 double result = 0;
 switch (getPriceCode()) { // Now, switch is on OWN data
    case Movie.REGULAR:
       result +=2;
       if (daysRented>2)  result += (daysRented-2)*1.5;
       break;
    case Movie.NEW_RELEASE:
       result += daysRented*3;
       break;

     case Movie.CHILDRENS:
       result += 1.5;
       if (getDaysRented>3) result += (getDaysRented-3)*1.5;
       break;
    }
 return result;
}
```

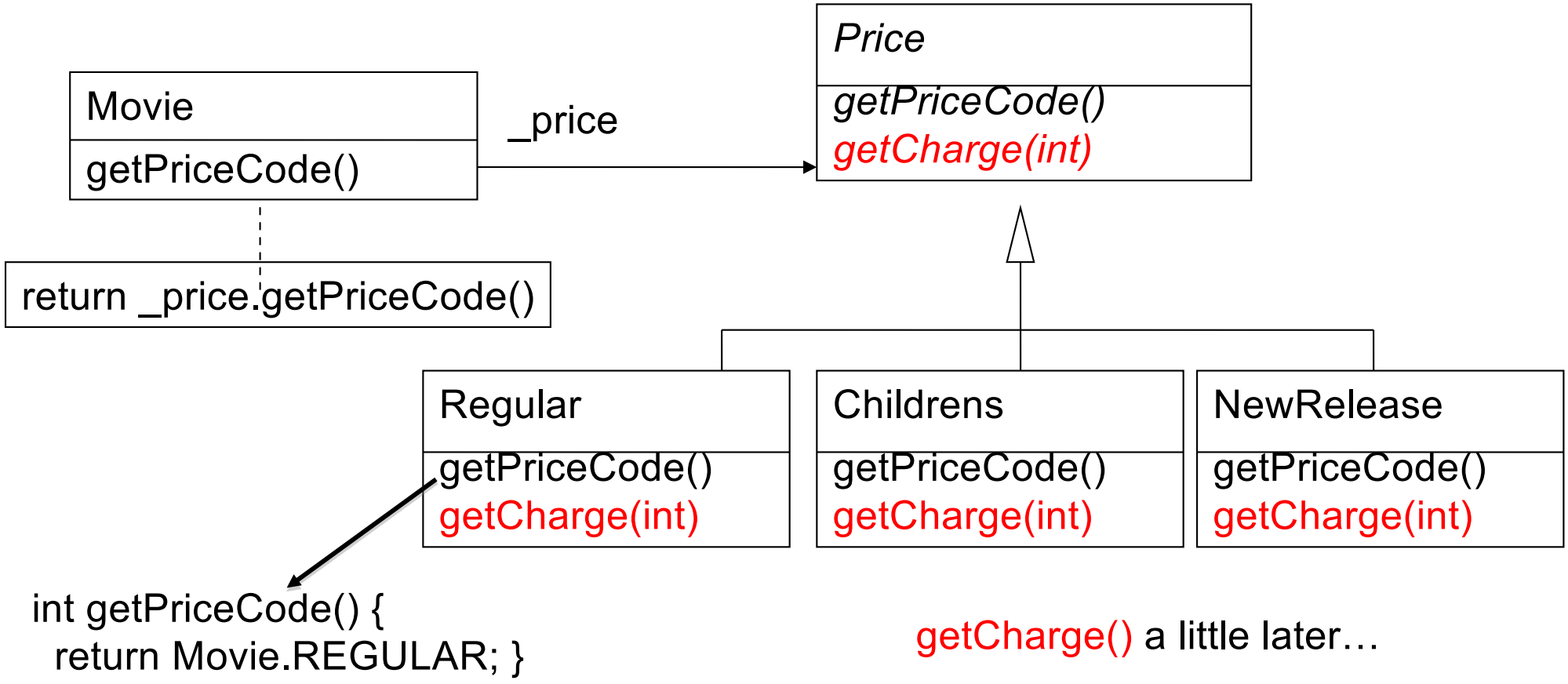
Replacing Conditional Logic

- Problem: a switch statement can be a bad idea; it is difficult to maintain and error prone
  - Switch statements in themselves aren't bad but be careful using them.
- Solution: replace switch with subtype polymorphism!
  - Abstract class ***Price*** with concrete subclasses **Regular, Childrens, NewRelease**
  - Each ***Price*** subclass defines its own
    - **getPriceCode()**
    - **getCharge()**

The State Design Pattern

- Question: Can we have an algorithm vary independently from the object that uses it?
- Example: Movie pricing…
  - Class **Movie** represents a movie
  - There are several pricing algorithms/strategies
  - We need to add new algorithms/strategies easily
  - Placing the pricing algorithms/strategies in **Movie** will make **Movie** too big and too complex
    - Switch statement code smell

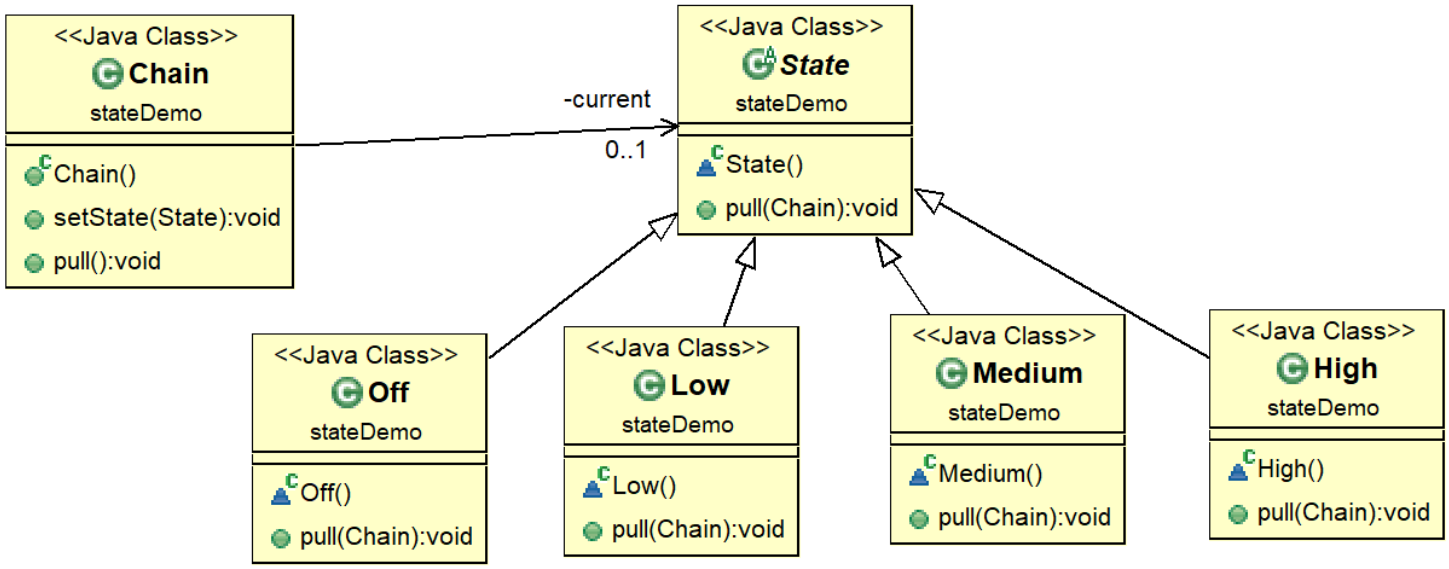
The Replace Type Code with State/Strategy Refactoring

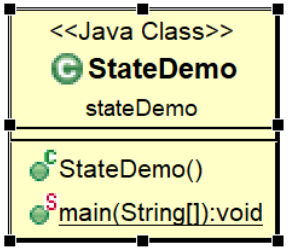


- Replaced _priceCode (the type code) with Price _price (Strategy)
- State and Strategy are often interchangeable
- Important point: replace switch with subtype polymorphism!

Aside: the State Design Pattern

- Question: How can an object alter its behavior when its internal state changes?
- Example: A TCPConnection class representing a network connection
  - TCPConnection can be in one of three states: Established, Listening or Closed
  - When a TCPConnection object receives requests (e.g., open, close) from the client, it responds differently depending on its current state
- Example: Pull chain on ceiling fan





StateDemo.java

State Pattern

- The State pattern is a solution to the problem of how to make behavior depend on state.
- Define a "context" class to present a single interface to the outside world.
  - In the example, Chain is the context
- Define a State abstract base class.
- Represent the different "states" of the state machine as derived classes of the State base class.
- Define state-specific behavior in the appropriate State derived classes.
- Maintain the current "state" in the "context" class by composition.
- To change the state of the state machine, change the current "state" reference.

Replace Type Code with State/Strategy

- Add the new concrete **Price** classes
- In **Movie**: **int _priceCode** becomes ***Price*** **_price**
- Change **Movie**'s accessors to use **_price**

```
int getPriceCode() { return _price.getPriceCode(); }

void setPriceCode(int arg) {
    switch (arg) {
            case REGULAR: _price = new Regular();
… }
```

Move Method **getCharge()** from **Movie** to **Price**

Move Method: getCharge() moves from Movie to *Price*

```
double getCharge(int daysRented) { // now in Price…
  double result = 0;

  switch (getPriceCode()) { // Note this stays the same!
        case REGULAR:
               result +=2;
               if (daysRented>2)  result += (daysRented-2)*1.5;
               break;
        case NEW_RELEASE:
               result += daysRented*3;
               break;

        case CHILDRENS:
               result += 1.5;
               if (getDaysRented>3) result += (getDaysRented-3)*1.5;
               break;
       }
  return result;
}
```

The Replace Conditional with Polymorphism Refactoring

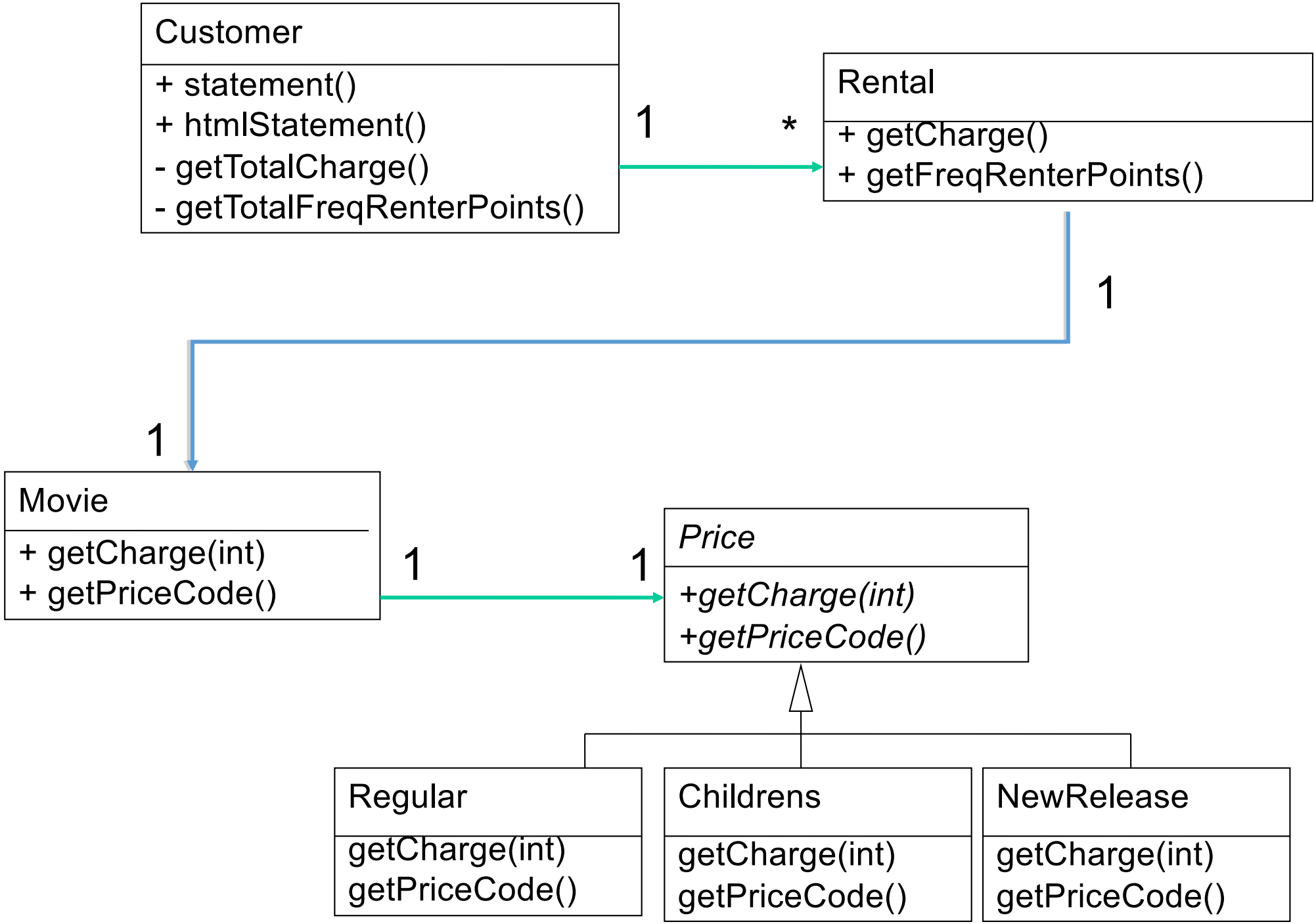
- **Regular** defines its **getCharge**:

```
double getCharge(int daysRented) {
  double result = 2;
  if (daysRented > 2)
       result += (daysRented - 2)*1.5;
  return result;
}
```

- **Childrens** and **NewRelease** define their **getCharge(int)**
- **getCharge(int)** in ***Price*** becomes abstract

So Far

- Extract Method
- Move Method
- Replace Temp with Query
- Replace Type Code with State/Strategy and
- Replace Conditional with Polymorphism
  - Last two refactorings go together, break transformation into small steps
  - Goal: replace switch with polymorphism
    - First, replace the type code with State/Strategy
    - Second, place each case branch into a subclass, add virtual call (e.g., **_price.getCharge(daysRented)**)


Customer
+ statement()
+ htmlStatement()
- getTotalCharge()
- getTotalFreqRenterPoints()
1
*
Rental
+ getCharge()
+ getFreqRenterPoints()
1
1
Movie
+ getCharge(int)
+ getPriceCode()
1
1
Price
+getCharge(int)
+getPriceCode()
Regular
getCharge(int)
getPriceCode()
Childrens
getCharge(int)
getPriceCode()
NewRelease
getCharge(int)
getPriceCode()

Now, let's add a method

```
public String htmlStatement() {
      Iterator<Rental> rentals =  _rentals.iterator();
      String result = "<H1>Rental Record for <EM>" +
             getName() + "</EM><H1><P>\n";
      while (rentals.hasNext()) {
       Rental each = rentals.next();
       result += …+each.getCharge()+…+"\n"; // add HTML…
     }
     result +=… +getTotalCharge()+…
                    +getFrequentRenterPoints() // + HTML
     return result;
}
```

•Key point: refactoring is intertwined with addition of new methods and functionality. What's the problem here?

Still Refactoring…

```
public String statement() {
    Iterator<Rental> rentals = _rentals.iterator();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasNext()) {
       Rental each = rentals.next();
      result += +String.valueOf(each.getCharge())+…+"\n";
    }
    result += …+getTotalCharge()+…
                        +getTotalFrequentRenterPoints()+…
    return result;
}
```

At some point, we created **htmlStatement()** which was the same, except that **result** had HTML symbols.

Still Refactoring…

```
public String htmlStatement() {
   Iterator<Rental> rentals = _rentals.iterator();
   String result = "<H1>Rental Record for <EM>" +
                                        getName() + "</EM><H1><P>\n";
   while (rentals.hasNext()) {
      Rental each = rentals.next();
      result += …+each.getCharge()+…+"\n"; // + HTML
    }
   result +=… +getTotalCharge()+…
                      +getFrequentRenterPoints();  // + HTML
}
```

We created htmlStatement using cut-and-paste. Antipattern?

There is a design pattern that helps deal with duplicate code!

Before we deal with duplicate code…

- Introduce Strategy for printing statements

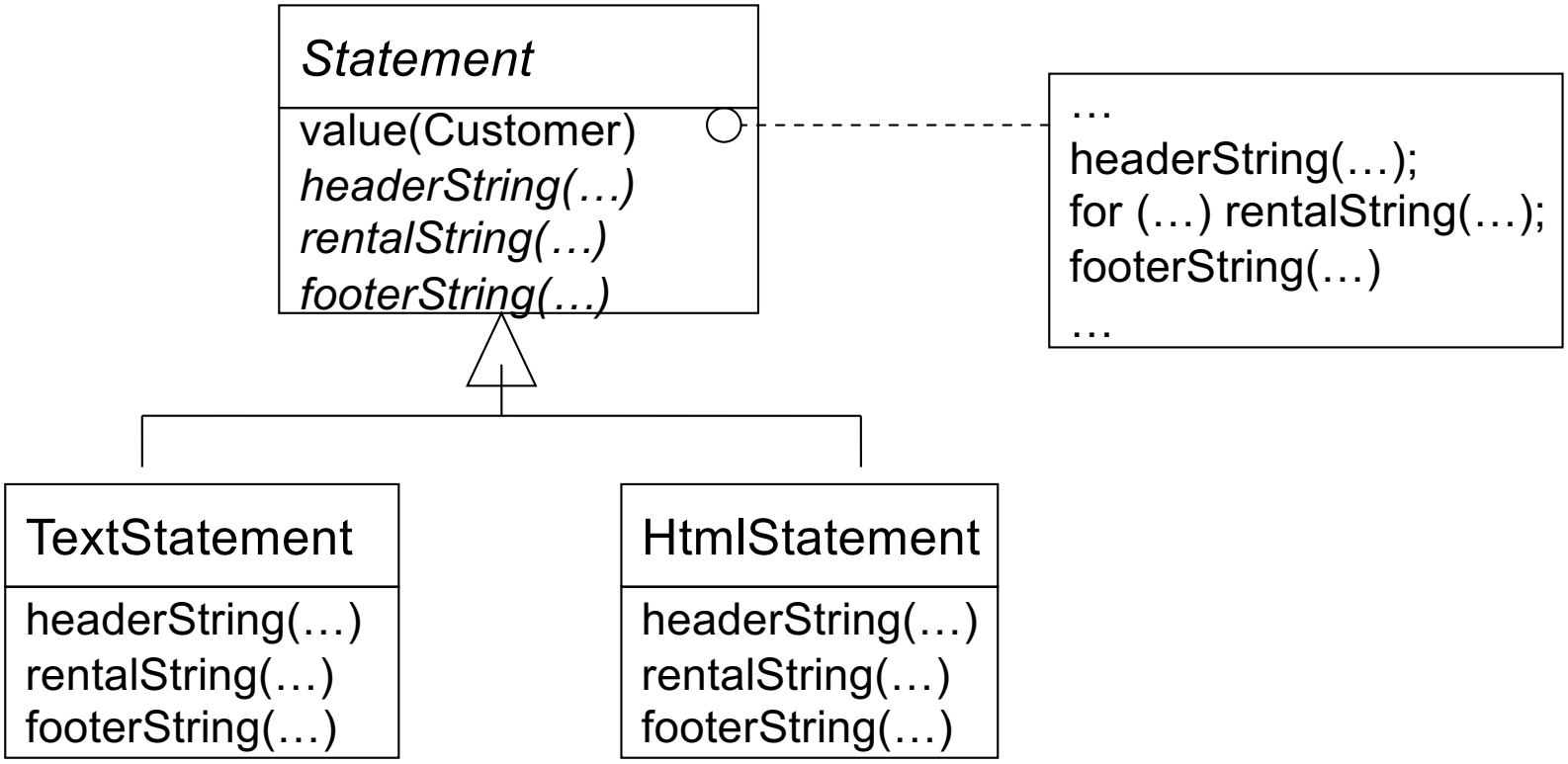
```
abstract class Statement {} // Abstract Strategy
class TextStatement extends Statement {}
class HtmlStatement extends Statement {}
```

- Move Method
  - Customer.statement() to TextStatement.value(Customer)
  - Customer.htmlStatement() to HtmlStatement.value(Customer)
- Delegation in Customer
  - String statement() { return (new TextStatement).value(this); }
  - String htmlStatement() { return (new HtmlStatement).value(this); }

Aside: The Template Method Design Pattern

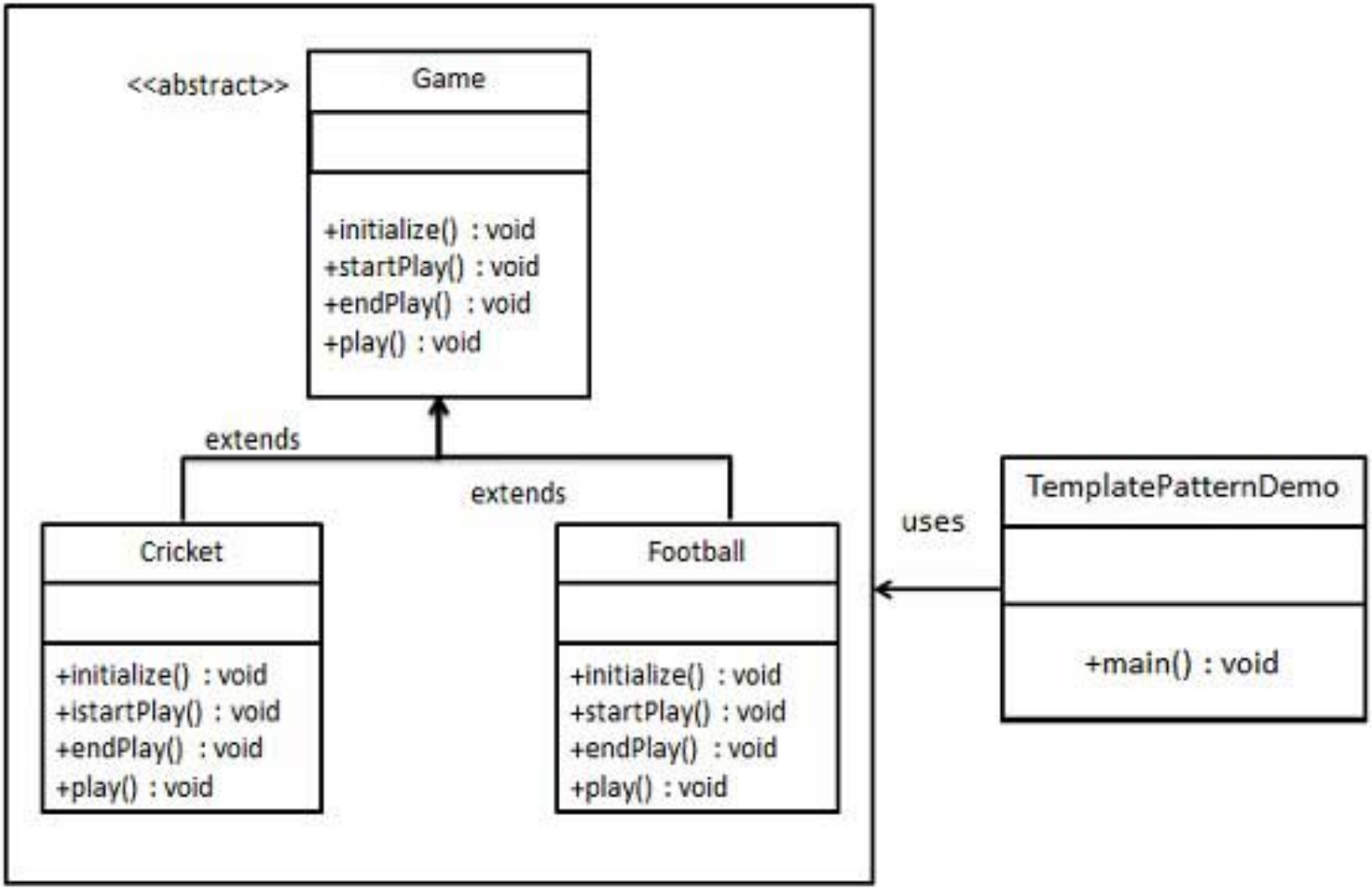
- Problem: We have several methods that implement the same algorithm, but differ at some steps
  - E.g., TextStatement.value and HtmlStatement.value
- Solution: Define the skeleton of the algorithm in a superclass, defer differing steps to subclasses
- Example: TextStatement and HtmlStatement
  - Same algorithm for TextStatement.value and HtmlStatement.value
    - First, record header substring: customer info
    - Iterate over rentals, record each rental substring
    - Finally, record footer substring: total charge
  - Recorded substrings differ from Text to Html

Aside: The Template Method Pattern



- value is the template method
- headerString, rentalString, footerString are hooks
- Hooks, abstract in Statement, defer to subclass

Simple Template Pattern Example



https://www.tutorialspoint.com/design_pattern/template_pattern.htm

```
public abstract class Game {
   abstract void initialize();
   abstract void startPlay();
   abstract void endPlay();

   //template method
   public final void play(){

      //initialize the game
      initialize();

      //start game
      startPlay();

      //end game
      endPlay();
   }
}
```

```
public class Cricket extends Game {

   @Override
   void endPlay() {
      System.out.println("Cricket Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Cricket Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Cricket Game Started. Enjoy the game!");
   }
}
```

```
public class Football extends Game {

   @Override
   void endPlay() {
      System.out.println("Football Game Finished!");
   }

   @Override
   void initialize() {
      System.out.println("Football Game Initialized! Start playing.");
   }

   @Override
   void startPlay() {
      System.out.println("Football Game Started. Enjoy the game!");
   }
}
```

```
public class TemplatePatternDemo {
   public static void main(String[] args) {

      Game game = new Cricket();
      game.play();
      System.out.println();
      game = new Football();
      game.play();
   }
}
```

Output:
Cricket Game Initialized! Start playing.
Cricket Game Started. Enjoy the game!
Cricket Game Finished!

Football Game Initialized! Start playing.
Football Game Started. Enjoy the game!
Football Game Finished!

The Form Template Method Refactoring

- Before refactoring TextStatement.value and HtmlStatement.value are very similar
  - The “duplicate code” smell
- Refactor to form a template method
  - Eliminates the “duplicate code” smell

The Form Template Method Refactoring

- Decompose the methods using **Extract Method** so that all extracted methods are either identical among the different subclasses, or completely different
- Use **Pull Up Method** (another refactoring!) to pull the identical methods, from one subclass, into the superclass
- For the different methods use **Rename Method**
  - Make sure that each one has the same signature
  - Declare them as abstract in the superclass
- Compile and test after the signature changes
- Remove the other identical methods, compile and test after each removal

Form Template Method step 1: Extract Method

```
class TextStatement extends Statement { …
  public String value(Customer c) {
     Iterator<Rental> rentals = c.getRentals();
     String result = "Rental Record for " + getName()+ \n";
     // new code (after Extract Method):
     String result = headerString(c);
     while (rentals.hasNext()) {
        Rental each = rentals.next();
        result += …+each.getCharge()+…+"\n";
        new code: result += eachRentalString(each);
     }
  result +=… +getTotalCharge()+…+getFrequentRenterPoints();
  new code:  result += footerString(c);
  return result;
}
```

From Template Method, step 2: Pull Up Method

Now, pull up value(Customer) from TextStatement into Statement:

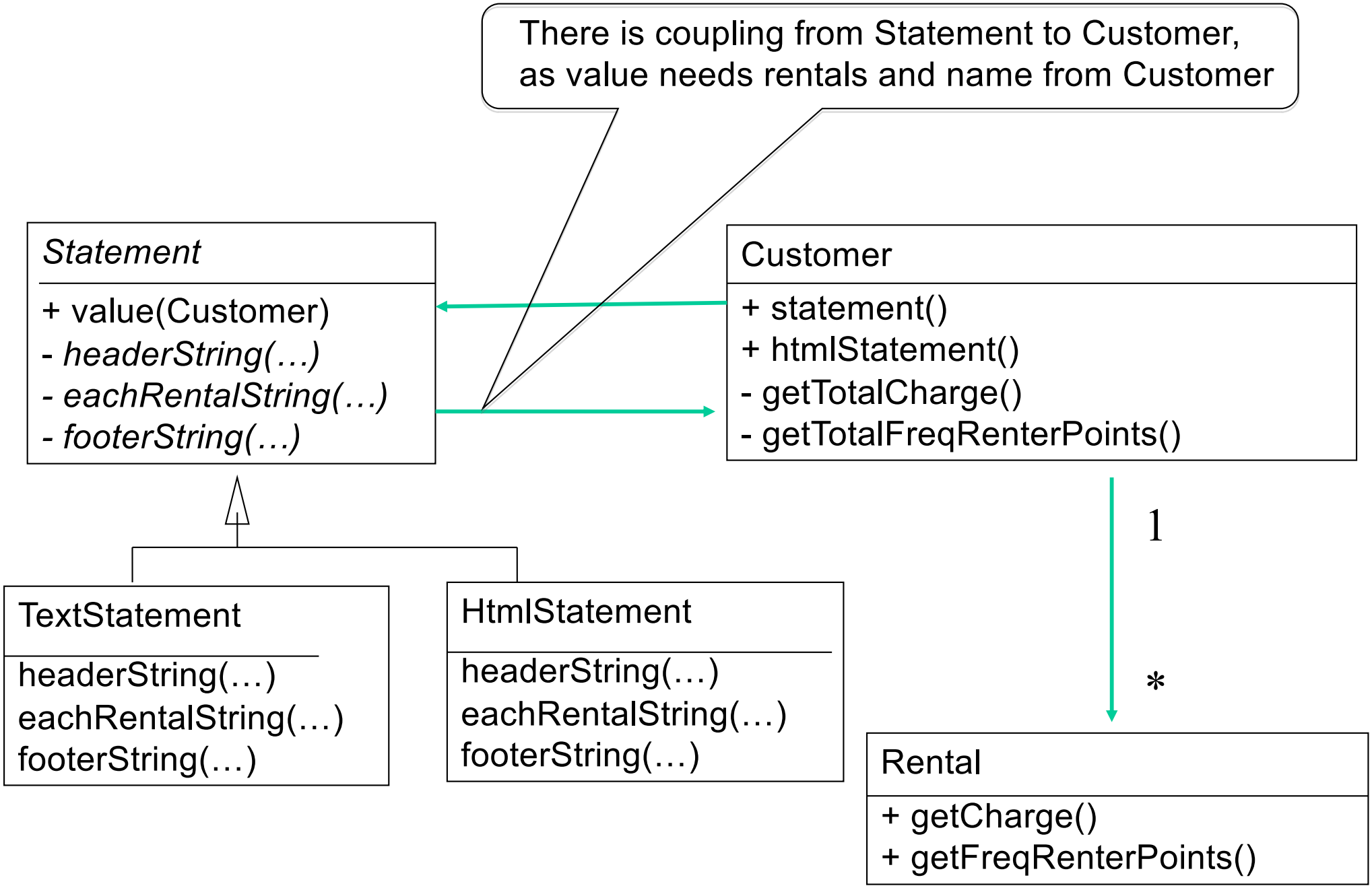
```
abstract class Statement {
  public String value(Customer c) {
            Iterator<Rental> rentals = c.getRentals();
            String result = headerString(c);
            while (rentals.hasNext()) {
                        Rental each = rentals.next ();
                        result += eachRentalString(each);
            }
            result += footerString(c);
            return result;
  }
}
```

Form Template Method

- Step 3: Make the hooks abstract in Statement

```
abstract String headerString(Customer c);
abstract String eachRentalString(Rental each);
abstract String footerString(Customer c);
```

- Step 4: Compile and test!
- Step 5: Remove value(Customer) from HtmlStatement. Compile and test!


There is coupling from Statement to Customer, as value needs rentals and name from Customer
Statement
+ value(Customer)
- headerString(…)
- eachRentalString(…)
- footerString(…)
Customer
+ statement()
+ htmlStatement()
- getTotalCharge()
- getTotalFreqRenterPoints()
1
*
TextStatement
headerString(…)
eachRentalString(…)
footerString(…)
HtmlStatement
headerString(…)
eachRentalString(…)
footerString(…)
Rental
+ getCharge()
+ getFreqRenterPoints()

One last refactoring. Back to getCharge(int)

```
class Regular extends Price {
   double getCharge(int daysRented) {
     double result = 2;
     if (daysRented > 2)
        result += (daysRented - 2)*1.5;
     return result;
  }
}
```

Replace Magic Number with Symbolic Constant

```
class Regular extends Price {
  final static double INTRO_DAYS = 2;
  final static double INTRO_RATE = 1;
  final static double REGULAR_RATE = 1.5;


  double getCharge(int daysRented) {
    double result = INTRO_DAYS * INTRO_RATE;
    if (daysRented > INTRO_DAYS)
      result += (daysRented - INTRO_DAYS)*REGULAR_RATE;
    return result;
  }
}
```