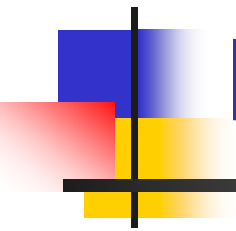
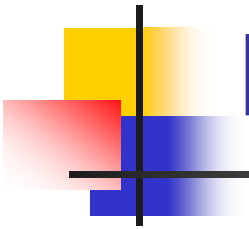


Refactoring

Paolo Tonella

ITC-irst, Centro per la Ricerca Scientifica e Tecnologica



Refactoring

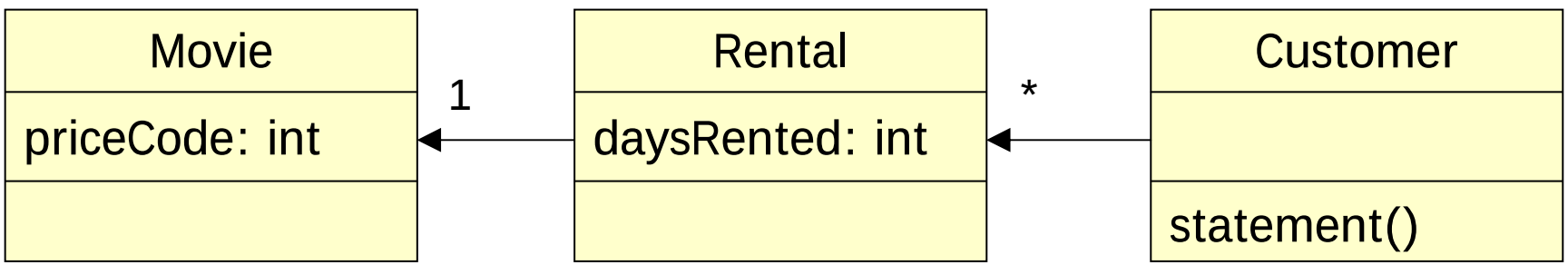
Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, while it improves its internal structure.

- Disciplined way to clean up code.
- The design of the system is improved after the code has been written.
- Design occurs continuously during development.
- The risks associated with the production of a good design from the very beginning are reduced.

Example: a video store

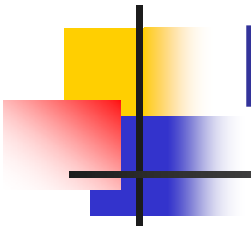
```
class Movie {
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;
  public static final int CHILDREN = 2;
  private String _title;
  private int _priceCode;
}
```

```
class Rental {
  private Movie _movie;
  private int _daysRented;
}
class Customer {
  private String _name;
  private Vector _rentals =
        new Vector();
}
```



Emitting statements

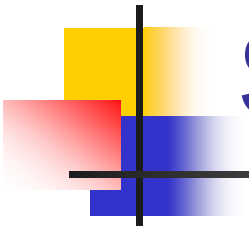
```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      double thisAmount = 0;
      Rental each = (Rental) rentals.nextElement();
      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.CHILDREN:
          thisAmount += 1.5;
          if (each.getDaysRented() > 3)
             thisAmount += (each.getDaysRented() - 3) * 1.5; break;
      }
      frequentRenterPoints++;
      if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
          each.getDaysRented() > 1) frequentRenterPoints++;
      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(thisAmount) + "\n";
      totalAmount += thisAmount;
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";
    return result;
  }
```



Method `statement`

- Not well designed - it has too many responsibilities;
- Not object oriented - computation is not delegated to objects.

Yet, it works: the compiler does not signal any error and testing does not reveal any defect. The problem is with its evolution: a poorly designed system is hard change. The compiler does not care whether the code is ugly or clean, but humans do care.

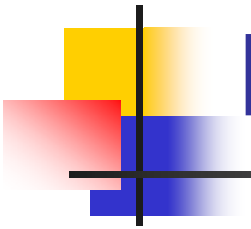


Software evolution

- Statements should also be printable in HTML format;
- The classification of the movies changes.

A quick-and-dirty solution could be “cloning” the `statement` method into a new one, to be named `HTMLStatement`.

When a feature has to be added to a program, if the code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.



Regression testing

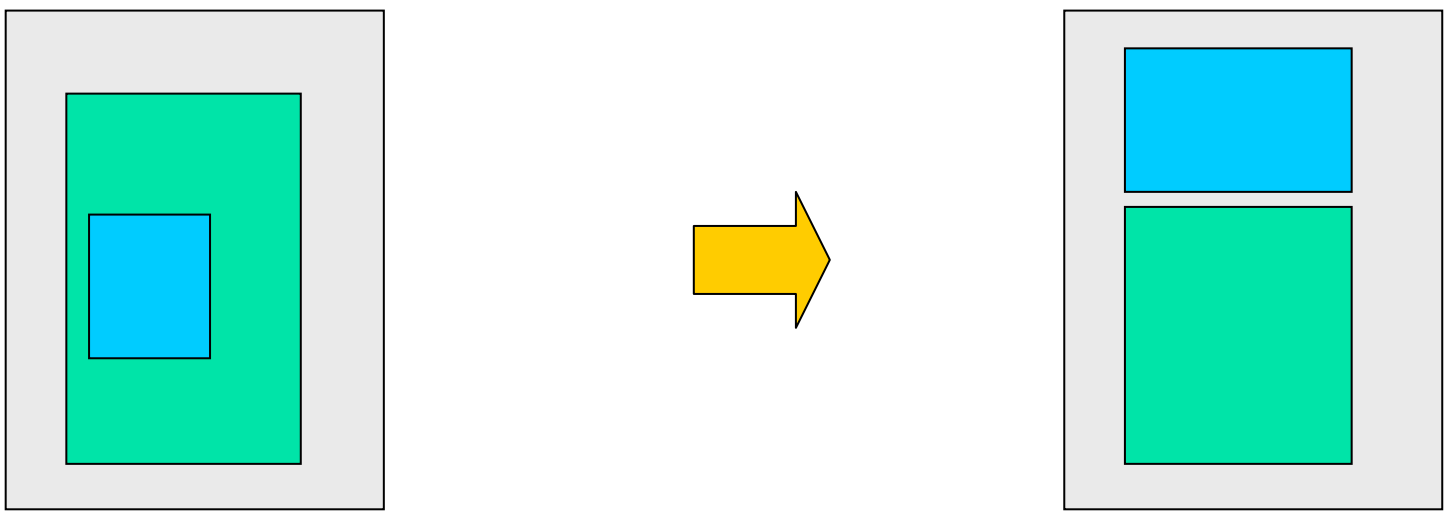
Before starting refactoring, it is important to have a solid test suite, with self-checking test cases. In fact, after refactoring the program, it is convenient to perform regression testing automatically, relying on its output to gain some confidence that bugs have not been introduced.

- Automatic test case execution.
- Automatic verification of the outputs.

Example of tool: JUnit

Extract method

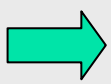
Refactoring Extract Method: *When a sequence of logically related statements can be grouped together, they can be turned into the body of a method, whose name should explain the isolated behavior. Referenced variables should be made available as parameters and/or return values, if not visible.*



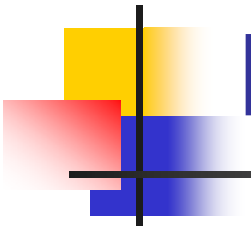
Extracting method amountFor

```
class Customer {
  ...
  public String statement() {
    ...
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      double thisAmount = amountFor(each);
      frequentRenterPoints++; ... }
  }
```

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



RUN REGRESSION TESTS

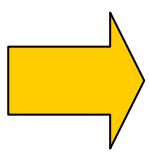


Renaming

Good code should communicate what it is doing clearly, and variable names are a key to clear code. Anybody can write code that a computer can understand. Good programmers write code that humans can understand.

Refactoring Renaming: *If the name of an entity does not reveal its purpose, it should be changed. All references to such an entity must be changed accordingly. Moreover, conflicts with existing entities must be avoided when choosing the new name.*

```
...
amountFor(Rental each)
```



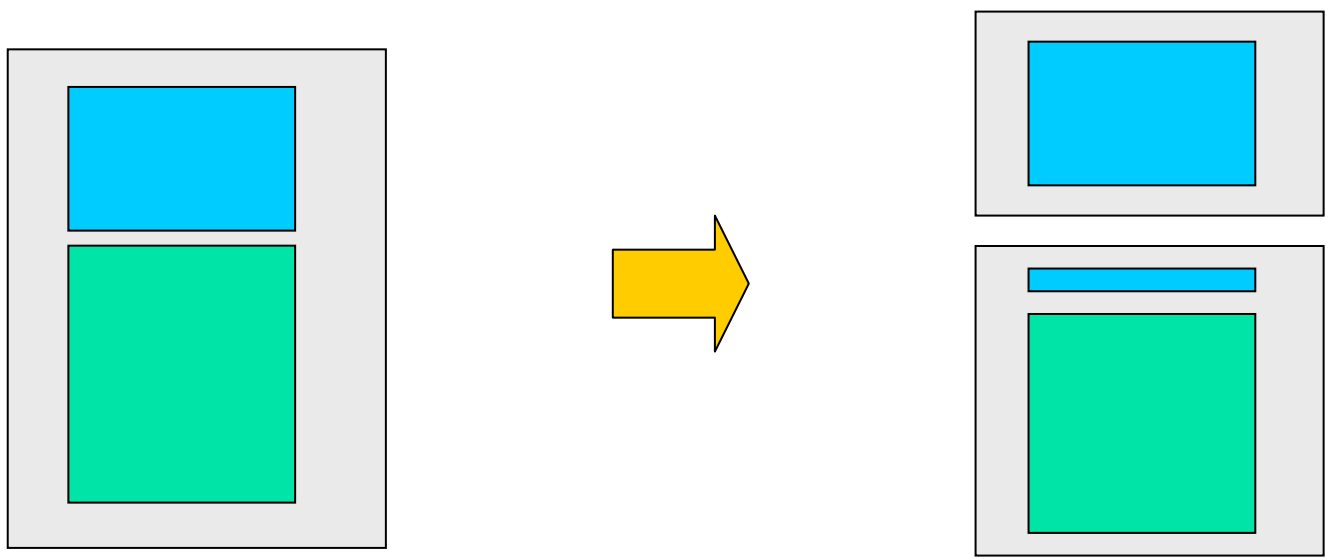
```
...
amountFor(Rental aRental)
```

Renaming each into aRental

```
private double amountFor(Rental aRental) {
    double result = 0;
    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
          result += 2;
          if (aRental.getDaysRented() > 2)
              result += (aRental.getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          result += aRental.getDaysRented() * 3;
          break;
        case Movie.CHILDREN:
          result += 1.5;
          if (aRental.getDaysRented() > 3)
             result += (aRental.getDaysRented() - 3) * 1.5;
          break;
    }
    return result;
  }
```

Move method

Refactoring Move Method: *If a method is, or will be, using or used by more features of another class than the class in which it is defined, a new method with a similar body can be created in the class it uses most. The old method can either be turned into a simple delegation, or it can be removed altogether.*



Moving amountFor to class Rental

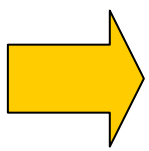
```
class Rental {
  ...
  private double getCharge() { // prev: amountFor(Rental aRental)
    double result = 0;
    switch (getMovie().getPriceCode()) {
        case Movie.REGULAR:
          result += 2;
          if (getDaysRented() > 2)
              result += (getDaysRented() - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          result += getDaysRented() * 3;
          break;
        case Movie.CHILDREN:
          result += 1.5;
          if (getDaysRented() > 3)
             result += (getDaysRented() - 3) * 1.5;
          break;
    }
    return result;
  }
}
class Customer {
   public String statement() {
    ...
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      double thisAmount = each.getCharge(); // prev: amountFor(each)
    ...
}
```

Replace temporary

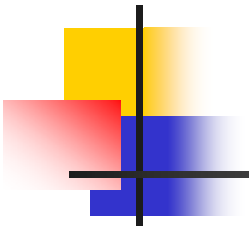
Temporary variables tend to encourage long methods, since the method body is the only place where the value of the local variable can be accessed. By replacing a temporary variable with a query method, any method in the class can reach the related information.

Refactoring Replace Temporary Variable with Query: *When a temporary variable is used to hold the result of an expression, it is possible to extract the expression into a query method. All references to the temporary variable can be replaced with invocations to the query method. Moreover, the new method can be used in other methods.*

String.valueOf(thisAmount)



String.valueOf(each.getCharge())



Replacing `thisAmount`

```
public String statement() {
    ...
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      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();
    }
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";
    return result;
  }
```

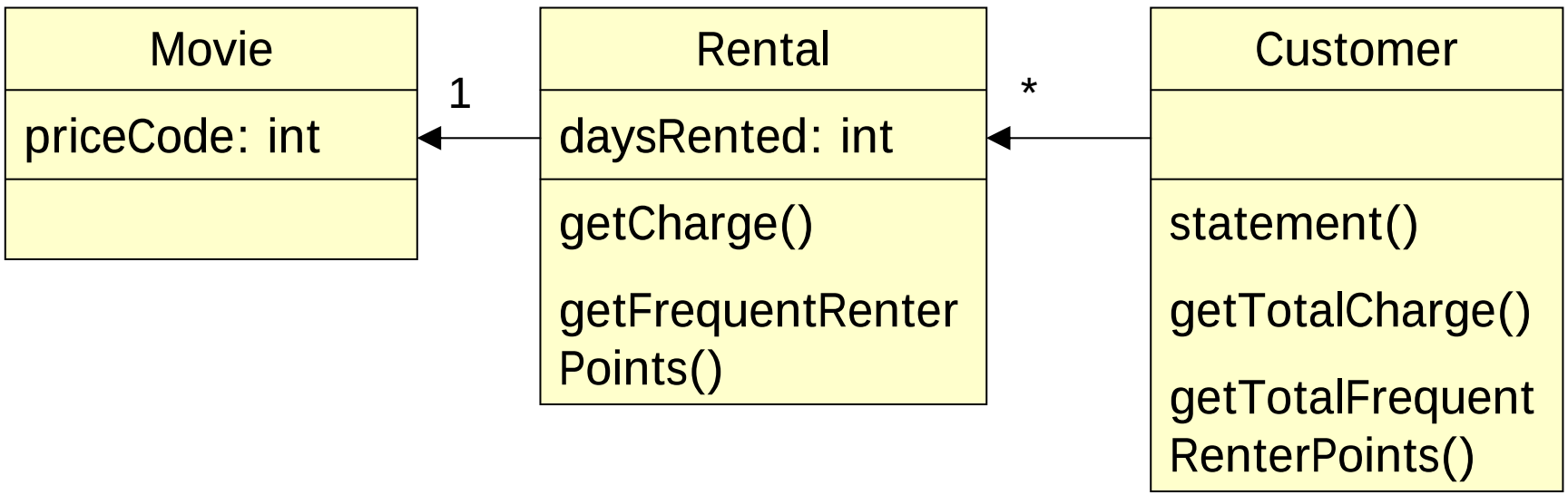
Moving the `frequent-RenterPoints` computation

```
class Customer {
  ...
  public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      frequentRenterPoints += each.getFrequentRenterPoints();
      ... }}
}
class Rental {
  ...
  public int getFrequentRenterPoints() {
    if ((getMovie().getPriceCode() == Movie.NEW_RELEASE) && getDaysRented() > 1)
       return 2;
    else
       return 1;
  }
}
```

Replacing totalAmount and frequentRenterPoints

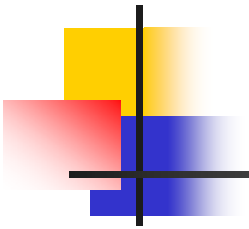
```
class Customer {
 public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += "\t" + each.getMovie().getTitle() + "\t" +
                String.valueOf(each.getCharge()) + "\n";
    }
    result += "Amount owed is " + String.valueOf(getTotalCharge());
    result += "You earned " +
        String.valueOf(getTotalFrequentRenterPoints()) +
         " frequent renter points";
    return result;
  }
  public double getTotalCharge() {
    double result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getCharge();
    } return result;
  }
  public int getTotalFrequentRenterPoints() {
    int result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getFrequentRenterPoints();
    } return result;
  }
}
```

New class diagram



Adding HTMLStatement

```
class Customer {
  ...
  public String HTMLStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Rentals for <EM>" + getName() +
                    "</EM></H1><P>\n";
    while (rentals.hasMoreElements()) {
      Rental each = (Rental) rentals.nextElement();
      result += each.getMovie().getTitle() + ": " +
                String.valueOf(each.getCharge()) + "<BR>\n";
    }
    result += "<P>You owe <EM>" +
              String.valueOf(getTotalCharge()) + "</EM><P>\n";
    result += "You earned <EM>" +
              String.valueOf(getTotalFrequentRenterPoints()) +
              "</EM> frequent renter points<P>";
    return result;
  }
}
```



Introducing polymorphism

Polymorphism allows avoiding an explicit conditional when the behavior of an object depends on its type.

- If conditional code is present, each time a new type is added, all conditionals sparsed in the code have to be found and updated.
- On the contrary, if conditional code is replaced with polymorphism, it is sufficient creating a new subclass and providing the appropriate methods.
- Clients of a class don't need to know about the subclasses, thus reducing the dependencies in the system and simplifying its update.

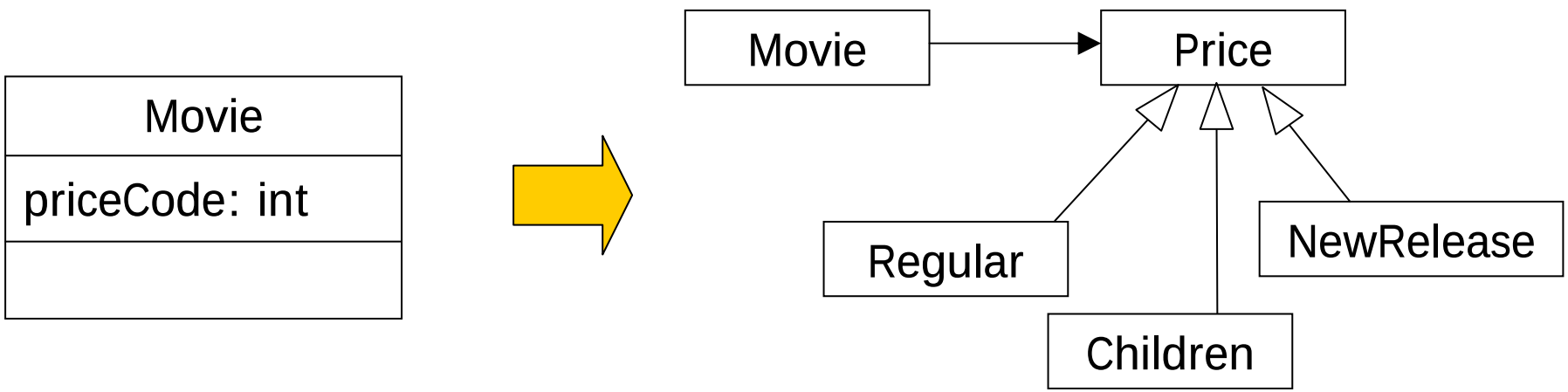
Moving getCharge and getFrequentRenterPoints

```
class Rental {
  public double getCharge() { return _movie.getCharge(_daysRented); }
  public int getFrequentRenterPoints() {
    return _movie.getFrequentRenterPoints(_daysRented);
  }
}
class Movie {
  public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case REGULAR:
          result += 2;
          if (daysRented > 2) result += (daysRented - 2) * 1.5; break;
        case NEW_RELEASE:
          result += daysRented * 3; break;
        case CHILDREN:
          result += 1.5;
          if (daysRented > 3) result += (daysRented - 3) * 1.5; break;
    } return result;
  }
  public int getFrequentRenterPoints(int daysRented) {
    if ((getPriceCode() == NEW_RELEASE) && daysRented > 1) return 2;
    else return 1;
  }
}
```

Replace type code with state/strategy

Introducing subclasses of **`Movie`** does not work in our case, since while a movie can change its classification over its lifetime, an object cannot. The solution is adopting the *State design pattern*.

Refactoring Replace Type Code with State/Strategy: *When a type code affects the behavior of a class, but subclassing cannot be used, the type code can be replaced with a state object.*



Adding class Price

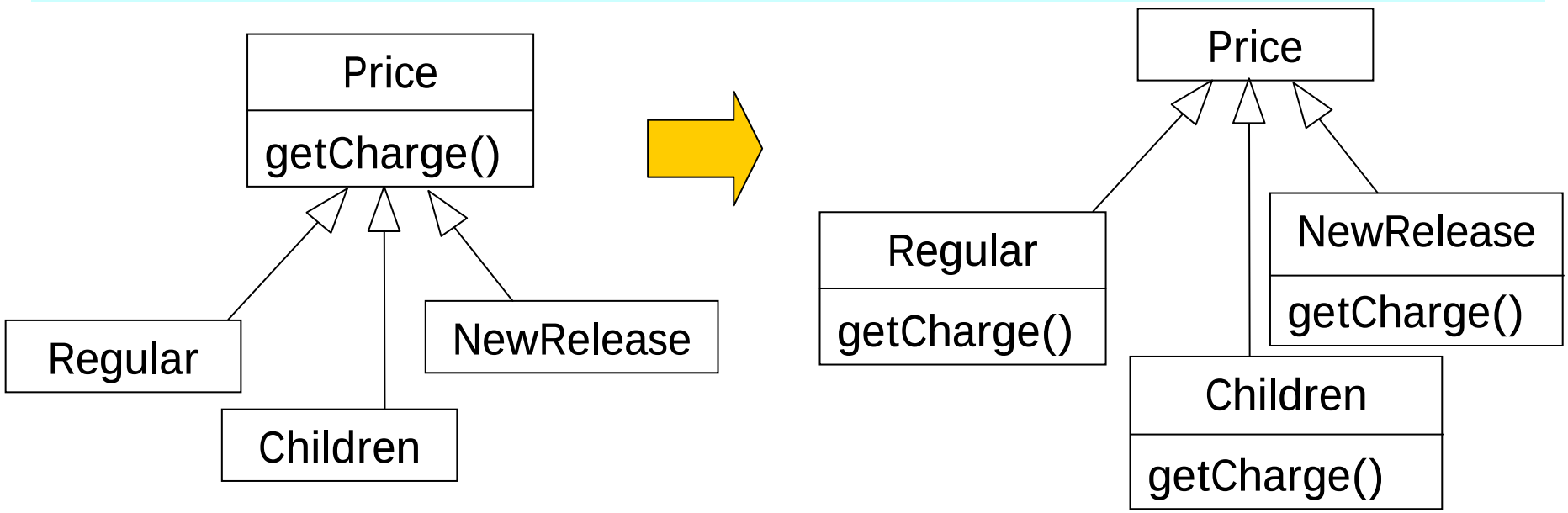
```
abstract class Price {
  abstract int getPriceCode();
  public double getCharge(int daysRented) {
    double result = 0;
    switch (getPriceCode()) {
        case Movie.REGULAR:
          result += 2;
          if (daysRented > 2)
              result += (daysRented - 2) * 1.5;
          break;
        case Movie.NEW_RELEASE:
          result += daysRented * 3;
          break;
        case Movie.CHILDREN:
          result += 1.5;
          if (daysRented > 3)
             result += (daysRented - 3) * 1.5;
          break;
    }
    return result;
  }
}
class ChildrenPrice extends Price {
  int getPriceCode() { return Movie.CHILDREN; }
}
class NewReleasePrice extends Price {
  int getPriceCode() { return Movie.NEW_RELEASE; }
}
class RegularPrice extends Price {
  int getPriceCode() { return Movie.REGULAR; }
}
```

Adding class Price

```
class Movie {
  public static final int REGULAR = 0;
  public static final int NEW_RELEASE = 1;
  public static final int CHILDREN = 2;

  private String _title;
  private Price _price;

  public void setPrice(int priceCode) {
    switch (priceCode) {
        case REGULAR:
          _price = new RegularPrice();
          break;
        case NEW_RELEASE:
          _price = new NewReleasePrice();
          break;
        case CHILDREN:
          _price = new ChildrenPrice();
          break;
    }
  }

  public double getCharge(int daysRented) {
    return _price.getCharge(daysRented);
  }
}
```

Replace conditional with polymorphism

Refactoring **Replace Conditional with Polymorphism**: *When a conditional statement chooses a different behavior depending on the type of an object, each leg of the conditional can be turned into an overriding method of a subclass associated with the object type. The original method becomes abstract.*



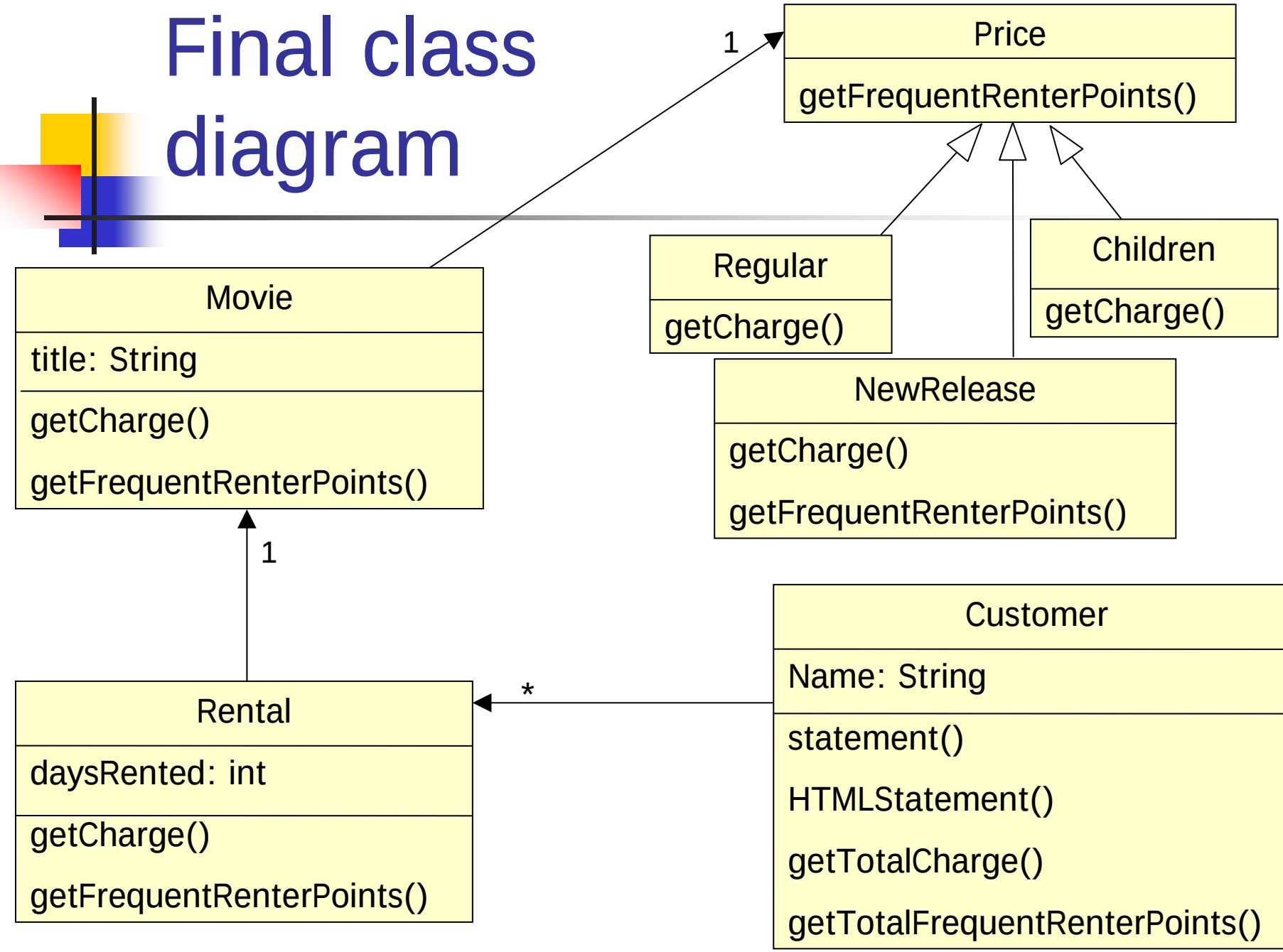
Replacing `getCharge`

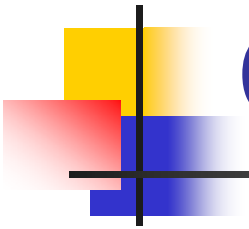
```
abstract class Price {
  abstract double getCharge(int daysRented);
}
class ChildrenPrice extends Price {
  public double getCharge(int daysRented) {
    double result = 1.5;
    if (daysRented > 3)
       result += (daysRented - 3) * 1.5;
    return result;
  }
}
class NewReleasePrice extends Price {
  public double getCharge(int daysRented) {
    return daysRented * 3;
  }
}
class RegularPrice extends Price {
  public double getCharge(int daysRented) {
    double result = 2;
    if (daysRented > 2)
       result += (daysRented - 2) * 1.5;
    return result;
  }
}
```

Replacing `getFrequentRenterPoints`

```
abstract class Price {
  abstract double getCharge(int daysRented);
  public int getFrequentRenterPoints(int daysRented) {
      return 1;
  }
}
class NewReleasePrice extends Price {
  ...
  public int getFrequentRenterPoints(int daysRented) {
    return (daysRented > 1) ? 2 : 1;
  }
}
class Movie {
  ...
  public double getCharge(int daysRented) {
    return _price.getCharge(daysRented);
  }
  public int getFrequentRenterPoints(int daysRented) {
    return _price.getFrequentRenterPoints(daysRented);
  }
}
```

Final class diagram





Conclusions

- Refactoring leads to better-distributed responsibilities and code that is easier to change.
- The overall organization is improved.
- The original code can be step by step migrated.
- Changing any behaviour, adding new functionalities, or adding extra type-dependent behaviour, become much easier tasks.