Software Engineering

Implementation

Prof. Dr. Peter Müller

Software Component Technology

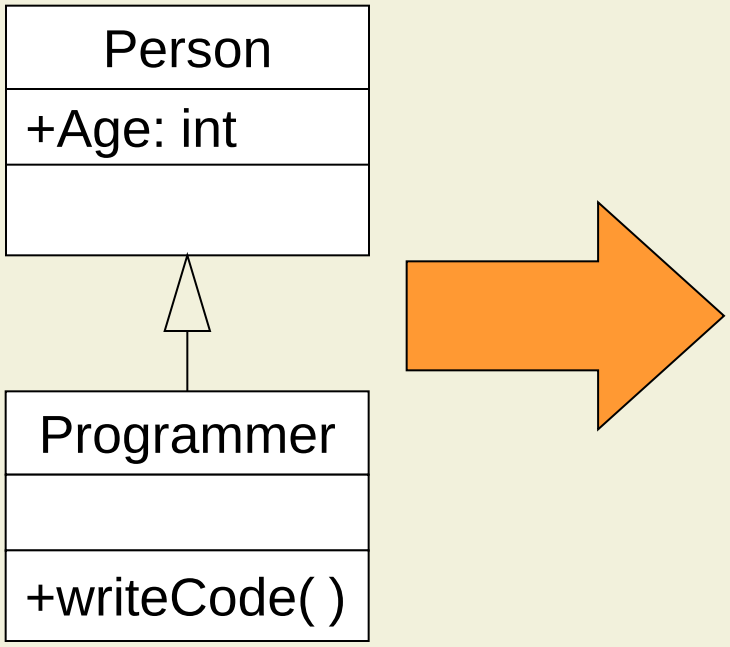
The slides in this section are partly based on the courses
“Software Engineering I” by Prof. Bernd Brügge, TU München and
“Software Engineering” by Prof. Jan Vitek, Purdue University

Summer Semester 06

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

6. Implementation

Implementation of UML Models in Java




```
class Person {
  private int age;

  public void setAge( int a )
    { age = a; }
  public int getAge( )
    { return age; }
}
```

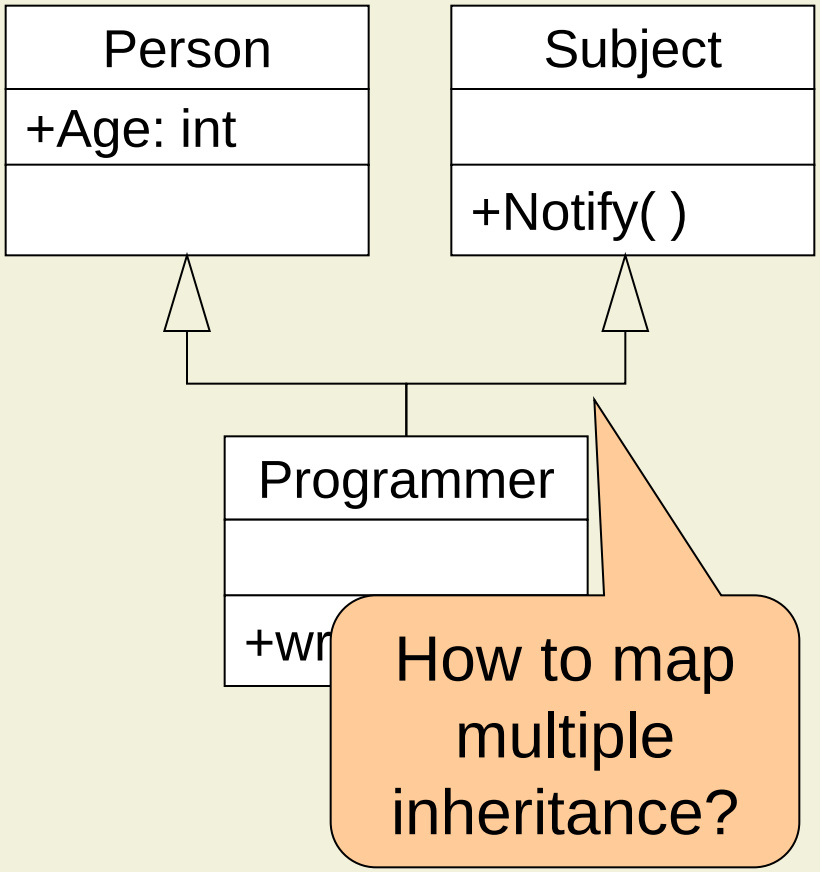
```
class Programmer extends Person {
  public void writeCode( )
    { … }
}
```

Model-Driven Development: Idea

- Work on the level of design models
- **Generate code** automatically (forward engineering)

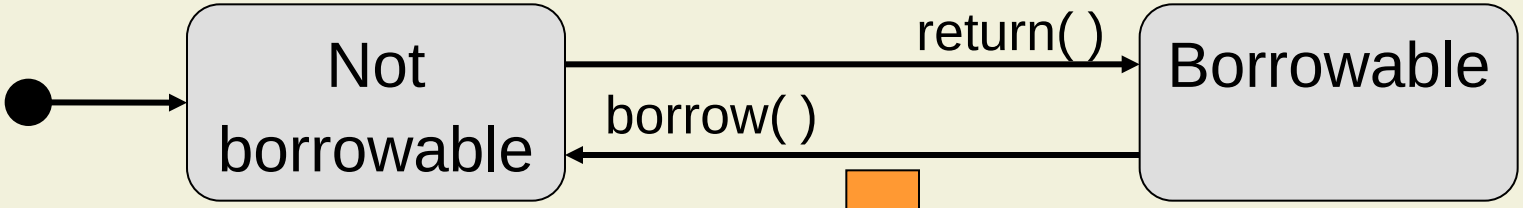
- Advantages
  - Supports many implementation platforms
  - Frees programmers from recurring activities
  - Leads to uniform code
  - Useful to enforce coding conventions (e.g., getters and setters)
  - Models are not mere documentation

Problem: Abstraction Mismatch



- UML models may use different abstractions than the programming language
- Model should not depend on implementation language
- Models cannot always be mapped directly to code

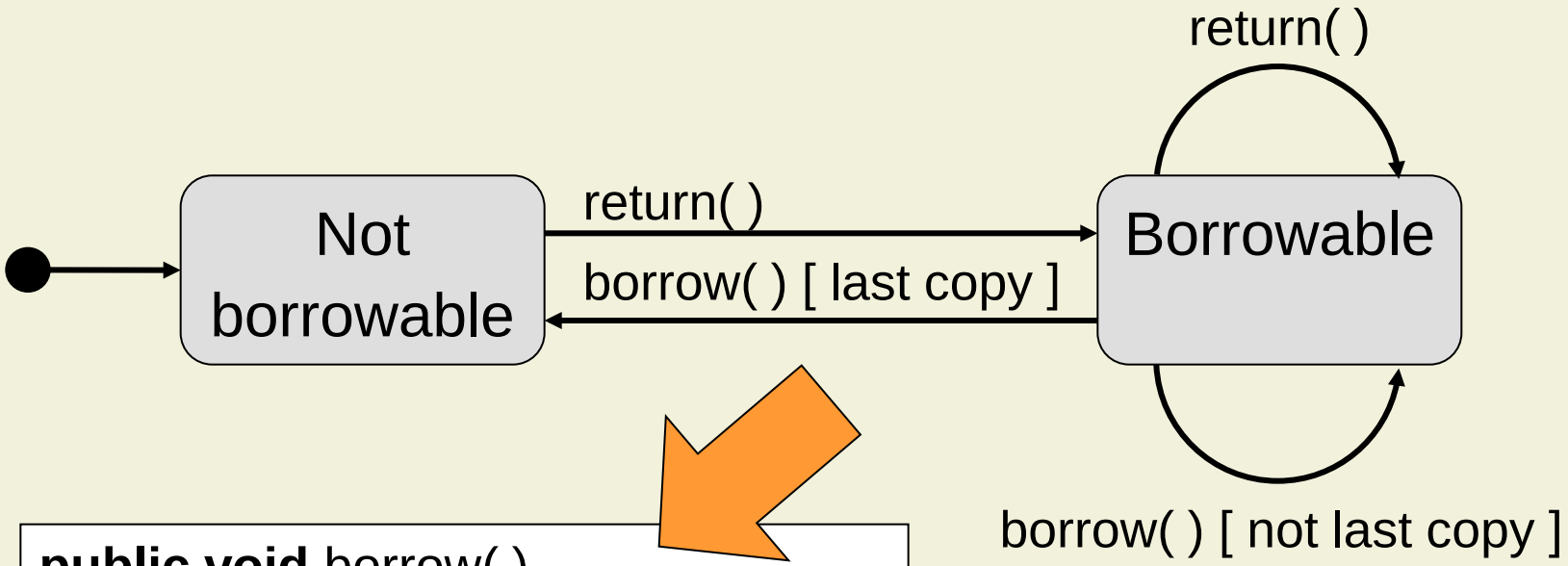
Problem: Specifications are Incomplete




```
class Book {
  private boolean borrowable;
  public Book( )
    { borrowable = false; }
  public void return( )
    requires borrowable == false;
    { borrowable = true; }
  public void borrow( )
    requires borrowable == true;
    { borrowable = false; }
}
```

Where is the interesting behavior?

Problem: Specifications are Informal




```
public void borrow( )
  requires borrowable == true;
  requires "last copy" || "not last copy";
{
  if ( "last copy" )  borrowable = false;
  else ;
}
```

How to map informal specifications?

Problem: Switching between Models and Code

- Code has to be **changed manually**
  - Add interesting behavior
  - Clarify informal specifications
  - Implement incomplete specifications
- … or do you want to **specify the whole program behavior formally** in UML and OCL?
  - All sequence charts, state charts, etc.
- Modification of code requires complicated synchronization between code and models

Model-Driven Development: Reality

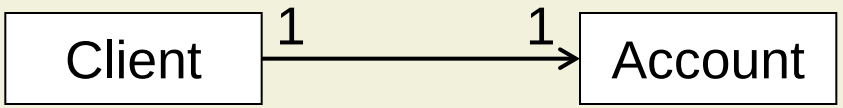
- Model-driven development is a buzzword
- **It does not work in practice**
- Code generation works for very **basic properties**
  - Class diagrams, simple state charts
- Interesting code is still **implemented manually**
- Problems
  - Maintaining code that has no models (reverse-engineering)
  - Once code has been modified manually, going back to the model is difficult (or impossible)

6. Implementation

Classes and Inheritance

- Classes may be split into interfaces and implementation classes
- Attributes should be non-public
  - Generate getters and setters with appropriate visibility
- Methods are straightforward
- Inheritance can be mapped to
  - Inheritance (**extends** in Java)
  - Subtyping without inheritance (**implements** in Java)
  - Aggregation and delegation

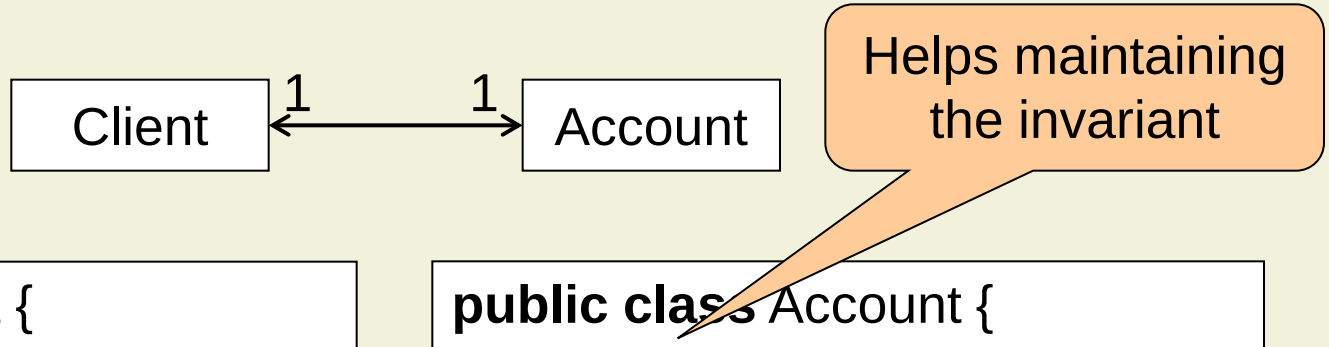
Unidirectional One-to-One Associations




```
public class Client {
  private Account account;
  invariant account != null;
  public Client( ) {
    account = new Account();
  }

  public Account getAccount() {
    return account;
  }
}
```

No invariant for 0..1 association

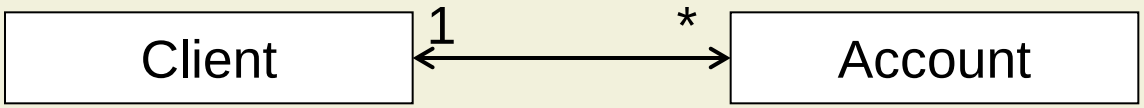
Bidirectional One-to-One Associations




```
public class Client {
  private final Account account;
  invariant account != null &&
      account.owner == this;
  public Client( ) {
    account = new Account( this );
  }
  public Account getAccount( ) {
    return account;
  }
}
```

```
public class Account {
  private final Client owner;
  invariant owner != null &&
      owner.account == this;
  public Account( Client owner ) {
    this.owner = owner;
  }
  public Client getOwner( ) {
    return owner;
  }
}
```

Bidirectional One-to-Many Associations



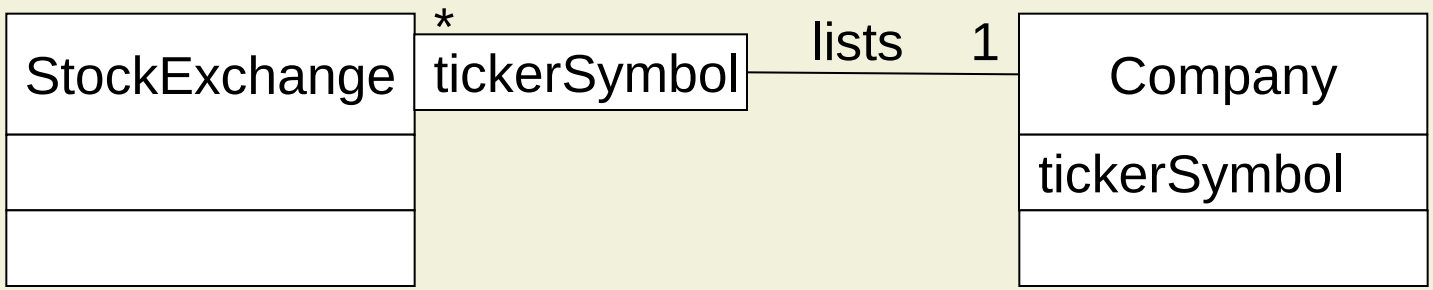

```
public class Client {
 private final Set<Account> accounts;
 public Client( ) {
  account =
      new HashSet<Account>( );
 }
 public void add( Account a )
  requires a != null;
{
  accounts.add( a );
  a.setOwner( this );
 }
}
```

```
public class Account {
 private Client owner;
 public void setOwner( Client n )
  requires n != null;
{
 if ( owner != n ) {
  owner.removeAccount( this );
  owner = n;
 }
```

Invariant is a bit complicated

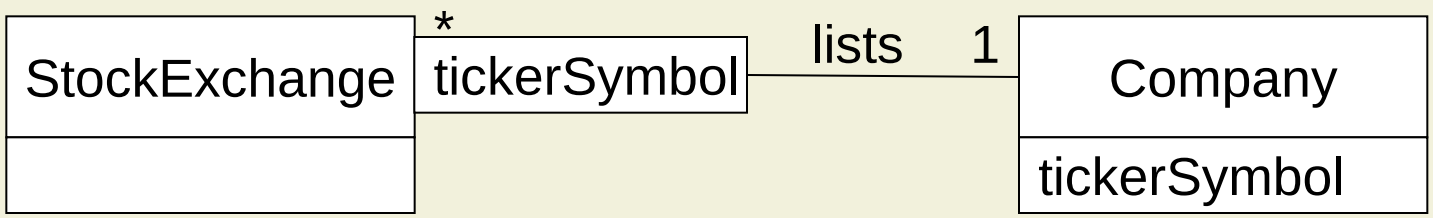
Repetition: Qualified Associations

- For each ticker symbol, a stock exchange lists exactly one company



- Qualifiers reduce the multiplicity of associations

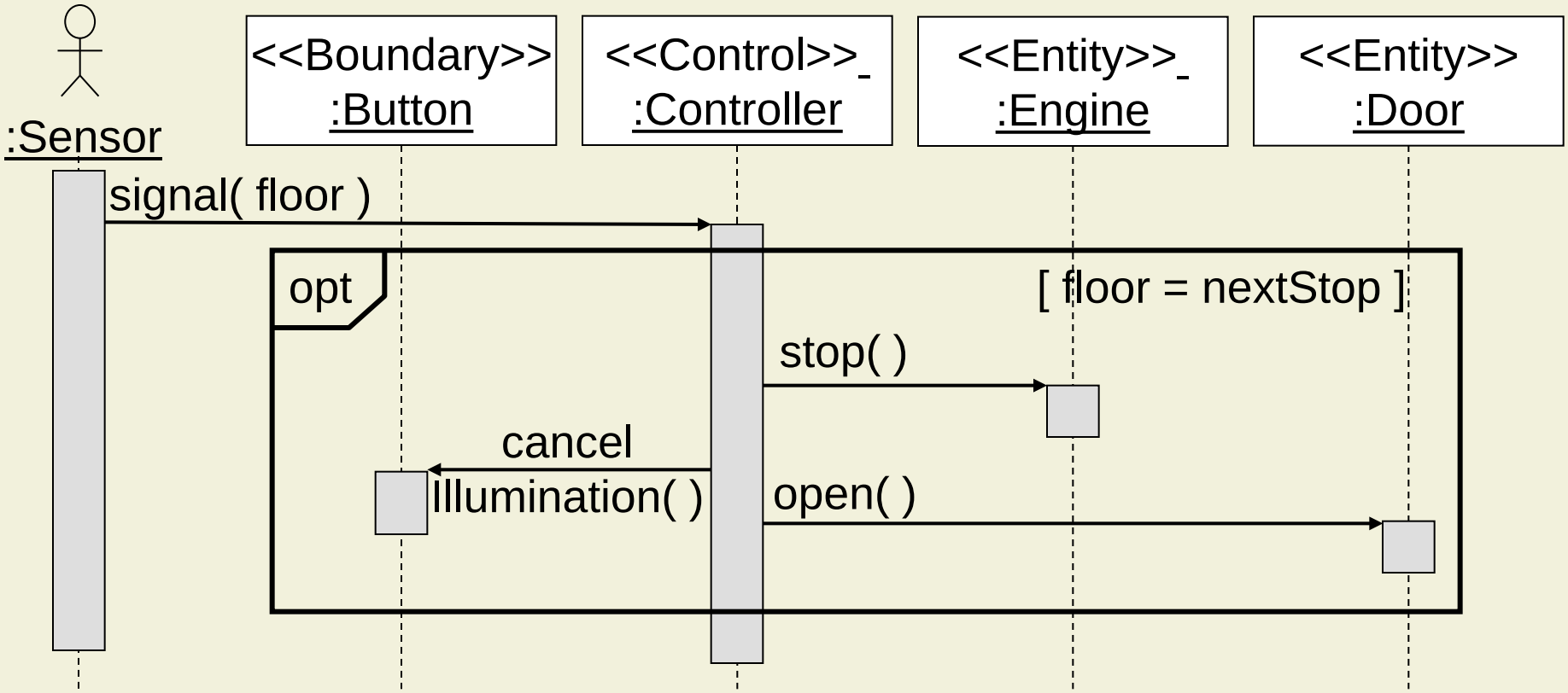
Bidirectional Qualified Associations



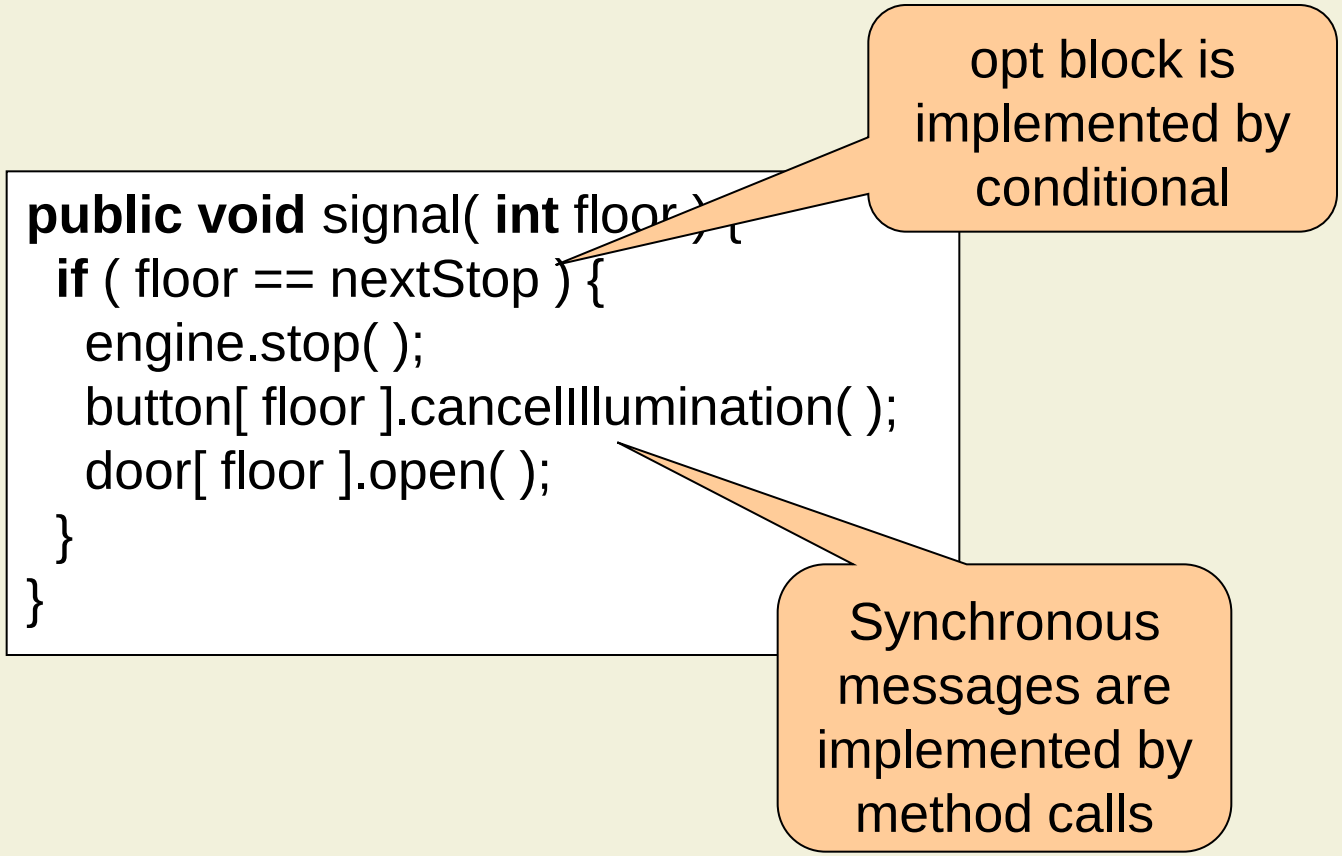

```
public class StockExchange{
  private Map<String, Company> companies;

  public void addCompany( Company c )
    requires c != null && !companies.containsKey( c.tickerSymbol );
  {
    companies.put( c.tickerSymbol, c );
    c.addStockExchange( this );
  }
}
```

6. Implementation

Sequence Diagram

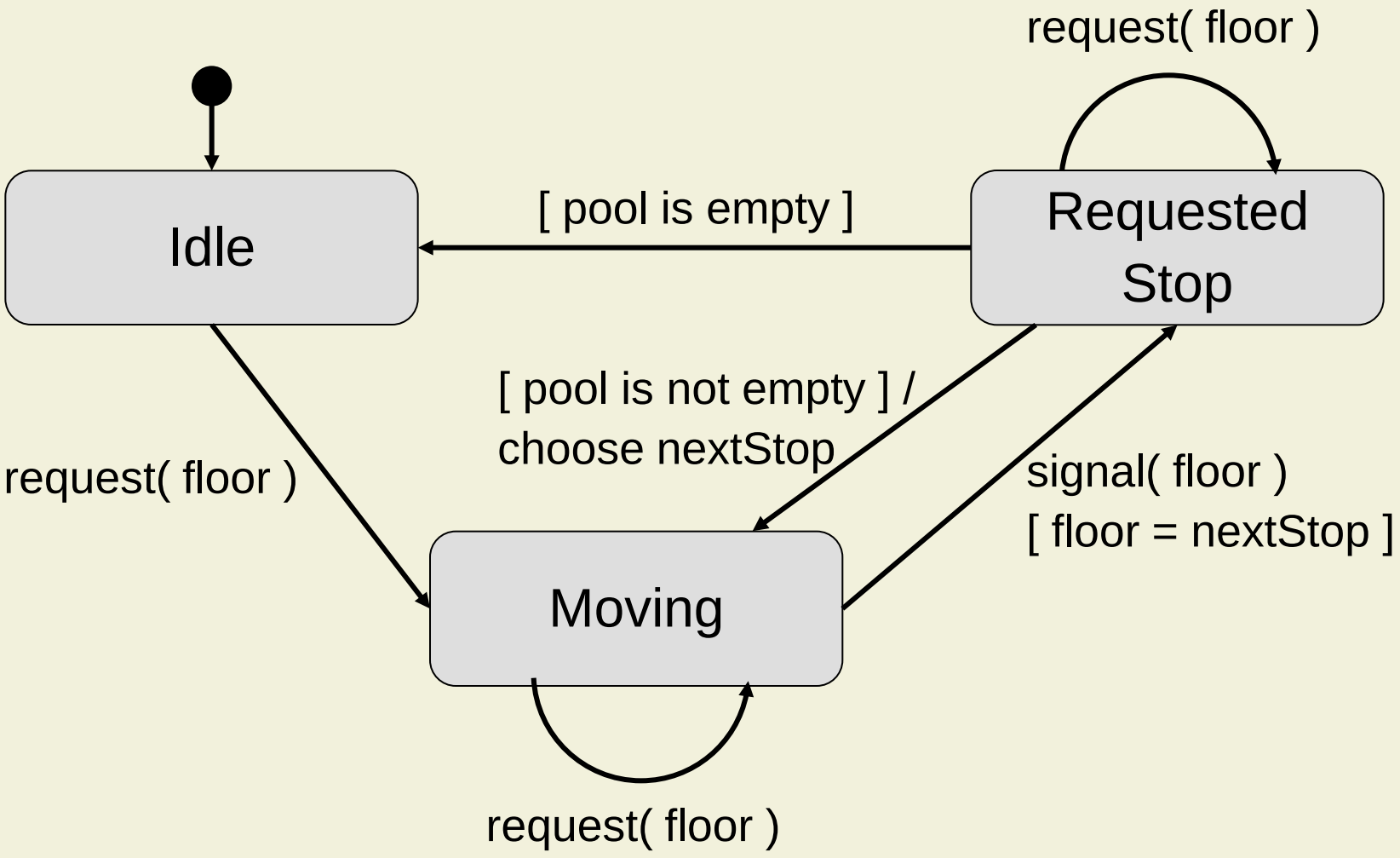


Mapping Sequence Diagrams



6. Implementation

State Diagrams



State Diagrams to Switch Statements

```
public void request( int floor ) throws … {
  switch( state ) {
    case Idle:            state = Moving; break;
    case Moving: break;
    case RequestedStop:
      if ( pool.IsEmpty( ) ) state = Idle;
      else {
        nextStop = pool.Choose( );
        state = Moving;
      }
      break;
    default: throw new UnexpectedStateException( );
  }
}
```

Introduce state variable for current state

Transition

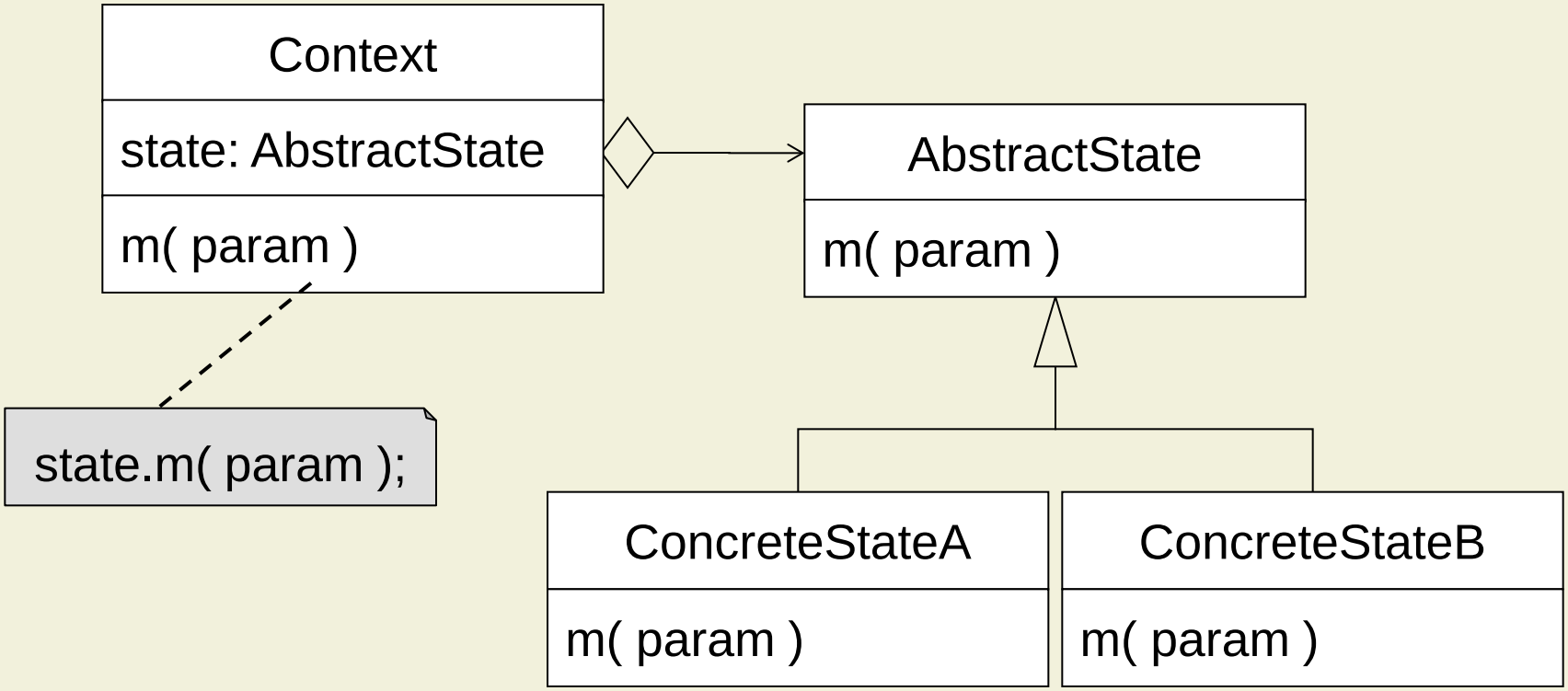
Check condition of transition

Perform action

Spontaneous transition from RequestedStop

Illegal state or message

State Pattern: Structure



State Diagrams to State Pattern

```
class IdleState extends ControllerState {
  public void request( int floor, Controller controller ) {
    controller.state = new MovingState( );
  }
}
```

switch statements replaced by dynamic method binding

```
class RequestedStopState extends ControllerState {
  public void request( int floor, Controller controller  ) {
    if ( controller.pool.IsEmpty( ) )
      controller.state = new IdleState( );
    else {
        controller.nextStop = controller.pool.Choose( );
        controller.state = new MovingState( );
    }
  }
}
```

Transition

6. Implementation

Mapping Contracts

- Many object-oriented languages do not include built-in support for contracts
  - But there are research compilers for JML, Spec#, etc.
- Possible workaround mechanisms
  - Assert statements (e.g., Java 5)
  - Conditionals plus exceptions

- Keep the contracts in the source code, not in the design model
  - More likely to be updated when source code changes

Implementing a Contract

- Check precondition
  - Before the beginning of the method
  - Throw exception if the precondition is false
- Check postcondition
  - At the end of the method
  - Throw exception if the postcondition is false
- Check invariant
  - At the same time as postconditions
- Deal with inheritance
  - Put checking code into separate methods that can be called from subclasses

Mapping Contracts: Examples

```
void add( Account a )
  requires a != null;
{
  accounts.add( a );
  a.setOwner( this );
}
```

<<precondition>>
a != null

```
void add( Account a )
{
  assert a != null;
  accounts.add( a );
  a.setOwner( this );
}
```

```
void add( Account a )
{
  if ( a == null )
    throw new IllegalArgumentException( );
  accounts.add( a );
  a.setOwner( this );
}
```

Heuristics for Implementing Contracts

- Be pragmatic, if you don't have enough time, change your tests in the following order
- Omit checking code for postconditions and invariants
  - Often redundant, if you are confident that the code accomplishes the functionality of the method
  - Not likely to detect many bugs unless written by a separate tester
- Omit the checking code for private and protected methods

6. Implementation

Quality Characteristics and Performance

- Performance is most important non-functional requirement, but:

- "*More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.*"
  [W. A. Wulf]

- "*Premature optimization is the root of all evil.*"
  [D. Knuth]

Design for Performance

- A **simple design** is likely to be both correct and fast
  - Use efficient algorithms
- **Avoid** inherently **expensive operations** whenever possible
  - System calls
  - Synchronization
  - Network message exchanges
  - Disk accesses (use buffered I/O)
  - Memory allocation and copying (use flyweight pattern)

Premature Optimization

- Jackson's Rules of Optimization:
  Rule 1. Don't do it.
  Rule 2. (for experts only) Don't do it yet.
  [M. A. Jackson]

- After creating a performance-aware design, forget all about efficiency until testing is completed
- During implementation, **focus on clarity and correctness**, not on performance

Performance Tuning

- Performance tuning of a working program

- Space versus time
  - The usual performance objective is maximum speed in a reasonable amount of space

Performance Tuning: Myths

- Reducing the lines of source code improves performance of machine code
  - No predictable relationship between number of lines of source code and ultimate performance

- Certain operations are faster or smaller than others
  - Performance varies significantly depending on the language, choice of compiler, libraries, operating system, hardware (especially CPU), amount of memory, etc.
  - The only way to reliably determine what is smaller or faster is to measure

Performance Tuning: Myths (cont'd)

- You should optimize as you go
  - Other qualities are usually more important and harder to fix later
  - It is very hard to correctly identify bottlenecks during coding

- A fast program is just as important as a correct one
  - Without correctness, speed is meaningless

6. Implementation

Refactoring: Definition

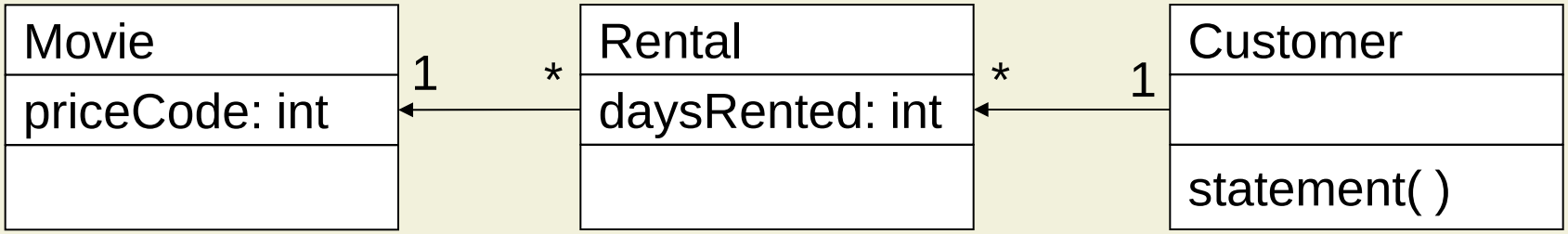
- Refactoring (noun):
  A change made to the internal structure of software to make it
  - *Easier to understand and*
  - *Cheaper to modify*
  - *Without changing its observable behavior*

- Refactor (verb):
  To restructure software by applying a series of refactorings

Refactoring: Example

- "a program to calculate and print a statement of a customer's charges at a video store"
  - Price depends on how long the movie is rented and the category of the movie
  - Also compute frequent renter points

- Initial class diagram



Movie Class

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

  private String _title;
  private int _priceCode;

  public Movie( String title, int priceCode ) {
    _title = title; _priceCode = priceCode;
  }
  public int getPriceCode( )              { return _priceCode; }
  public void setPriceCode( int arg ) { _priceCode = arg; }
  public String getTitle( )               { return _title; }
}
```

Rental and Customer Class

```
public class Rental {
  private Movie _movie;  private int _daysRented;
  public Rental( Movie movie, int daysRented ) {
    _movie = movie; _daysRented = daysRented;
  }
  public int getDaysRented( )      { return _daysRented; }
  public Movie getMovie( )         { return _movie; }
}
```

```
public class Customer {
  private String _name;
  private Vector _rentals = new Vector( );
  public Customer( String name )          { _name = name; }
  public void addRental( Rental arg )     { _rentals.addElement( arg ); }
  public String getName( )                { return _name; }
  …
```

Customer.statement Method

```
public String statement( ) {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rental.elements( );
  String result = “Rental Record for “ + getName( ) + “\n”;
  while ( rentals.hasMoreElements( ) ) {
    double thisAmount = 0;
    Rental each = ( Rental ) rentals.nextElement( );

    // determine amounts for each line
    switch ( each.getMovie( ).getPriceCode( ) ) {
      case Movie.REGULAR:
        thisAmount += 2;
        if ( each.getDaysRented( ) > 2 )
          thisAmount += ( each.getDaysRented( ) – 2 ) * 1.5;
        break;
```

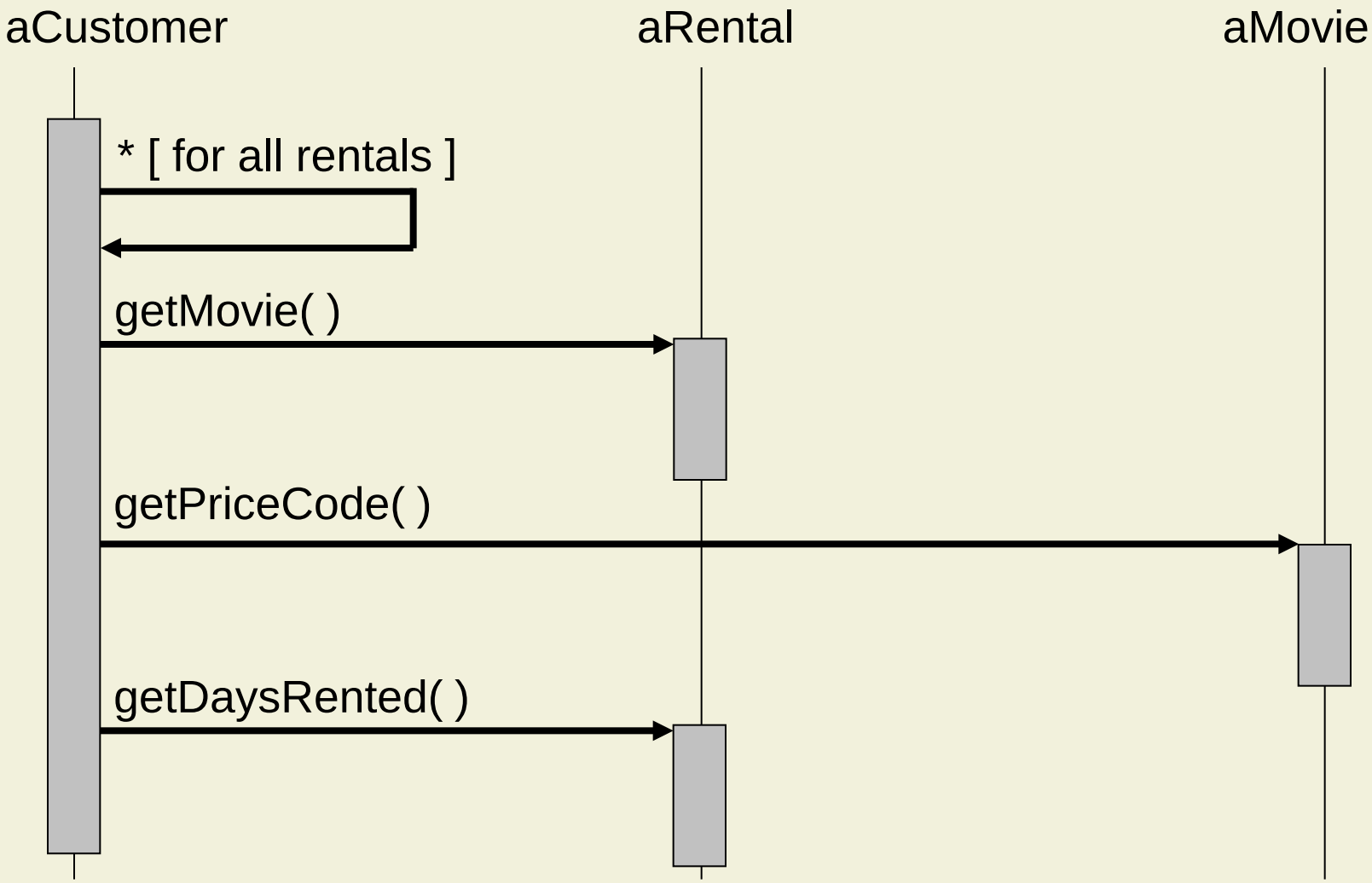
Customer.statement Method (cont'd)

```
    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;
  } /* switch */

  // add frequent renter points
  frequentRenterPoints++;
  // add bonus for a two day new release rental
  if ( ( each.getMovie( ).getPriceCode( ) == Movie.NEW_RELEASE ) &&
       each.getDaysRented( ) > 1 )
    frequentRenterPoints++;
```

Customer.statement Method (cont'd)

```
    //show figures for this rental
    result += "\t" + each.getMovie( ).getTitle( )+ "\t";
    result += String.valueOf( thisAmount ) + "\n";
    totalAmount += thisAmount;
  } /* while */

  // add footer lines
  result += "Amount owed is "+String.valueOf( totalAmount ) + "\n";
  result += "You earned " + String.valueOf( frequentRenterPoints );
  result += "frequent renter points\n";
  return result;
}
```

Sequence Diagram for statement Method



Refactoring in Action

- Refactoring: Problem Statement
  - Add an htmlStatement method which returns a customer statement string containing html tags
  - Change the way movies are classified
  - Change frequent renter points and charging

- Refactoring: Step 1
  - Write a test suite!
  - Refactoring should not affect the outcome of tests
  - The test suite must exercise the published interface of the classes

Step 2: Extract Method

Method is overly long

```
public String statement( ) {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rental.elements( );
  String result = “Rental Record for “ + getName( ) + “\n”;
  while ( rentals.hasMoreElements( ) ) {
    double thisAmount = 0;
    Rental each = ( Rental ) rentals.nextElement( );

    // determine amounts for each line
    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;
    } /* switch */
```

```
uent renter points
rPoints++;
us for a two day new release rental
    if ( ( each.getMovie( ).getPriceCode( ) ==
        Movie.NEW_RELEASE ) &&
        each.getDaysRented( ) > 1 )
      frequentRenterPoints++;
    //show figures for this rental
    result += “\t” + each.getMovie( ).getTitle( )+ “\t”;
    result += String.valueOf( thisAmount ) + “\n”;
    totalAmount += thisAmount;
  } /* while */

  // add footer lines
  result += “Amount owed is “+String.valueOf( totalAmount ) +
“\n”;
  result += “You earned “ + String.valueOf( frequentRenterPoints
);
  result += “frequent renter points\n”;
  return result;
}
```

Apply “Extract Method”

Step 2: Extract Method (cont'd)

```
public int amountFor( Rental each ) {
  int 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;
  } /* switch */
  return thisAmount;
}
```

Local variable becomes parameter

Local variable becomes result

Step 2: Extract Method (cont'd)

```
public String statement( ) {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rental.elements( );
  String result = "Rental Record for " + getName( ) + "\n";
  while ( rentals.hasMoreElements( ) ) {
    double thisAmount = 0;
    Rental each = ( Rental ) rentals.nextElement( );

    // determine amounts for each line
    thisAmount = amountFor( each );

    // add frequent renter points
    frequentRenterPoints++;
    // add bonus for a two day new release rental
    if ( ( each.getMovie( ).getPriceCode( ) ==
        Movie.NEW_RELEASE ) &&
        each.getDaysRented( ) > 1 )
      frequentRenterPoints++;
    //show figures for this rental
    result += "\t" + each.getMovie( ).getTitle( )+ "\t";
    result += String.valueOf( thisAmount ) + "\n";
    totalAmount += thisAmount;
  } /* while */

  // add footer lines
  result += "Amount owed is "+String.valueOf( totalAmount ) +
"\n";
  result += "You earned " + String.valueOf( frequentRenterPoints
);
  result += "frequent renter points\n";
  return result;
}
```

Step 3: Test

```
public 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;
  } /* switch */
  return thisAmount;
}
```

Oops, there is a bug

Variable names not helpful

Step 4: Rename Local

```
public 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;
  } /* switch */
  return result;
}
```

Method does not use data from Customer

Step 5: Move Method

```
class Rental {
  public double getCharge( ) {
    double result = 0;
    switch ( this.getMovie( ).getPriceCode( ) ) {
      case Movie.REGULAR:
        result += 2;
        if ( this.getDaysRented( ) > 2 ) result += ( this.getDaysRented( ) – 2 ) * 1.5;
        break;
      case Movie.NEW_RELEASE:
        result += this.getDaysRented( ) * 3;
        break;
      case Movie.CHILDREN:
        result += 1.5;
        if ( this.getDaysRented( ) > 3 ) result += ( this.getDaysRented( ) – 3 ) * 1.5;
        break;
    } /* switch */
    return result;
  }
  …
```

Make sure super- and subclasses do not contain a getCharge method

Adapt references to data

Step 5: Move Method (cont'd)

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

Delegate calls to new method

Useless method

Step 6: Inline Method

```
public String statement( ) {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rental.elements( );
  String result = “Rental Record for “ +
getName( ) + “\n”;
  while ( rentals.hasMoreElements( ) ) {
    double thisAmount = 0;
    Rental each = ( Rental )
rentals.nextElement( );

    // determine amounts for each line
    thisAmount = each.getCharge( );

    // add frequent renter points
    frequentRenterPoints++;
```

```
  // add bonus for a two day new release rental
  if ( ( each.getMovie( ).getPriceCode( ) ==
      Movie.NEW_RELEASE ) &&
      each.getDaysRented( ) > 1 )
                                    ++;
                                    ental
                                    Movie( ).getTitle( )+

                                    ( thisAmount ) + “\n”;
                                    talAmount += thisAmount;
  } /* while */


                                    ed is “+String.valueOf(

                                    “ + String.valueOf(
                                    );
  result += “frequent renter points\n”;
  return result;
```

Make sure method is not dynamically-bound

Replace all calls and remove method

thisAmount is assigned only once

Step 7: Replace Temp with Query

```
public String statement( ) {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rental.elements();
  String result = “Rental Record for “ +
    getName( ) + “\n”;
  while ( rentals.hasMoreElements( ) ) {
    Rental each = ( Rental )
      rentals.nextElement( );

    // add frequent renter points
    frequentRenterPoints++;
    // add bonus for a two day new release
    if ( ( each.getMovie( ).getPriceCode( )==
        Movie.NEW_RELEASE ) &&
        each.getDaysRented( ) > 1 )
      frequentRenterPoints++;

    //show figures for this rental
    result += “\t”+each.getMovie( ).getTitle( )+“\t”;
    result += String.valueOf( each.getCharge( ) )
+ “\n”;
    totalAmount += each.getCharge( );
  } /* while */

  // add footer lines
  result += “Amount owed is “+String.valueOf(
totalAmount ) + “\n”;
  result += “You earned “ + String.valueOf(
frequentRenterPoints );
  result += “frequent renter points\n”;
  return result;
}
```

Frequent renter bonus is a separate computation

Step 8: Extract Method

```
public class Rental {
  public int getFrequentRenterPoints( ) {
    if ( ( this.getMovie( ).getPriceCode( ) == Movie.NEW_RELEASE ) &&
         this.getDaysRented( ) > 1 )
      return 2;
    else
      return 1;
  }
  …
}
```

Step 8: Extract Method (cont'd)

```
public String statement( ) {
  double totalAmount = 0;
  int frequentRenterPoints = 0;
  Enumeration rentals = _rental.elements();
  String result = "Rental Record for " +
    getName( ) + "\n";
  while ( rentals.hasMoreElements( ) ) {
    Rental each = ( Rental )
      rentals.nextElement( );

    // add frequent renter points
    frequentRenterPoints +=
          each.getFrequentRenterPoints();

    //show figures for this rental
    result += "\t"+each.getMovie( ).getTitle( )+"\t";
    result += String.valueOf( each.getCharge( ) )
+ "\n";
    totalAmount += each.getCharge( );
  } /* while */

  // add footer lines
  result += "Amount owed is "+String.valueOf(
totalAmount ) + "\n";
  result += "You earned " + String.valueOf(
requentRenterPoints );
  result += "frequent renter points\n";
  return result;
```

Local variables clutter up code

Step 9: Replace Temp with Query

```
class Customer {
  private double getTotalCharge( ) {
    double result = 0;
    Enumeration rentals = _rentals.elements( );
    while ( rentals.hasMoreElements( ) ) {
      Rental each = ( Rental ) rentals.nextElement( );
      result += each.getCharge( );
    }
    return result;
  }
  …
}
```

Step 9: Replace Temp with Query (cont'd)

```
public String statement( ) {
  int frequentRenterPoints = 0;
  Enumeration rentals = _rental.elements();
  String result = "Rental Record for " +
    getName( ) + "\n";
  while ( rentals.hasMoreElements( ) ) {
    Rental each = ( Rental )
      rentals.nextElement( );

    // add frequent renter points
    frequentRenterPoints +=
          each.getFrequentRenterPoints();

    //show figures for this rental
    result += "\t"+each.getMovie( ).getTitle( )+"\t";
    result += String.valueOf( each.getCharge( ) )
+ "\n";
  } /* while */

  // add footer lines
  result += "Amount owed is "+String.valueOf(
getTotalCharge( ) ) + "\n";
  result += "You earned " + String.valueOf(
frequentRenterPoints );
  result += "frequent renter points\n";
  return result;
}
```

Local variables clutter up code

Step 10: Replace Temp with Query

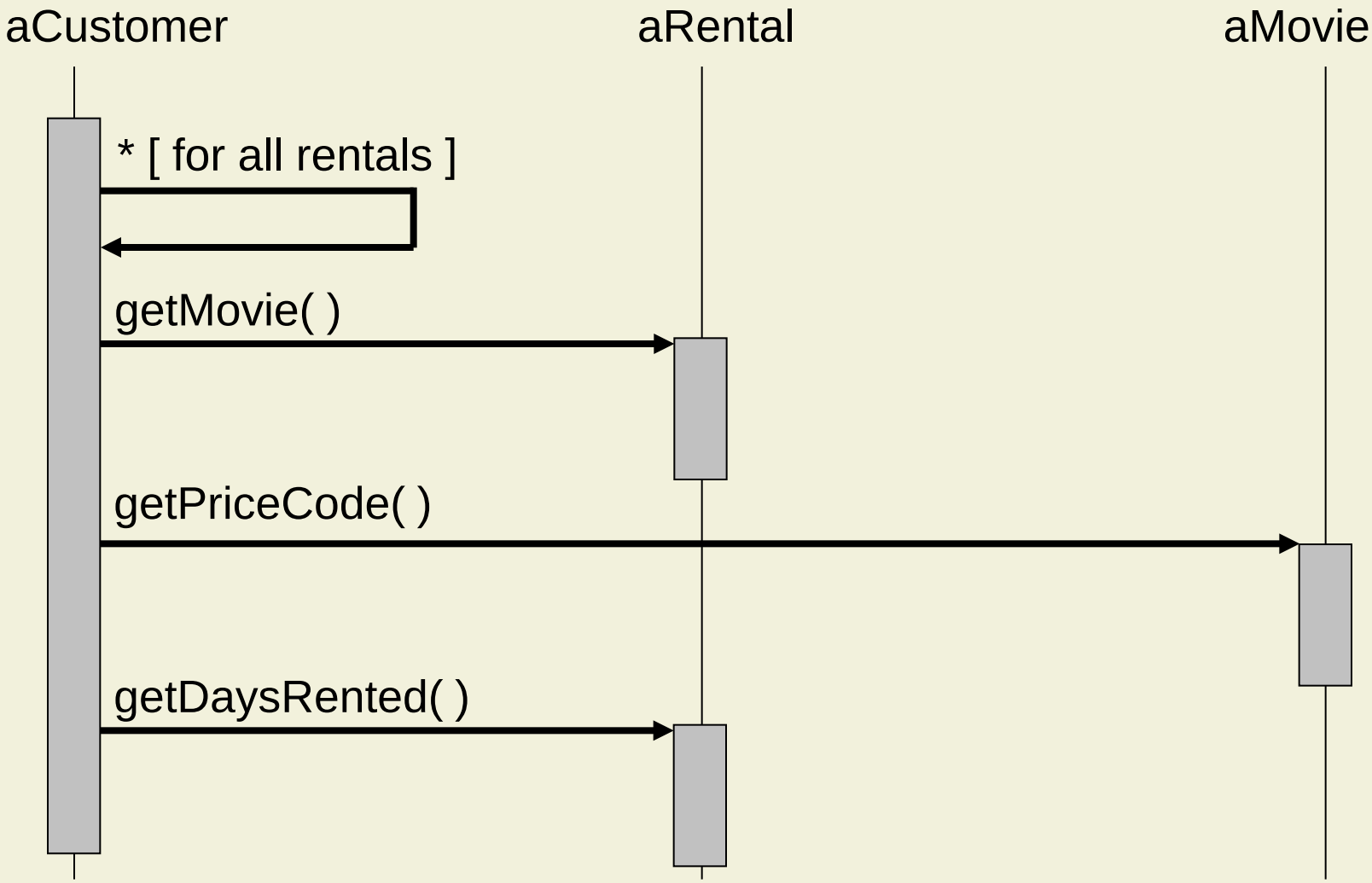
```
class Customer {
  private int getTotalRenterPoints( ) {
    int result = 0;
    Enumeration rentals = _rentals.elements( );
    while ( rentals.hasMoreElements( ) ) {
      Rental each = ( Rental ) rentals.nextElement( );
      result += each.getFrequentRenterPoints( );
    }
    return result;
  }
  …
}
```

Step 10: Replace Temp with Query (cont'd)

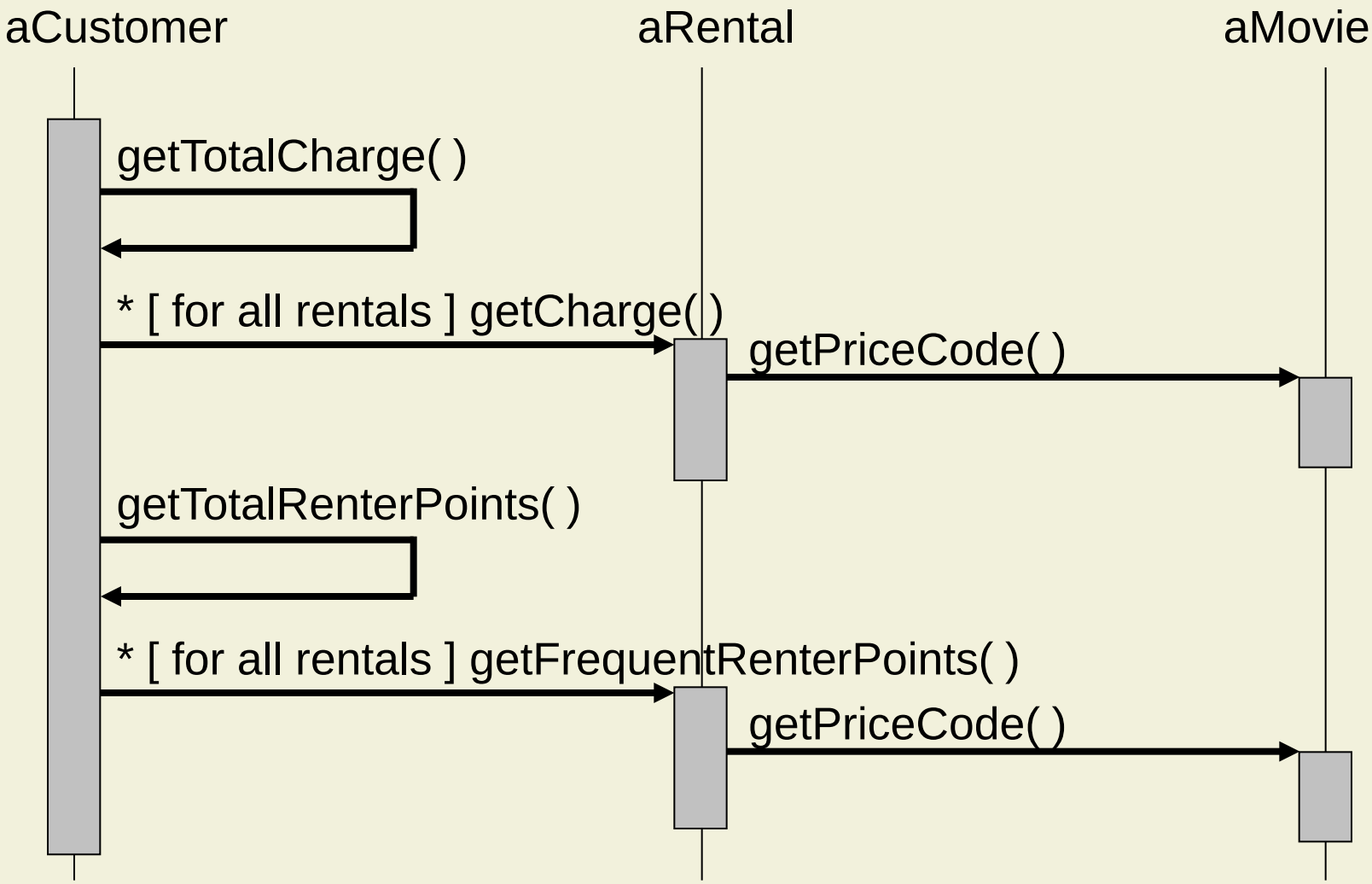
```
public String statement( ) {
  Enumeration rentals = _rental.elements();
  String result = "Rental Record for " +
    getName( ) + "\n";
  while ( rentals.hasMoreElements( ) ) {
    Rental each = ( Rental )
      rentals.nextElement( );

    //show figures for this rental
    result += "\t"+each.getMovie( ).getTitle(
)+"\t";
    result += String.valueOf(
each.getCharge( ) ) + "\n";
  } /* while */
```

```
  // add footer lines
  result += "Amount owed is "+String.valueOf(
getTotalCharge( ) ) + "\n";
  result += "You earned " + String.valueOf(
getTotalRenterPoints( ) );
  result += "frequent renter points\n";
  return result;
}
```

Initial Sequence Diagram for statement Method



Revised Sequence Diagram for statement



Observations

- Most refactorings reduce code size, but this is not necessarily the case
  - The point is to make code easier to modify and more readable
- Performance suffers by running the same loop three times, or does it?
  - Profile the program and find the answer

Making Changes

```
class Customer {
  public String htmlStatement( ) {
    Enumeration rentals = _rental.elements();
    String result = “<H1>Rental Record for<EM> “ + getName( ) + “<EM></H1><P>\n”;
    while ( rentals.hasMoreElements( ) ) {
      Rental each = ( Rental ) rentals.nextElement( );
      //show figures for this rental
      result += each.getMovie( ).getTitle( ) +  “: ”;
      result += String.valueOf( each.getCharge( ) ) + “<BR>\n”;
    }
    // add footer lines
    result += “<P>Amount owed is<EM> “ + String.valueOf(getTotalCharge()) +“</EM><P>\n”;
    result += “You earned<EM>“ + String.valueOf( getTotalRenterPoints( ) );
    result += “</EM> frequent renter points<P>\n”;
    return result;
  }
  …
}
```

The requested method can be added with minimal code duplication

New Functionality

- Getting ready to change the classification of the movies in the store
- Perhaps new classification, perhaps modification to existing
- Charging and frequent renting will be affected

Method Rental.getCharge

switch makes extension difficult

```
class Rental {
  public double getCharge( ) {
    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;
    } /* switch */
    return result;
  }
  …
```

Step 11: Move Method

```
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;
    } /* switch */
    return result;
  }
  …
```

Data from Rental passed as parameter

switch works on data from Movie

Step 11: Move Method (cont'd)

```
class Rental {
  public double getCharge( ) {
    return _movie.getCharge( _daysRented );
  }
  …
}
```

Method Rental.getFrequentRenterPoints

```
public class Rental {
  public int getFrequentRenterPoints( ) {
    if ( ( getMovie( ).getPriceCode( ) == Movie.NEW_RELEASE ) &&
         getDaysRented( ) > 1 )
      return 2;
    else
      return 1;
  }
  …
}
```

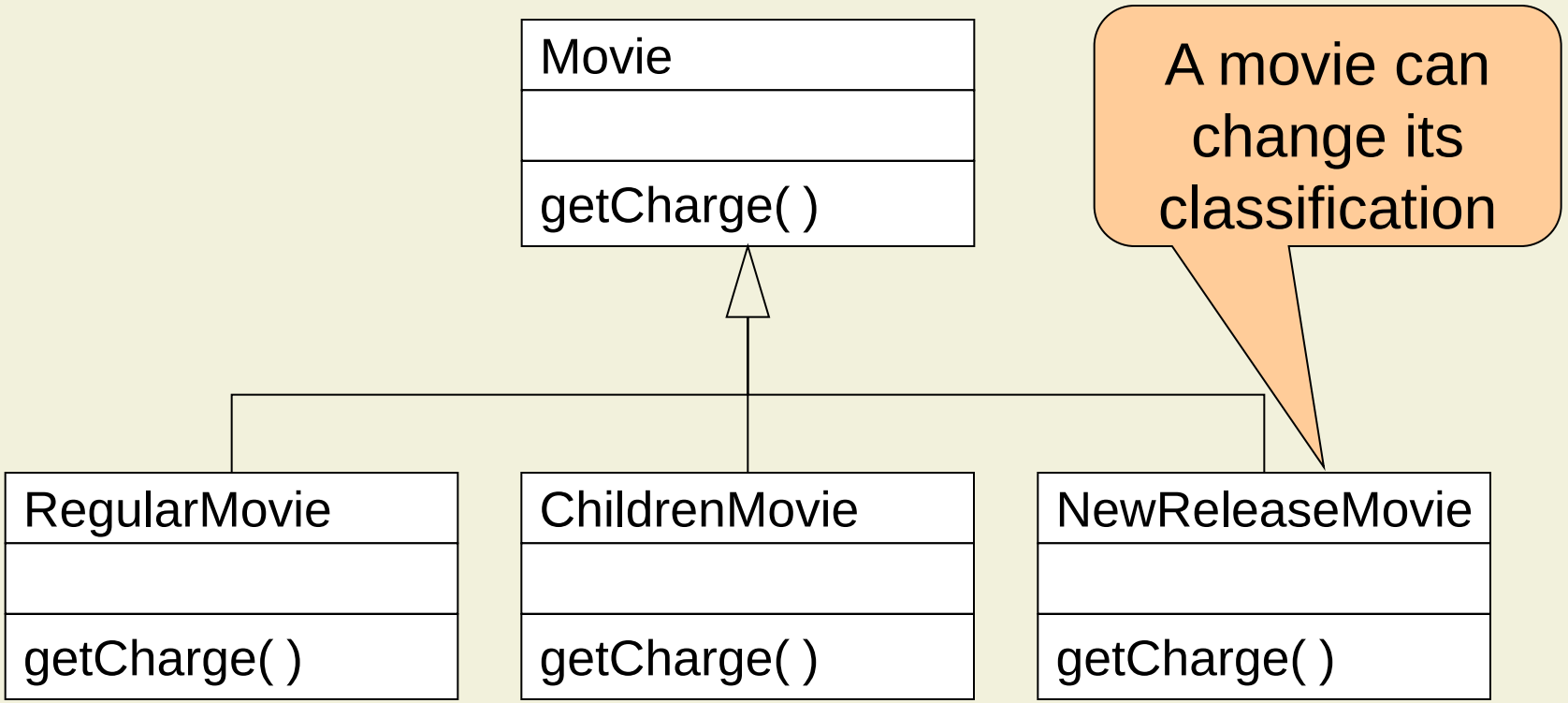
Works on data from Movie

Step 12: Move Method

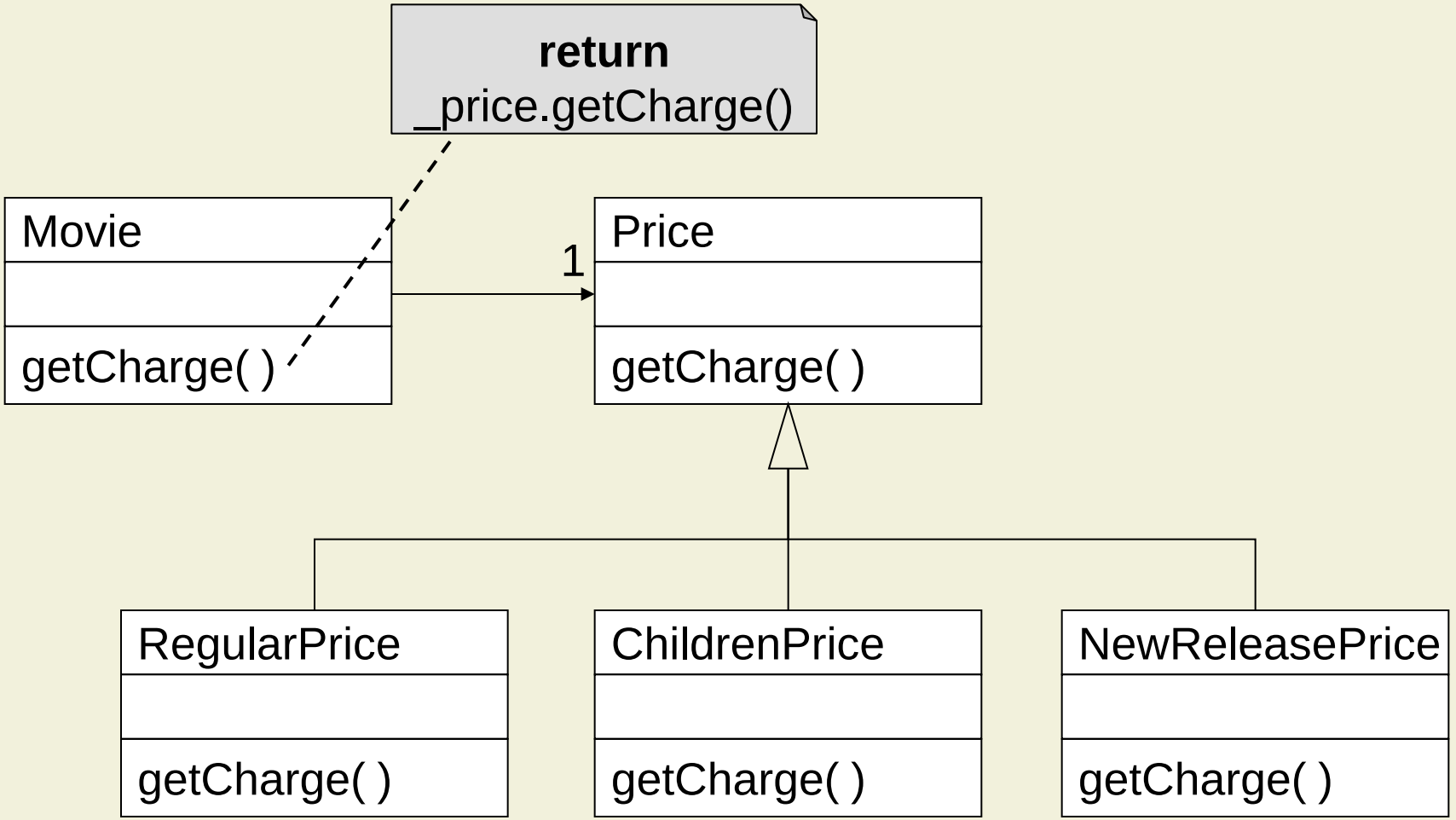
```
public class Movie {
  public int getFrequentRenterPoints( int daysRented ) {
    if ( getPriceCode( ) == NEW_RELEASE && daysRented > 1 )
      return 2;
    else
      return 1;
  }
  …
}
```

```
public class Rental {
  public int getFrequentRenterPoints( ) {
    return _movie.getFrequentRenterPoints( _daysRented );
  }
  …
}
```

Subtyping


Movie
getCharge( )
A movie can change its classification
RegularMovie
getCharge( )
ChildrenMovie
getCharge( )
NewReleaseMovie
getCharge( )

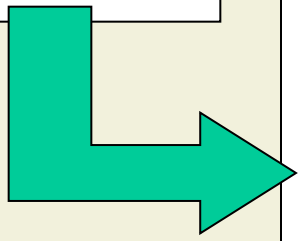
Subtyping



Step 13: Replace Type Code with State

```
class Movie {
  public Movie( String name,
                int priceCode ) {
    _name = name;
    _priceCode = priceCode;
  }
}
```

Apply “Self-Encapsulate Field”



```
class Movie {
  public Movie( String name,
                int priceCode ) {
    _name = name;
    setPriceCode( priceCode );
  }

  public int getPriceCode( ) {
    return _priceCode;
  }

  public void setPriceCode( int arg ) {
    _priceCode = arg;
  }
  ...
}
```

Step 13: Replace Type Code with State (cont'd)

Old methods persist because of possible clients

```
class Movie {
  public int getPriceCode( ) {
    return _price.getPriceCode( );
  }

  private Price _price;
  …
```

Delegate call to new classes

```
public void setPriceCode( int arg ) {
  switch ( arg ) {
    case REGULAR:
      _price = new RegularPrice( );
      break;
    case CHILDREN:
      _price = new ChildrenPrice( );
      break;
    case NEW_RELEASE:
      _price = new NewReleasePrice( );
      break;
    default:
      throw new IllegalArgumentException(
                        "Incorrect Price Code" );
```

Replace type code field by reference to new object

Method Movie.getCharge

```
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;
    } /* switch */
    return result;
  }
  …
```

switch should work on Price

Step 14: Move Method

```
class Price {
  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;
    } /* switch */
    return result;
  }
  …
```

Step 14: Move Method (cont'd)

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

Step 15: Replace Conditional with Polymorph.

```
class RegularPrice {
  public double getCharge( int daysRented ) {
    double result = 2;
    if ( daysRented > 2 ) result += ( daysRented – 2 ) * 1.5;
    return result;
  }
  …
}
```

Put each case of the switch into an overriding method

```
class NewReleasePrice {
  public double getCharge( int daysRented ) {
    return daysRented * 3;
  }
  …
}
```

```
class ChildrenPrice {
  …
}
```

Method Movie.getFrequentRenterPoints

```
public class Movie {
  public int getFrequentRenterPoints( int daysRented ) {
    if ( getPriceCode( ) == NEW_RELEASE && daysRented > 1 )
      return 2;
    else
      return 1;
  }
  …
}
```

Apply "Move Method"

Also belongs to Price

```
public class Movie {
  public int getFrequentRenterPoints( int daysRented ) {
    return _price. getFrequentRenterPoints( daysRented );
  }
  … }
```

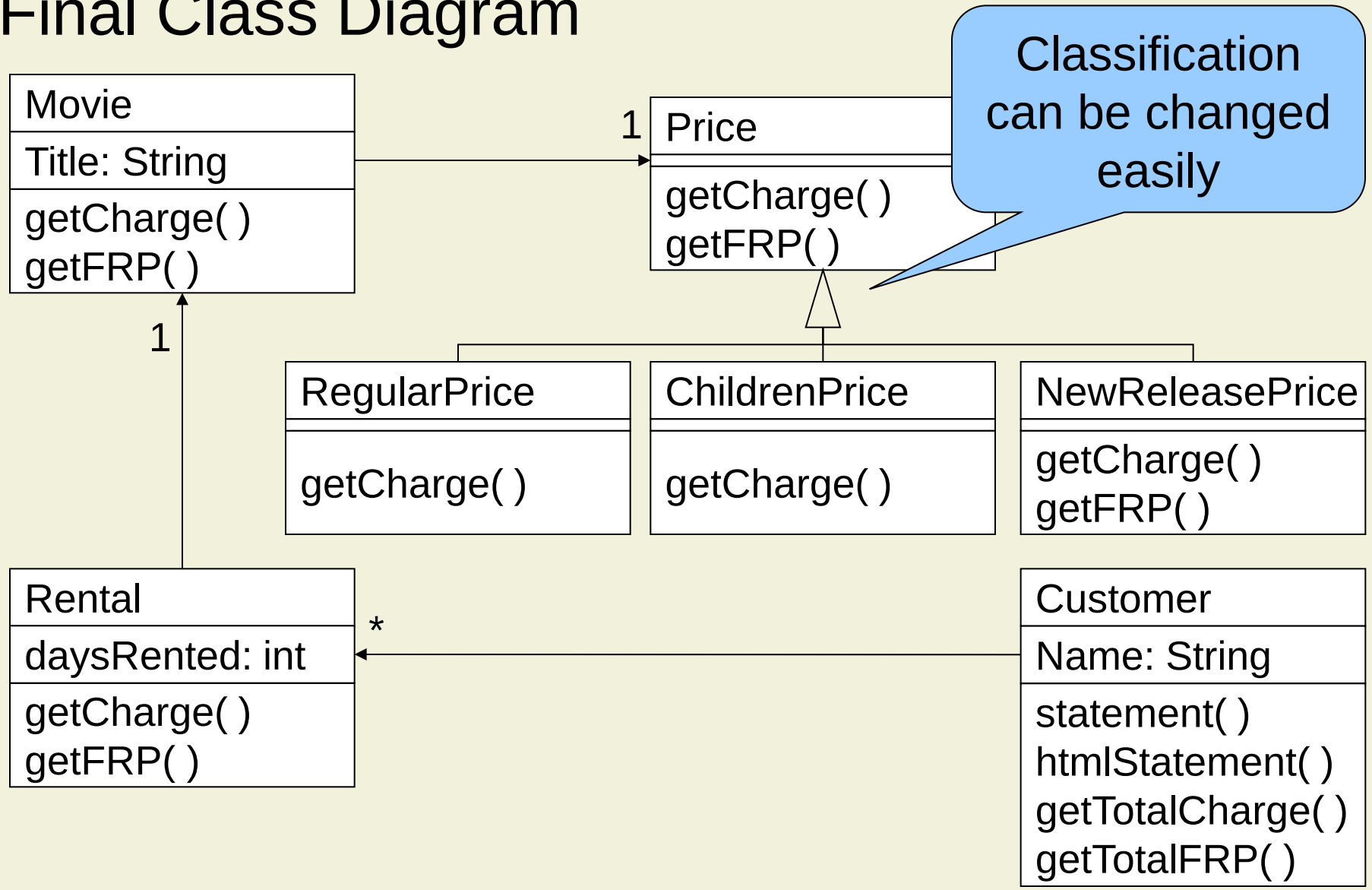
Step 16: Replace Conditional with Polymorph.

```
class Price {
  public int getFrequentRenterPoints( int daysRented ) {
    return 1;
  }
  … }
```

RegularPrice and ChildrenPrice use inherited method

```
class NewReleasePrice {
  public int getFrequentRenterPoints( int daysRented ) {
    return daysRented > 1 ? 2 : 1;
  }
  … }
```

Final Class Diagram

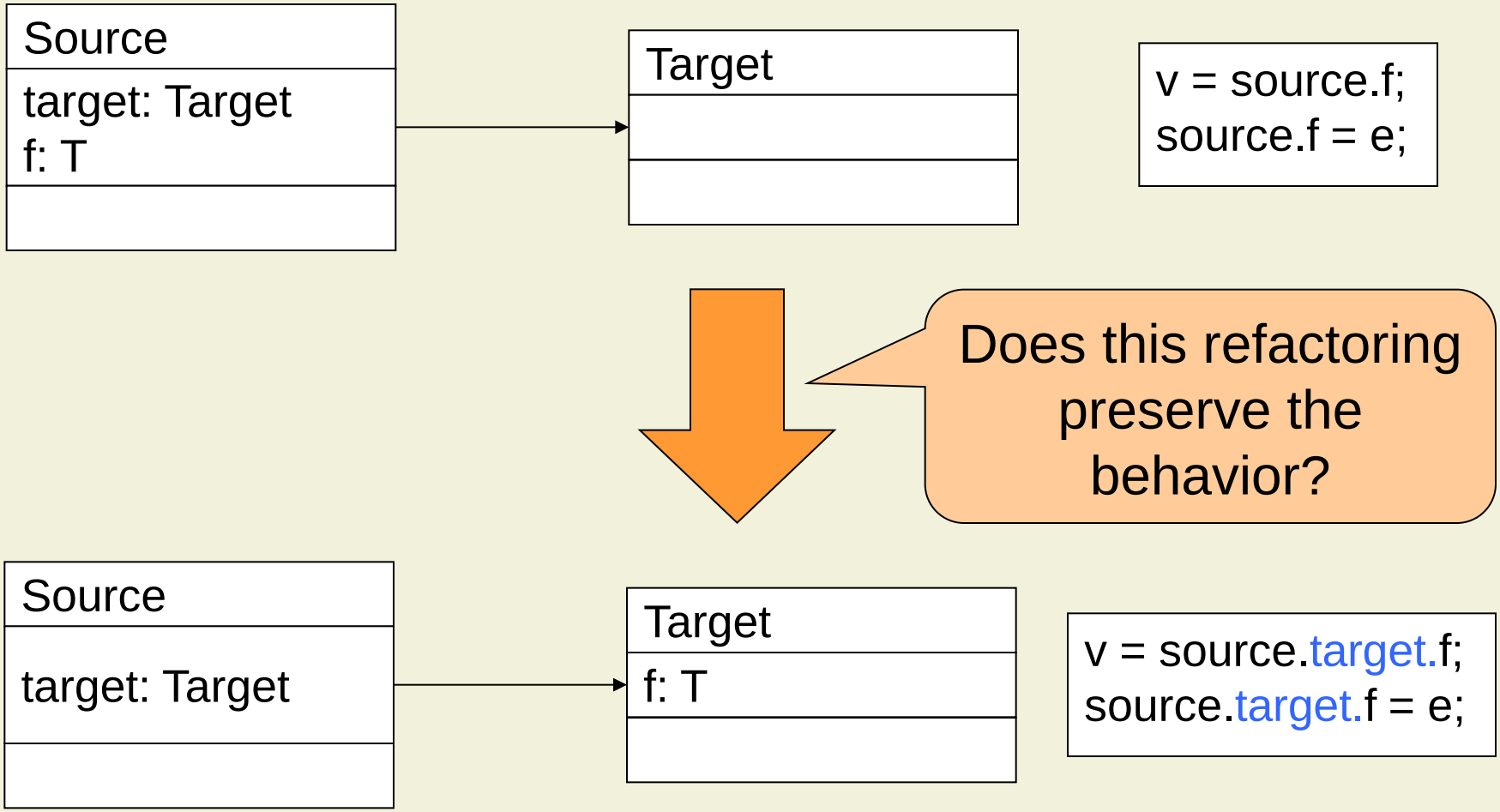


6. Implementation

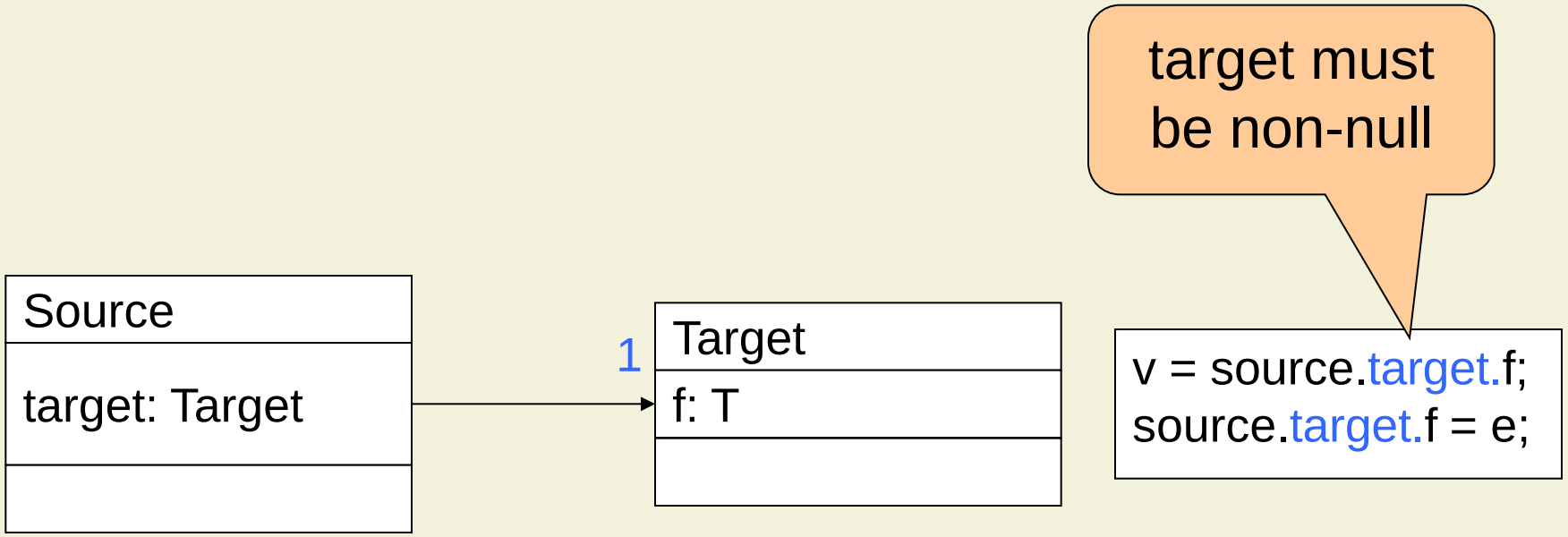
Tool Support

- Refactorings can be applied to models and to code
- IDE provide support for code refactorings
- However, the support is very limited
  - Only simple refactorings
  - Preservation of behavior is not guaranteed
  - Sometimes, not even syntactic correctness is guaranteed

Complex Refactoring: Move Field

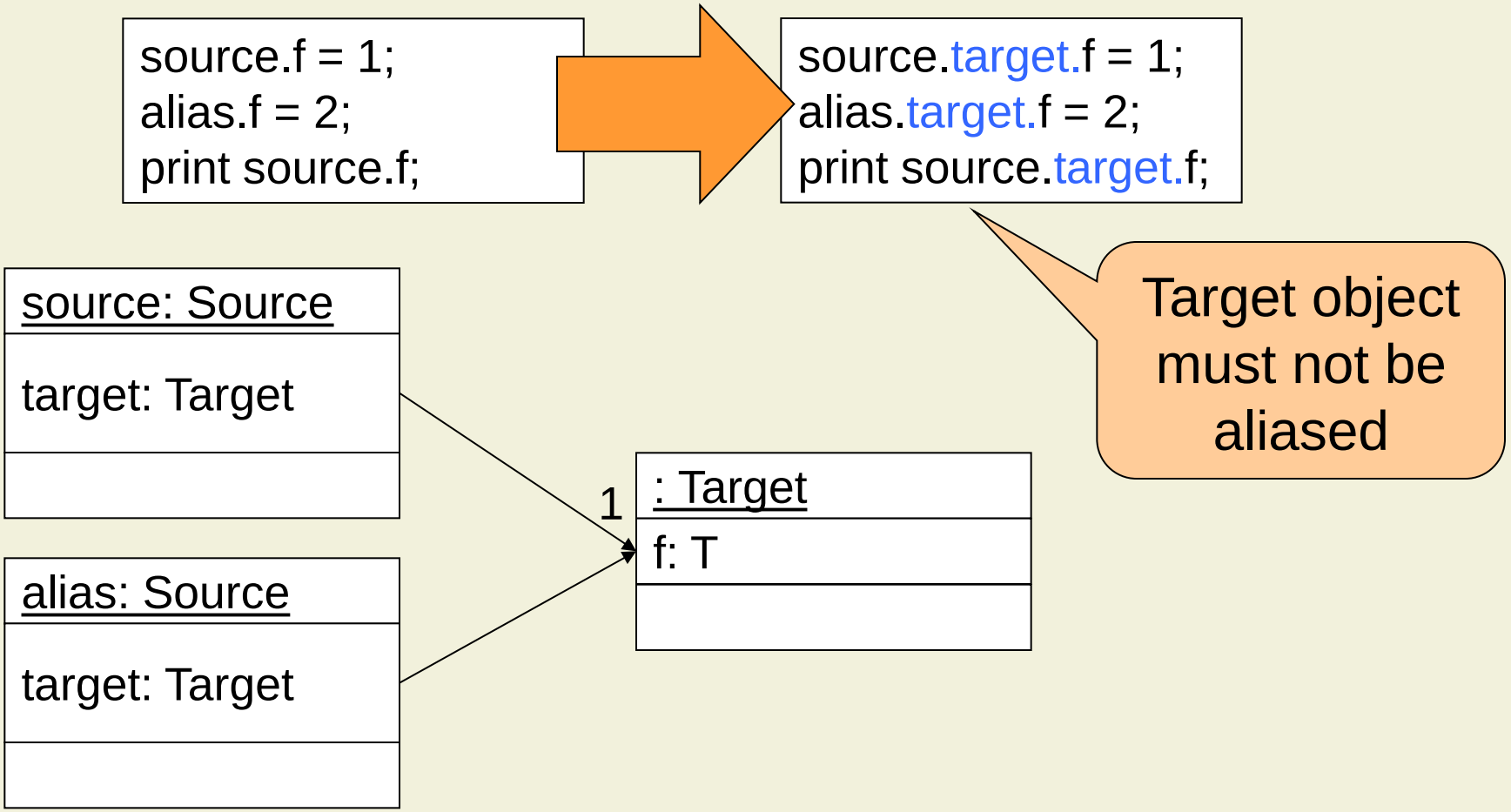


Correctness Conditions for Move Field

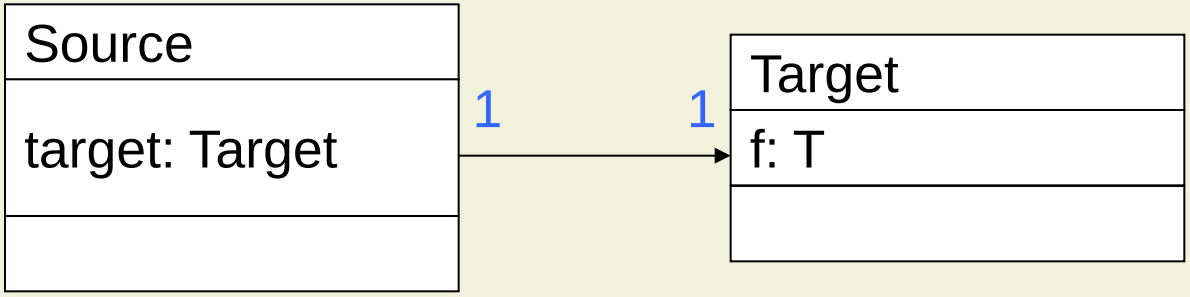


- Treatment of constructors is difficult
  - target must be initialized before f?

Correctness Conditions for Move Field (cont'd)



Correctness Conditions for Move Field (cont'd)



- Move field requires a one-to-one association
  - Difficult to check for code refactorings

Refactoring Principles

- Why do we refactor?
  - To **improve the design** of software
  - To make software **easier to understand**
- When should we refactor?
  - Refactor when you **add functionality**
  - Refactor when you need to **fix a bug**
  - Refactor as you do **code reviews**
  - Refactor when the code **starts to smell**
- What about performance?
  - Worry about performance **only when you have identified a performance problem**