F. Tip and
M. Weintraub

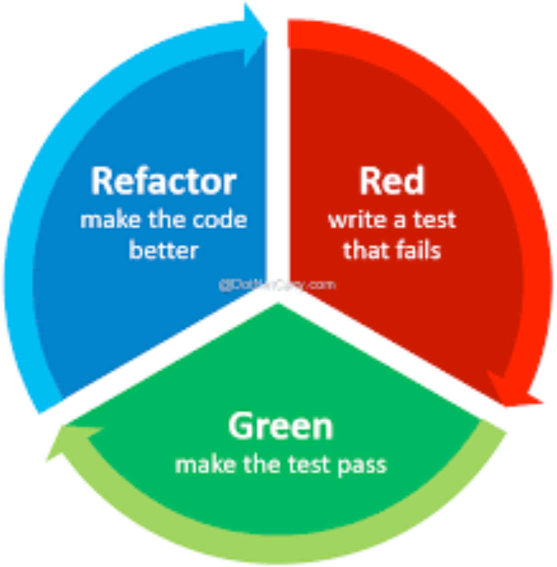
REFACTORING

Thanks go to Andreas Zeller for allowing incorporation of his materials

TODAY'S LECTURE

- **anti-patterns**
  - common response to a recurring problem that is usually ineffective and risks being highly counterproductive

- **refactoring**
  - improving the design of existing code



ANTI-PATTERNS

If the following patterns occur in your software project, you're doing it wrong!

THE BLOB

INDESCRIBABLE...
INDESTRUCTIBLE!
NOTHING CAN STOP IT!
THE BLOB
'X'
STEVEN McQUEEN
ANETA CORSEAUT · EARL ROWE
PRODUCED BY JACK H. HARRIS · DIRECTED BY IRVIN S. YEAWORTH, JR. · SCREENPLAY BY THEODORE SIMONSON AND KATE PHILLIPS
FROM AN IDEA BY IRVINE H. MILLGATE
A TONYLYN PRODUCTION · COLOUR BY DELUXE

- **The Blob**. (aka “God Class”) One object("blob") has the majority of the responsibilities, while most of the others just store data or provide only primitive services.

- *Solution*: refactoring

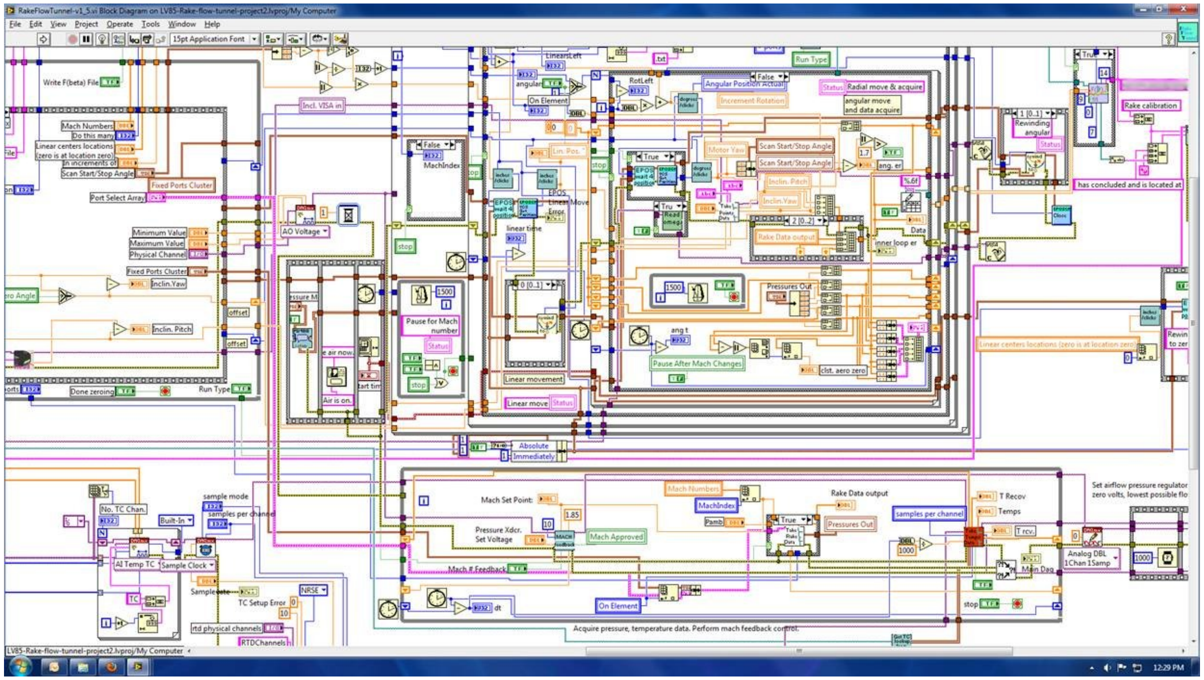
THE GOLDEN HAMMER

- **The Golden Hammer.** A favorite solution ("Golden Hammer") is applied to every single problem: With a hammer, every problem looks like a nail.
- *Solution*: improve level of education

COPY AND PASTE PROGRAMMING

- **Copy-and-Paste Programming.** Code is reused in multiple places by being copied and adjusted, causing maintenance problems.
- *Solution*: Identification of common features; refactoring

SPAGHETTI CODE



- **Spaghetti Code**. The code is mostly unstructured; it's neither particularly modular nor object-oriented; control flow is obscure.

- *Solution*: Prevent by designing first, and only then implementing. Existing spaghetti code should be refactored.

MUSHROOM MANAGEMENT

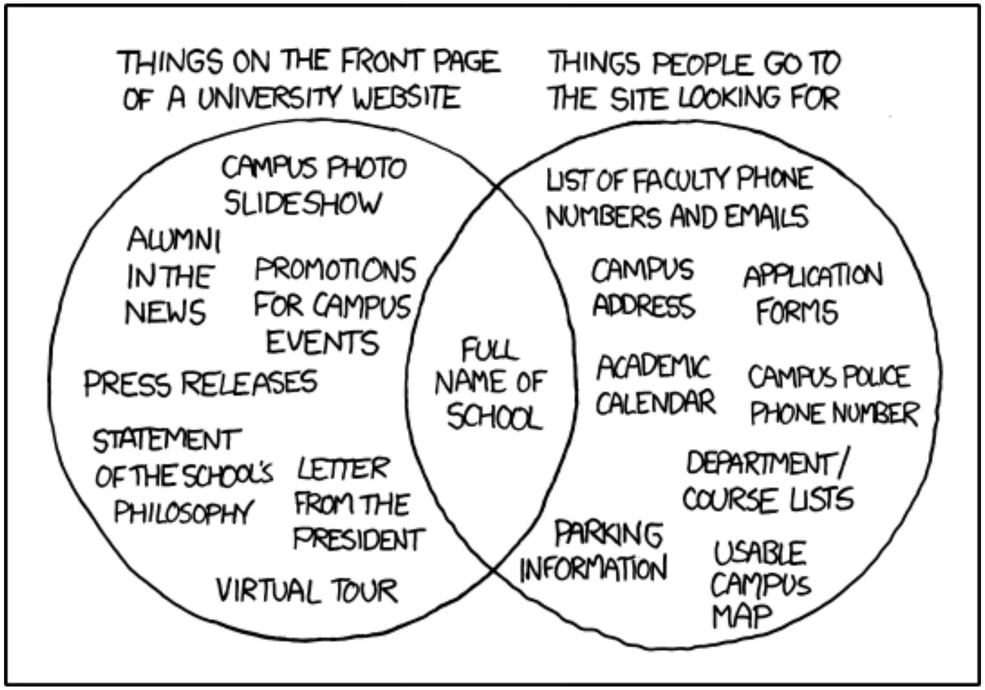
- **Mushroom Management**. Developers are kept away from users.
- *Solution*: Improve communication.

VENDOR LOCK-IN

- **Vendor Lock-In**. A system is dependent on a proprietary architecture or data format.

- *Solution*: Improve portability, introduce abstractions.

DESIGN BY COMMITTEE


THINGS ON THE FRONT PAGE OF A UNIVERSITY WEBSITE
THINGS PEOPLE GO TO THE SITE LOOKING FOR
CAMPUS PHOTO SLIDESHOW
ALUMNI IN THE NEWS
PROMOTIONS FOR CAMPUS EVENTS
PRESS RELEASES
STATEMENT OF THE SCHOOL'S PHILOSOPHY
LETTER FROM THE PRESIDENT
VIRTUAL TOUR
FULL NAME OF SCHOOL
LIST OF FACULTY PHONE NUMBERS AND EMAILS
CAMPUS ADDRESS
APPLICATION FORMS
ACADEMIC CALENDAR
CAMPUS POLICE PHONE NUMBER
DEPARTMENT/ COURSE LISTS
PARKING INFORMATION
USABLE CAMPUS MAP

- **Design by Committee**. The typical anti-pattern of standardizing committees, that tend to satisfy every single participant, and create overly complex and ambivalent designs ("A camel is a horse designed by a committee").
  - Known examples: SQL and COBRA.
- *Solution*: Improve group dynamics and meetings (teamwork)

REINVENT THE WHEEL

- **Reinvent the Wheel.** Due to lack of knowledge about existing products and solutions, the wheel gets reinvented over and over, which leads to increased development costs and problems with deadlines.

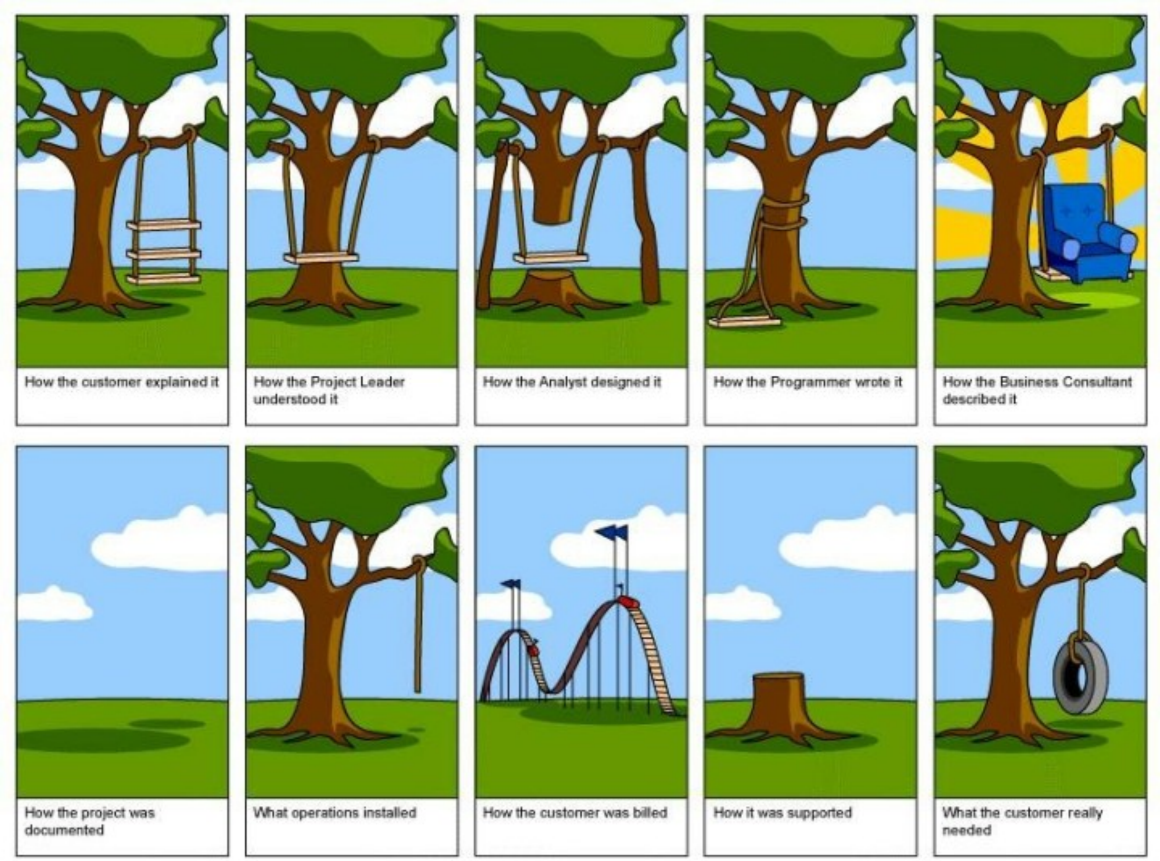
- *Solution*: Improve knowledge management.

INTELLECTUAL VIOLENCE

- **Intellectual Violence.** Someone who has mastered a new theory, technique or buzzwords, uses his knowledge to intimidate others.

- *Solution*: Ask for clarification!

PROJECT MISMANAGEMENT

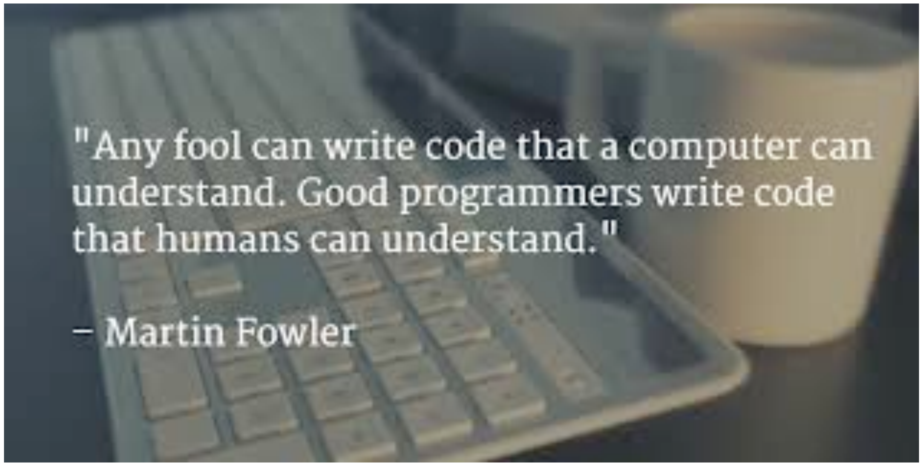

How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

- **Project Mismanagement**. The manager of the project is unable to make decisions.
- *Solution*: Admit having the problem; set clear short-term goals.

OTHER ANTI-PATTERNS

- **Boat anchor**: Retaining a part of a system that has become obsolete
- **Premature optimization**: Focusing on performance of the code too early
- **Lava flow**: code written under sub-optimal conditions is put into production and added to while still in a developmental state.
- **Dependency hell**: Problems with versions of required products
- …

REFACTORING



"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

– Martin Fowler

- **refactoring** is the process of applying transformations (refactorings) to a program, with the goal of improving its design

- goals:
  - keep program readable, understandable, and maintainable
  - by eliminating small problems soon, you can avoid big trouble later

- characteristics:
  - behavior-preserving: make sure the program works after each step
  - typically small steps

WHY REFACTOR?

- why does refactoring become necessary?
  - requirements have changed, and a different design is needed
  - design needs to be more flexible (so new features can be added)
  - address sloppiness by programmers (e.g., cut-and-paste programming)

- refactoring often has the effect of making a design more flexible
  - design patterns are often a target for refactoring

HISTORY

- refactoring is something good programmers have always done
  - popularized by various agile development methodologies

- especially popular in the context of object-oriented languages
  - perhaps because object-oriented features are well-suited to make designs flexible and reusable
  - but refactoring is not specific to OO

- Opdyke's PhD thesis (1990) presents refactoring tools for Smalltalk
  - since then various other students of Ralph Johnson have worked on refactoring tools, mostly for Smalltalk

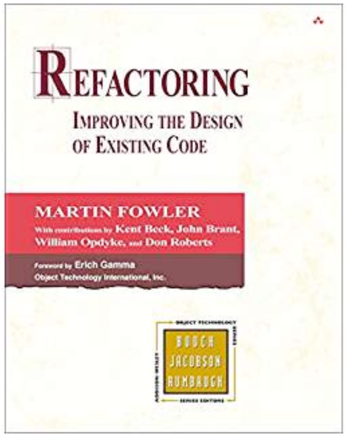
PRESERVING PROGRAM BEHAVIOR

- How to ensure that the program does the same thing before and after applying a refactoring?

- **testing**: write tests that exercise the parts of the program affected by the refactoring
  - in general, no guarantees

- **program analysis**: perform a static analysis of the program using techniques similar to those used in compilers
  - difficult to implement; analysis may be imprecise and say that a refactoring cannot be applied safely
  - modern IDEs provide refactoring support (e.g., Eclipse, IDEA)

FOWLER’S BOOK



REFACTORING

IMPROVING THE DESIGN OF EXISTING CODE

MARTIN FOWLER

With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International, Inc.

OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

- presents a **catalogue of refactorings**, similar to the catalogue of design patterns in the GoF book
- catalogues “**bad smells**” - indications that refactoring may be needed
- explains when to apply refactorings
  - UML diagrams to illustrate the situation before and after
- **examples** of code before and after each refactoring
  - small examples that are representative of larger systems
- many of Fowler’s refactorings are the **inverse** of another refactoring
  - often there is not a unique “best” solution
  - discussion of the tradeoffs

REFACTORING: AN EXTENDED EXAMPLE

```
public class Example {

  void printOwing(){
    Iterator<Order> it = _orders.iterator();
    double outstanding = 0.0;
    // print banner
    System.out.println("*************************");
    System.out.println("***** Customer Owes *****");
    System.out.println("*************************");
    // calculate outstanding
    while (it.hasNext()){
      Order each = it.next();
      outstanding += each.getAmount();
    }
    // print details
    System.out.println("name:" + _name);
    System.out.println("amount" + outstanding);
  }

  private String _name;
  private double _outstanding;
  private Set<Order> _orders;
}
```

1. “EXTRACT METHOD”

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  // print banner
  System.out.println("*************************");
  System.out.println("***** Customer Owes *****");
  System.out.println("*************************");

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  // print details
  System.out.println("name:" + _name);
  System.out.println("amount" + outstanding);
}
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  // print details
  System.out.println("name:" + _name);
  System.out.println("amount" + outstanding);
}
private void printBanner() {
  // print banner
  System.out.println("*************************");
  System.out.println("***** Customer Owes *****");
  System.out.println("*************************");
}
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  // print details
  System.out.println("name:" + _name);
  System.out.println("amount" + outstanding);
}
private void printBanner() {
  // print banner
  System.out.println("*************************");
  System.out.println("***** Customer Owes *****");
  System.out.println("*************************");
}
```

2. “EXTRACT METHOD”

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  // print details
  System.out.println("name:" + _name);
  System.out.println("amount" + outstanding);
}
private void printBanner() {
  // print banner
  System.out.println("*************************");
  System.out.println("***** Customer Owes *****");
  System.out.println("*************************");
}
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);
}
private void printDetails(double outstanding) {
  // print details
  System.out.println("name:" + _name);
  System.out.println("amount" + outstanding);
}
private void printBanner() {
  // print banner
  System.out.println("*************************");
  System.out.println("***** Customer Owes *****");
  System.out.println("*************************");
}
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);
}
private void printDetails(double outstanding) {
  // print details
  System.out.println("name:" + _name);
  System.out.println("amount" + outstanding);
}
private void printBanner() {
  // print banner
  System.out.println("*************************");
  System.out.println("***** Customer Owes *****");
  System.out.println("*************************");
}
```

3. REMOVE COMMENTS

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);
}
private void printDetails(double outstanding) {
  ~~// print details~~
  System.out.println("name:" + _name);
  System.out.println("amount" + outstanding);
}
private void printBanner() {
  ~~// print banner~~
  System.out.println("*************************");
  System.out.println("***** Customer Owes *****");
  System.out.println("*************************");
}
```

```java
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

4. “EXTRACT METHOD”

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }

  printDetails(outstanding);
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double outstanding) {
  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double outstanding) {
  // calculate outstanding
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

5. REMOVE COMMENT

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double outstanding) {
  ~~// calculate outstanding~~
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double outstanding) {
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

6. “RENAME VARIABLE”

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double outstanding) {
  while (it.hasNext()){
    Order each = it.next();
    outstanding += each.getAmount();
  }
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```java
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

7. MOVE STATEMENT

```
void printOwing(){
  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;

  printBanner();

  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;
  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

8. “INLINE LOCAL VARIABLE”

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double outstanding = 0.0;
  outstanding = getOutstanding(it, outstanding);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double outstanding = getOutstanding(it, 0.0);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```java
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double outstanding = getOutstanding(it, 0.0);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

GOAL: REFACTOR THIS INTO A SINGLE METHOD

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double outstanding = getOutstanding(it, 0.0);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

9. “INLINE METHOD”

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double outstanding = getOutstanding(it, 0.0);

  printDetails(outstanding);
}

private double getOutstanding(Iterator<Order> it, double result) {
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double result = 0.0;
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  double outstanding = result;

  printDetails(outstanding);
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double result = 0.0;
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  double outstanding = result;

  printDetails(outstanding);
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

10. “EXTRACT METHOD”

```
void printOwing(){
  printBanner();

  Iterator<Order> it = _orders.iterator();
  double result = 0.0;
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  double outstanding = result;

  printDetails(outstanding);
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  printBanner();
  double outstanding = getOutstanding();
  printDetails(outstanding);
}

private double getOutstanding() {
  Iterator<Order> it = _orders.iterator();
  double result = 0.0;
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  double outstanding = result;
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  printBanner();
  double outstanding = getOutstanding();
  printDetails(outstanding);
}

private double getOutstanding() {
  Iterator<Order> it = _orders.iterator();
  double result = 0.0;
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  double outstanding = result;
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

11. “INLINE LOCAL VARIABLE”

```
void printOwing(){
  printBanner();
  double outstanding = getOutstanding();
  printDetails(outstanding);
}

private double getOutstanding() {
  Iterator<Order> it = _orders.iterator();
  double result = 0.0;
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  double outstanding = result;
  return outstanding;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```java
void printOwing(){
  printBanner();
  double outstanding = getOutstanding();
  printDetails(outstanding);
}

private double getOutstanding() {
  Iterator<Order> it = _orders.iterator();
  double result = 0.0;
  while (it.hasNext()){
    Order each = it.next();
    result += each.getAmount();
  }
  return result;
}

private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

```
void printOwing(){
  printBanner();
  double outstanding = getOutstanding();
  printDetails(outstanding);
}

private double getOutstanding() { … }
private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

12. “INLINE LOCAL VARIABLE”

```
void printOwing(){
  printBanner();
  double outstanding = getOutstanding();
  printDetails(outstanding);
}

private double getOutstanding() { … }
private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

DONE!

```
void printOwing(){
  printBanner();
  printDetails(getOutstanding());
}

private double getOutstanding() { … }
private void printDetails(double outstanding) { … }
private void printBanner() { … }
```

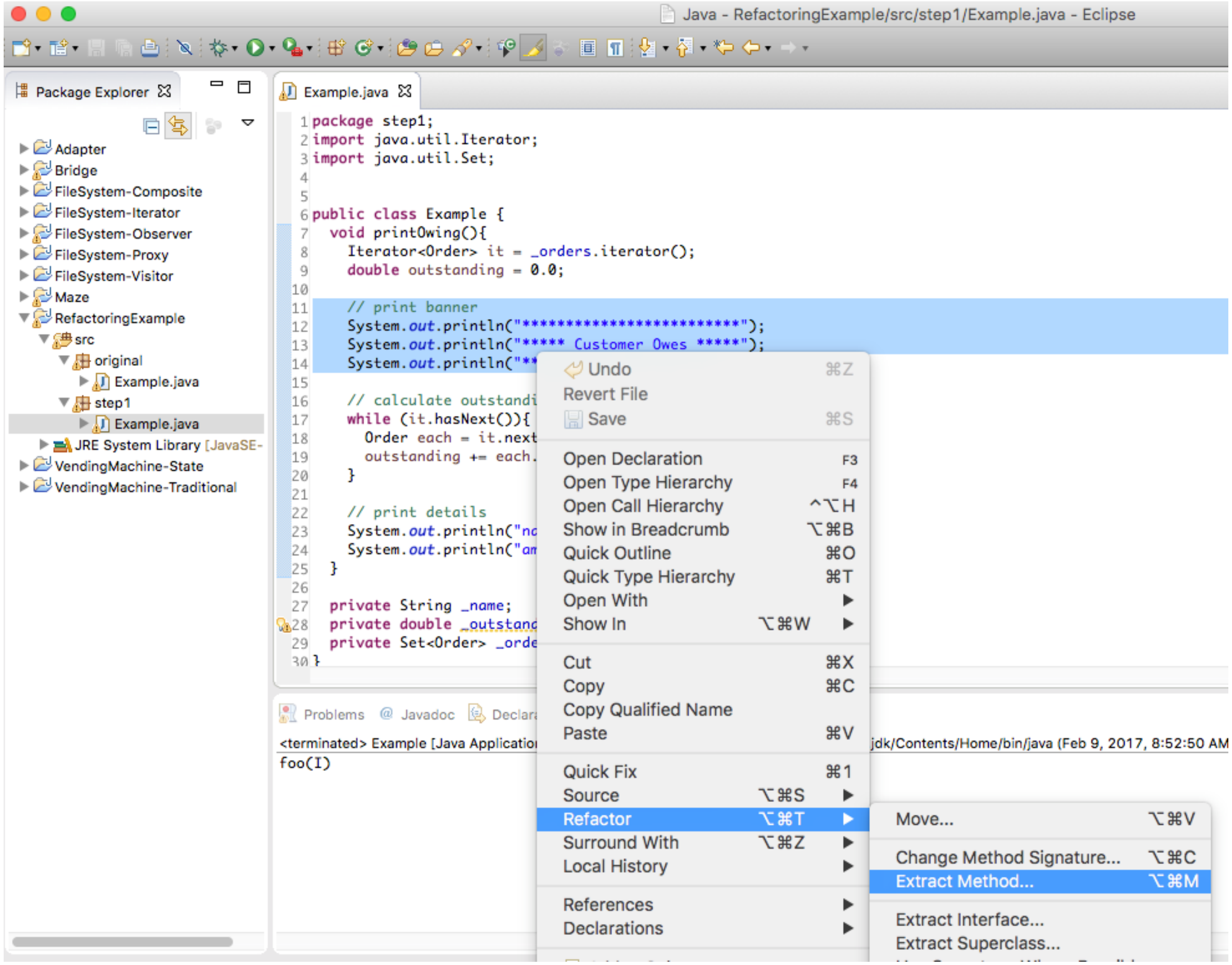
OBSERVATIONS

- **small incremental steps** that preserve program behavior

- most steps are so simple that they can be **automated**
  - *exercise*: try to refactor the example using Eclipse (or IntelliJ IDEA)
  - automation limited in complex cases

- refactoring does not always proceed “in a straight line”
  - sometimes, undo a step you did earlier…
  - …when you have insights for a better design

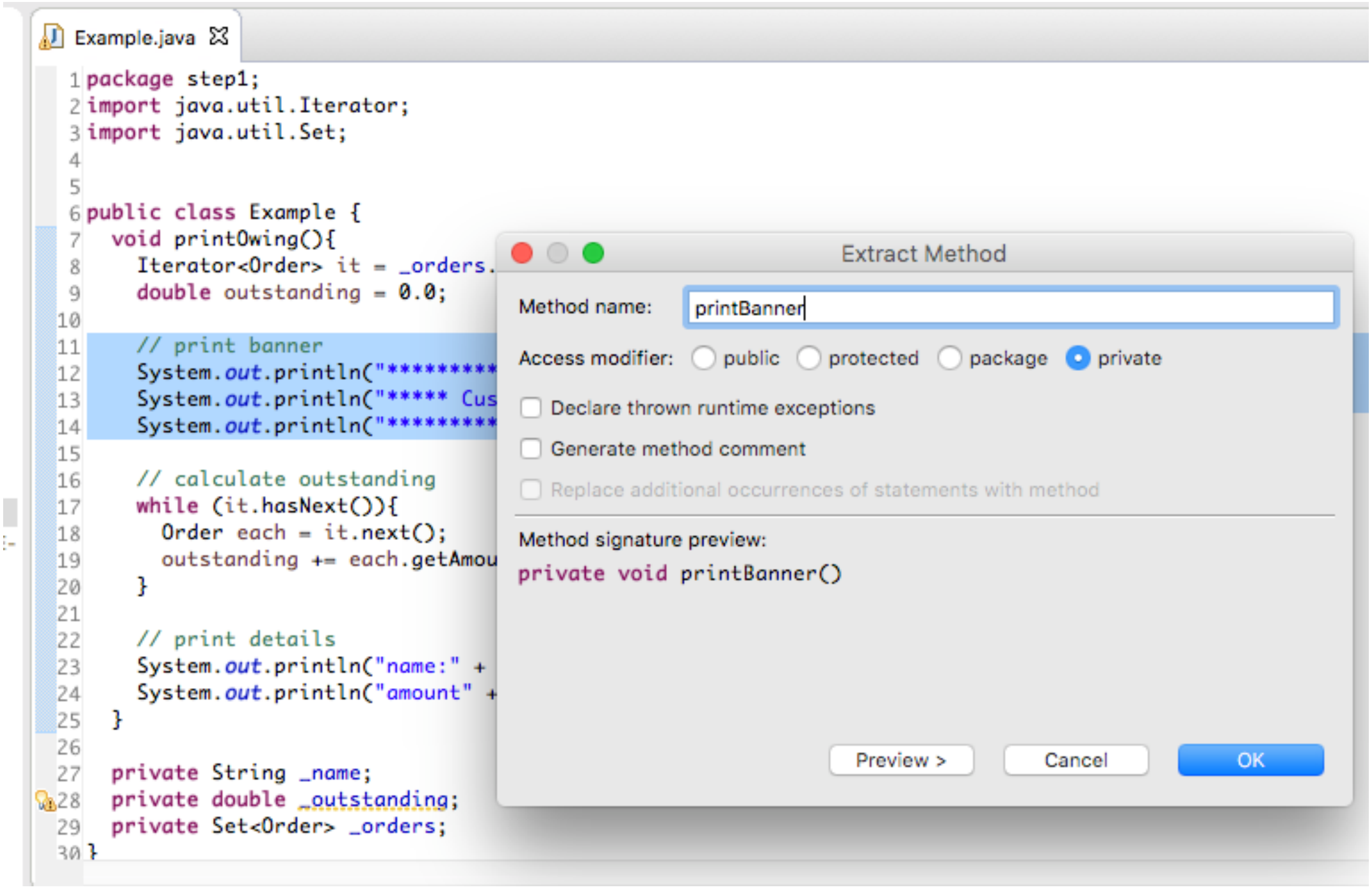
REFACTORING IN ECLIPSE



```
package step1;
import java.util.Iterator;
import java.util.Set;


public class Example {
  void printOwing(){
    Iterator<Order> it = _orders.iterator();
    double outstanding = 0.0;

    // print banner
    System.out.println("*************************");
    System.out.println("***** Customer Owes *****");
    System.out.println("**

    // calculate outstandi
    while (it.hasNext()){
      Order each = it.next
      outstanding += each.
    }

    // print details
    System.out.println("na
    System.out.println("am
  }

  private String _name;
  private double _outstand
  private Set<Order> _orde
}
```

```
<terminated> Example [Java Applicatio   jdk/Contents/Home/bin/java (Feb 9, 2017, 8:52:50 AM
foo(I)
```

REFACTORING IN ECLIPSE



```
package step1;
import java.util.Iterator;
import java.util.Set;


public class Example {
  void printOwing(){
    Iterator<Order> it = _orders.
    double outstanding = 0.0;

    // print banner
    System.out.println("*********
    System.out.println("***** Cus
    System.out.println("*********

    // calculate outstanding
    while (it.hasNext()){
      Order each = it.next();
      outstanding += each.getAmou
    }

    // print details
    System.out.println("name:" +
    System.out.println("amount" +
  }

  private String _name;
  private double _outstanding;
  private Set<Order> _orders;
}
```

Extract Method

Method name: printBanner

Access modifier: ○ public ○ protected ○ package ● private

□ Declare thrown runtime exceptions

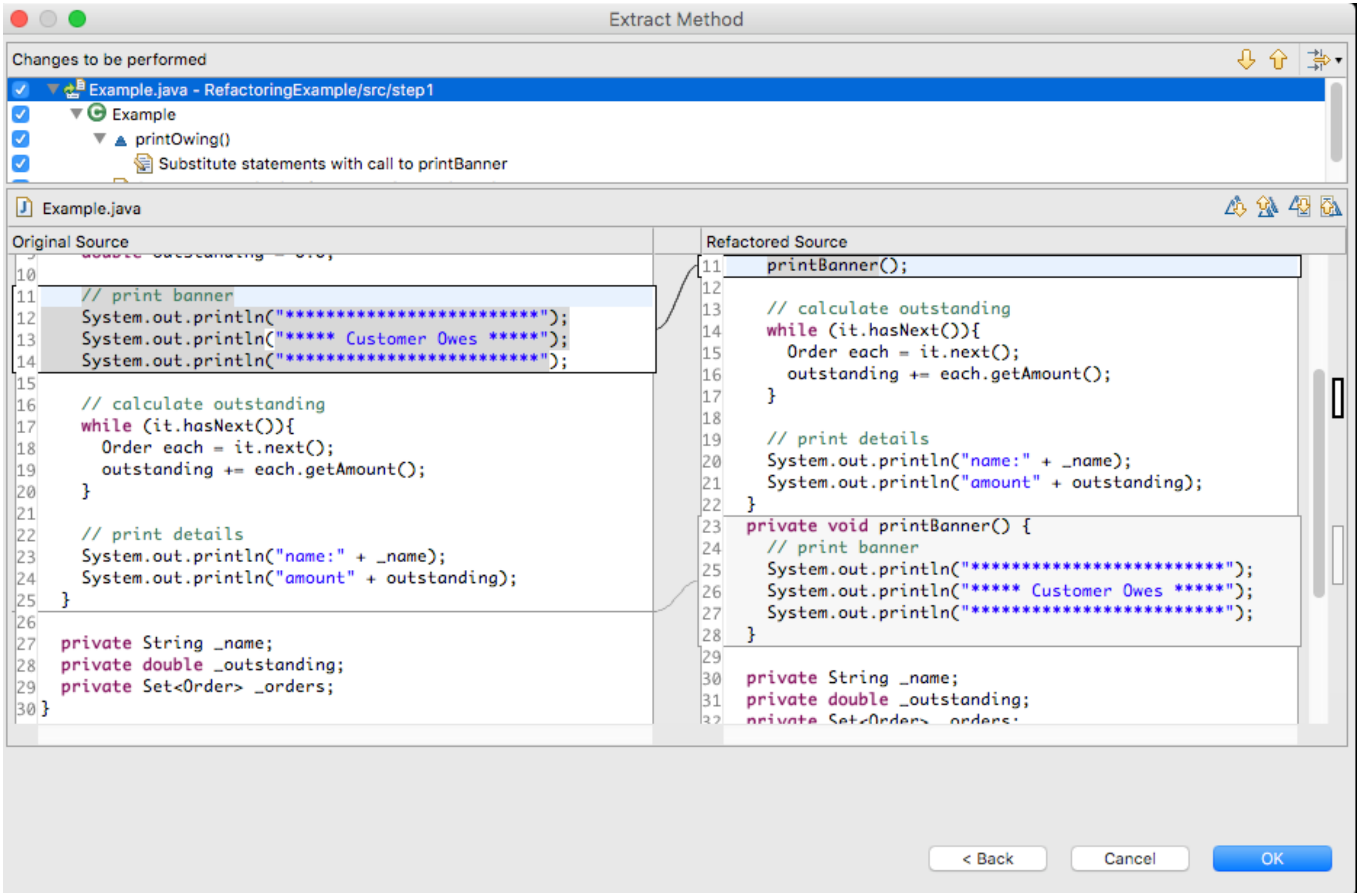
□ Generate method comment

□ Replace additional occurrences of statements with method

Method signature preview:

```
private void printBanner()
```

REFACTORING IN ECLIPSE



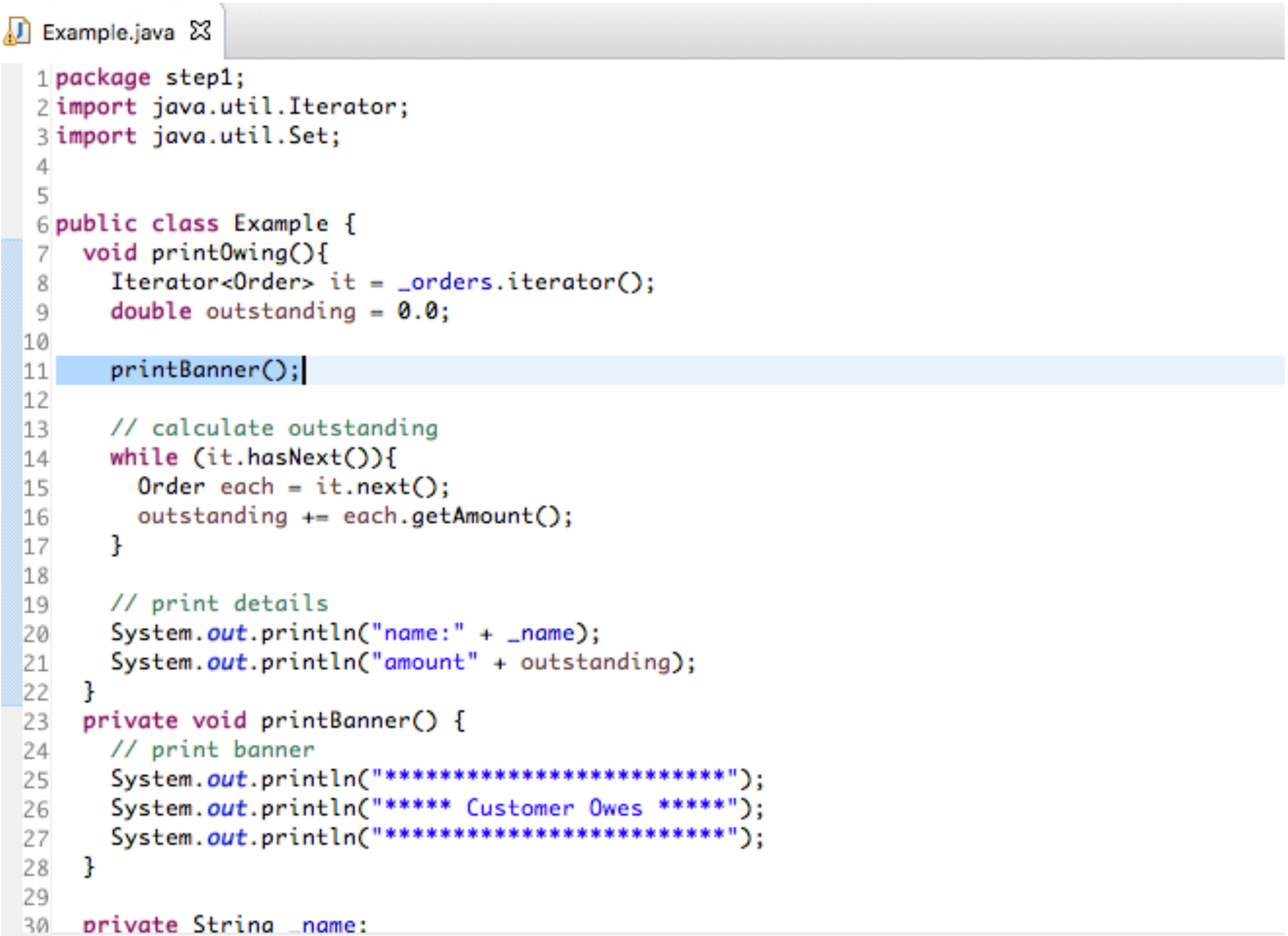
Original Source

```
    double outstanding = 0.0;

    // print banner
    System.out.println("*************************");
    System.out.println("***** Customer Owes *****");
    System.out.println("*************************");

    // calculate outstanding
    while (it.hasNext()){
      Order each = it.next();
      outstanding += each.getAmount();
    }

    // print details
    System.out.println("name:" + _name);
    System.out.println("amount" + outstanding);
  }

  private String _name;
  private double _outstanding;
  private Set<Order> _orders;
}
```

Refactored Source

```
    printBanner();

    // calculate outstanding
    while (it.hasNext()){
      Order each = it.next();
      outstanding += each.getAmount();
    }

    // print details
    System.out.println("name:" + _name);
    System.out.println("amount" + outstanding);
  }
  private void printBanner() {
    // print banner
    System.out.println("*************************");
    System.out.println("***** Customer Owes *****");
    System.out.println("*************************");
  }

  private String _name;
  private double _outstanding;
  private Set<Order> _orders;
```

REFACTORING IN ECLIPSE



```java
package step1;
import java.util.Iterator;
import java.util.Set;


public class Example {
  void printOwing(){
    Iterator<Order> it = _orders.iterator();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (it.hasNext()){
      Order each = it.next();
      outstanding += each.getAmount();
    }

    // print details
    System.out.println("name:" + _name);
    System.out.println("amount" + outstanding);
  }
  private void printBanner() {
    // print banner
    System.out.println("************************");
    System.out.println("***** Customer Owes *****");
    System.out.println("************************");
  }

  private String _name;
```

“LOCAL” REFACTORINGS

Rename	rename variables, fields methods, classes, packages provide better intuition for the renamed element’s purpose
Extract Method	extract statements into a new method enables reuse; avoid cut-and-paste programming improve readability
Inline Method	replace a method call with the method’s body often useful as intermediate step
Extract Local	introduce a new local variable for a designated expression
Inline Local	replace a local variable with the expression that defines its value
Change Method Signature	reorder a method’s parameters
Encapsulate Field	introduce getter/setter methods
Convert Local Variable to Field	convert local variable to field sometimes useful to enable application of Extract Method

RENAME: KEY ISSUES

- **renaming methods**
  - when renaming a method **m**, ensure that all methods overriding **m** and all methods overridden by **m** are renamed as well
- **renaming fields**
  - when renaming a field **f**, need to ensure that hiding relationships are preserved
- **renaming variables**
  - need to ensure that hiding relationships are preserved

EXTRACT METHOD: KEY ISSUES

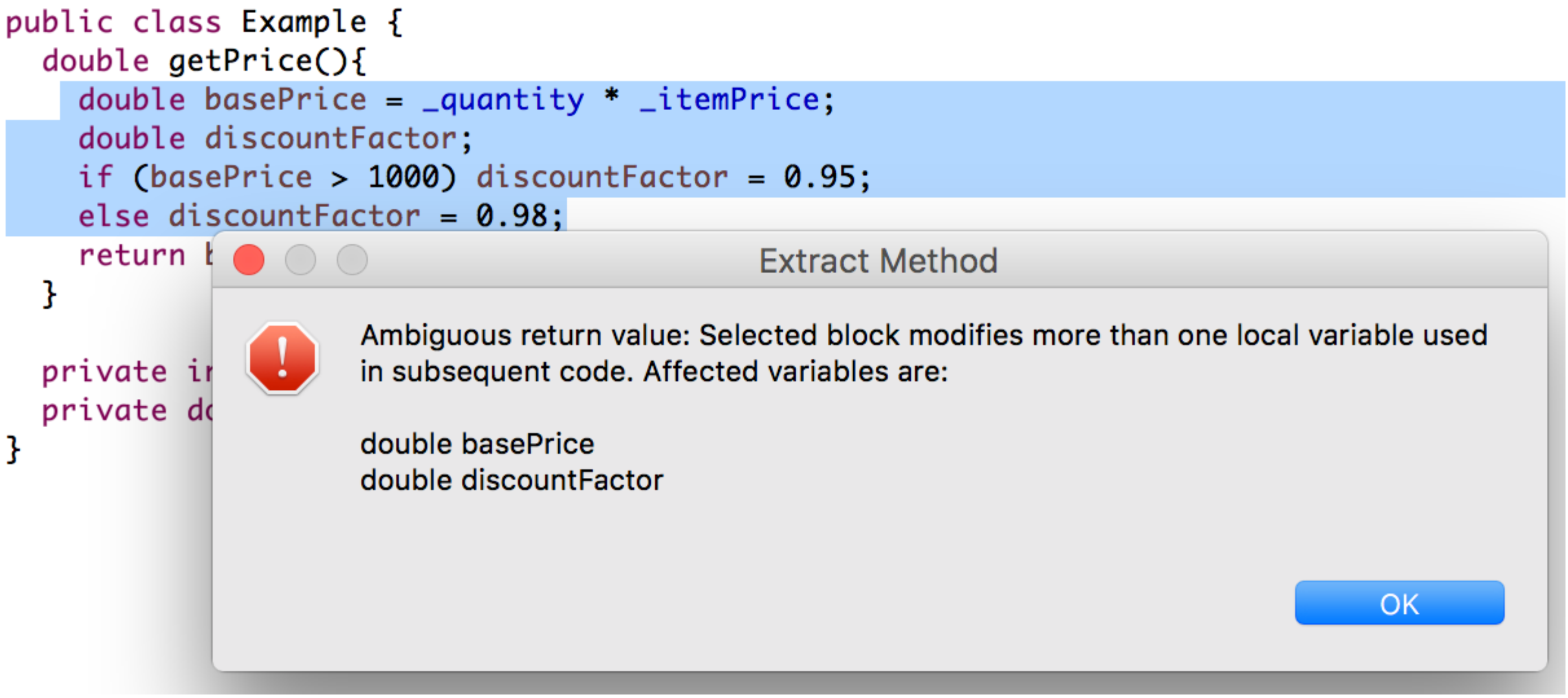
- **analyze usage of local variables**
  - only used in target method? Declare in target method.
  - used in source method & read but not assigned in target method? Pass in as parameters.
  - used in source method & assigned in target method & new value used in source method? Pass in as parameters and return changed value.

- if more than one variable is modified:
  - select different code to extract
  - apply additional refactorings (e.g., **Replace Temp with Query**, or **Replace Method with Method Object**)

INLINE METHOD: KEY ISSUES

- **polymorphism**
  - if the invoked method is overridden, cannot be sure which method will be invoked
- **multiple return values**
  - need to be eliminated before applying the refactoring
  - e.g., by introducing an additional variable in the invoked method
- **recursion**
  - prevents this refactoring from being applied

REPLACE TEMP WITH QUERY



```
public class Example {
  double getPrice(){
    double basePrice = _quantity * _itemPrice;
    double discountFactor;
    if (basePrice > 1000) discountFactor = 0.95;
    else discountFactor = 0.98;
    return b
  }

  private i
  private d
}
```

Ambiguous return value: Selected block modifies more than one local variable used in subsequent code. Affected variables are:

double basePrice
double discountFactor

- sometimes, **Extract Method** cannot be applied because too many local variables are modified
- solution: **Replace Local Variable (Temp) with Query**

REPLACE TEMP WITH QUERY

```
double getPrice(){
  double basePrice = _quantity * _itemPrice;
  double discountFactor;
  if (basePrice > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return basePrice * discountFactor;
}
```

```
double getPrice(){
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return basePrice() * discountFactor;
}
```

```
private double basePrice() {
  return _quantity * _itemPrice;
}
```

```
double getPrice(){
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return basePrice() * discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

EXTRACT METHOD

```
double getPrice(){
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return basePrice() * discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

```java
double getPrice(){
  double discountFactor = discountFactor();
  return basePrice() * discountFactor;
}

private double discountFactor() {
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

```
double getPrice(){
  double discountFactor = discountFactor();
  return basePrice() * discountFactor;
}

private double discountFactor() {
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

INLINE LOCAL VARIABLE

```
double getPrice(){
  double discountFactor = discountFactor();
  return basePrice() * discountFactor;
}

private double discountFactor() {
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

```
double getPrice(){
  return basePrice() * discountFactor();
}

private double discountFactor() {
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

```
double getPrice(){
  return basePrice() * discountFactor();
}

private double discountFactor() {
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

REPLACE IF WITH CONDITIONAL EXPRESSION

```
double getPrice(){
  return basePrice() * discountFactor();
}

private double discountFactor() {
  double discountFactor;
  if (basePrice() > 1000) discountFactor = 0.95;
  else discountFactor = 0.98;
  return discountFactor;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

DONE!

```
double getPrice(){
  return basePrice() * discountFactor();
}

private double discountFactor() {
  return (basePrice() > 1000) ? 0.95 : 0.98;
}

private double basePrice() {
  return _quantity * _itemPrice;
}
```

MOVE METHOD

- A method is using or is used by more features of another class than the class on which it is defined
  - “feature envy”

- Create a new method with a similar body in the class it uses most.

- either turn the old method into a simple delegation, or remove it altogether.

EXAMPLE: MOVE METHOD

```java
class Account {
  double overdraftCharge(){
    if (_type.isPremium()){
      double result = 10;
      if (_daysOverdrawn > 7)
        result += (_daysOverdrawn - 7)*0.85;
      return result;
    } else return _daysOverdrawn*1.75;
  }
  double bankCharge(){
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
}
```

```java
class AccountType {
  …
  public boolean isPremium() { … }
   …
}
```

MOVE METHOD

```
class Account {
  double overdraftCharge(){
    if (_type.isPremium()){
      double result = 10;
      if (_daysOverdrawn > 7)
        result += (_daysOverdrawn - 7)*0.85;
      return result;
    } else return _daysOverdrawn*1.75;
  }
  double bankCharge(){
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
}
```

```
class AccountType {
  …
  public boolean isPremium() { … }
   …

}
```

MOVE METHOD

```
class Account {
  double overdraftCharge(){
    return _type.overdraftCharge(_daysOverdrawn);
  }
  double bankCharge(){
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
}
```

```
class AccountType {

  public boolean isPremium() { return false; }

  double overdraftCharge(int daysOverdrawn){
    if (isPremium()){
      double result = 10;
      if (daysOverdrawn > 7)
        result += (daysOverdrawn - 7)*0.85;
      return result;
    } else return daysOverdrawn*1.75;
  }
  …
}
```

MOVE METHOD

```
class Account {
  double overdraftCharge(){
    return _type.overdraftCharge(_daysOverdrawn);
  }
  double bankCharge(){
    double result = 4.5;
    if (_daysOverdrawn > 0)
      result += overdraftCharge();
    return result;
  }
  private AccountType _type;
  private int _daysOverdrawn;
}
```

```
class AccountType {

  public boolean isPremium() { return false; }

  double overdraftCharge(int daysOverdrawn){
    if (isPremium()){
      double result = 10;
      if (daysOverdrawn > 7)
        result += (daysOverdrawn - 7)*0.85;
      return result;
    } else return daysOverdrawn*1.75;
  }
  …
}
```

- if the method is not needed in its old location, you can remove the forwarding method altogether
- supported in Eclipse, but instead of passing field as the parameter, it passes the entire Account object.

MOVE METHOD: RELATED REFACTORINGS

- **Move Field**
  - move a field from source class to target class
  - similar issues

- **Extract Class**
  - break up a single class into two classes
  - create a new class that will contain some of the functionality of the source class
  - create link between source and target class (e.g., in constructor of source class)
  - move functionality to target class with repeated applications of Move Method and Move Field

TYPE-RELATED REFACTORINGS

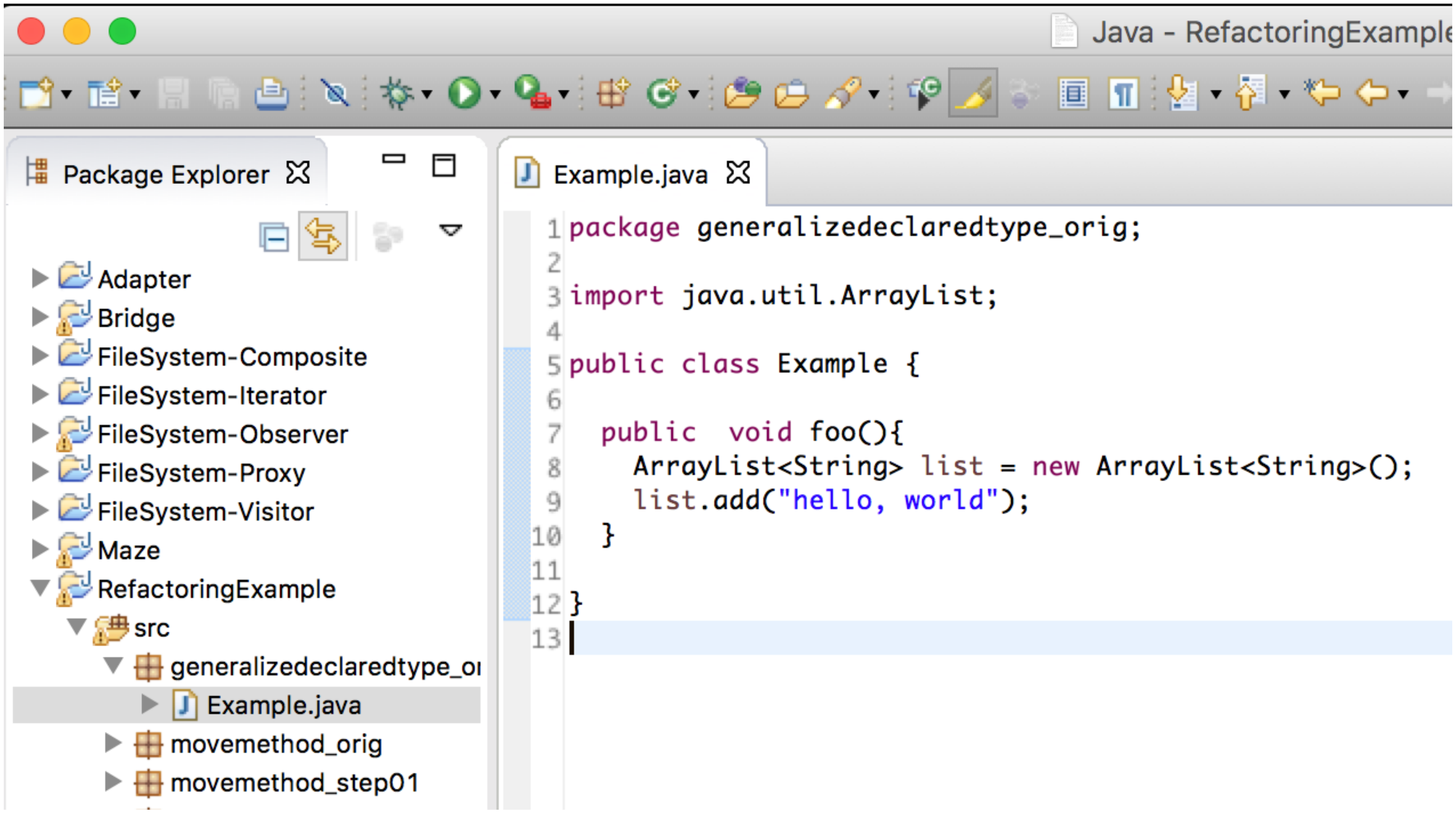
- refactorings for changing the class hierarchy and/or the types of declarations of variables and fields
- purpose is to make designs more flexible, e.g., by facilitating the introduction of design patterns

Generalize Declared Type	replace the type of a declaration with a more general type
Extract Interface	create a new interface, and update declarations to use it where possible
Pull Up Members	move methods and fields to a superclass
Infer Generic Type Arguments	infer type arguments for “raw” uses of generic types

GENERALIZE DECLARED TYPE

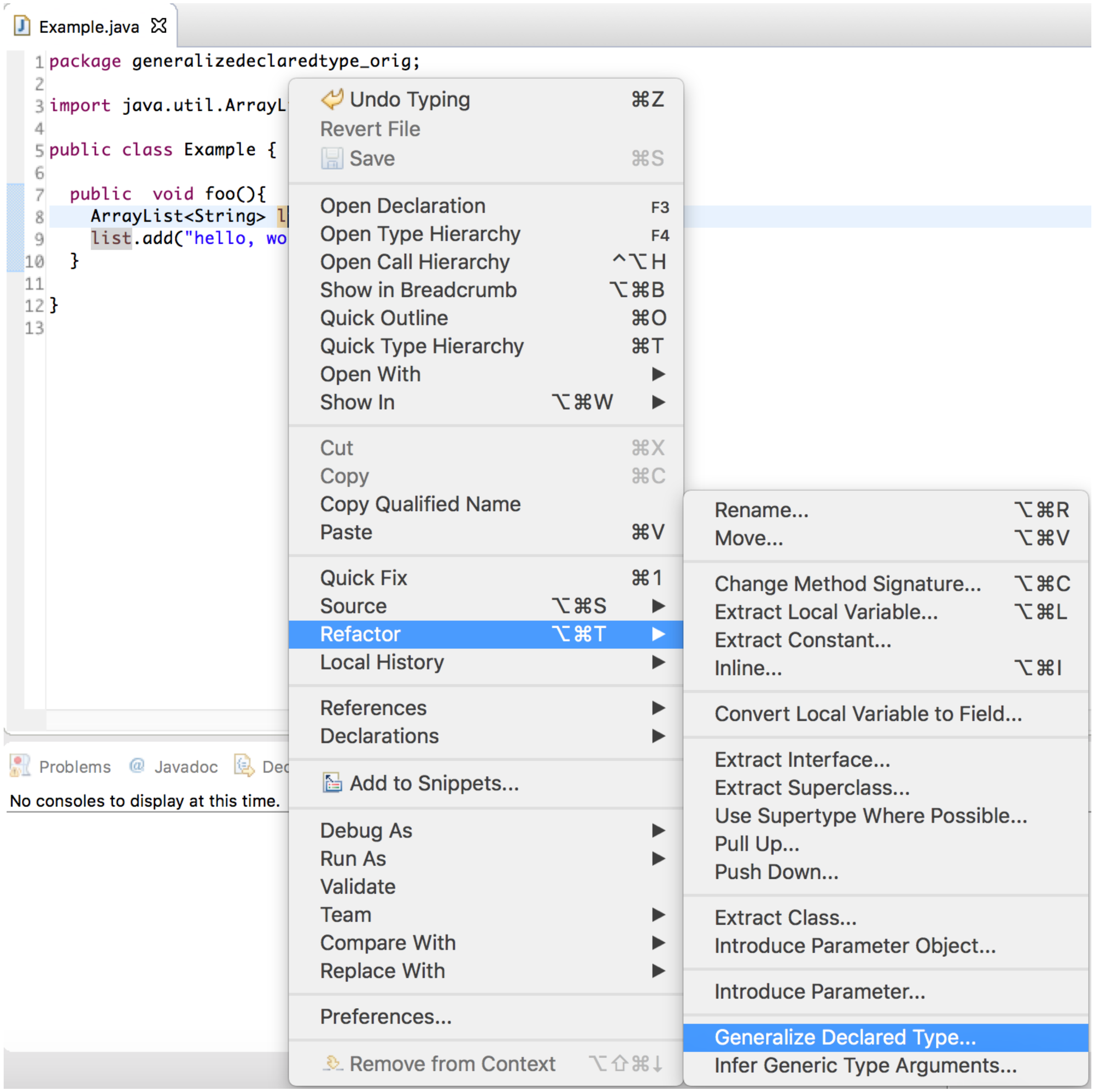
```
public  void foo(){
   ArrayList<String> list = new ArrayList<String>();
   list.add("hello, world");
 }
```

GENERALIZE DECLARED TYPE



```
package generalizedeclaredtype_orig;

import java.util.ArrayList;

public class Example {

  public  void foo(){
    ArrayList<String> list = new ArrayList<String>();
    list.add("hello, world");
  }

}

```

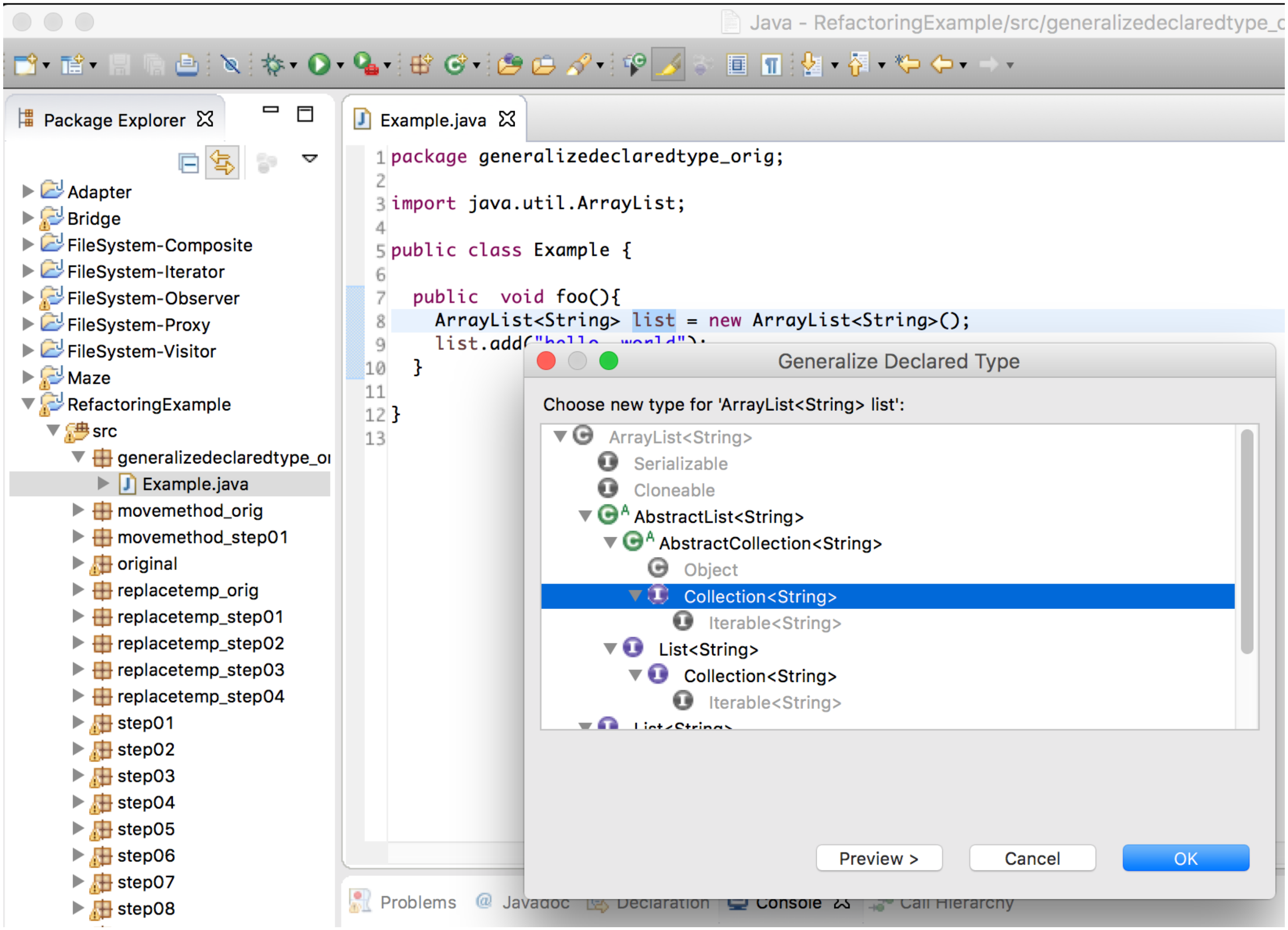
GENERALIZE DECLARED TYPE



```
package generalizedeclaredtype_orig;

import java.util.ArrayL

public class Example {

  public  void foo(){
    ArrayList<String> l
    list.add("hello, wo
  }

}
```

No consoles to display at this time.

GENERALIZE DECLARED TYPE

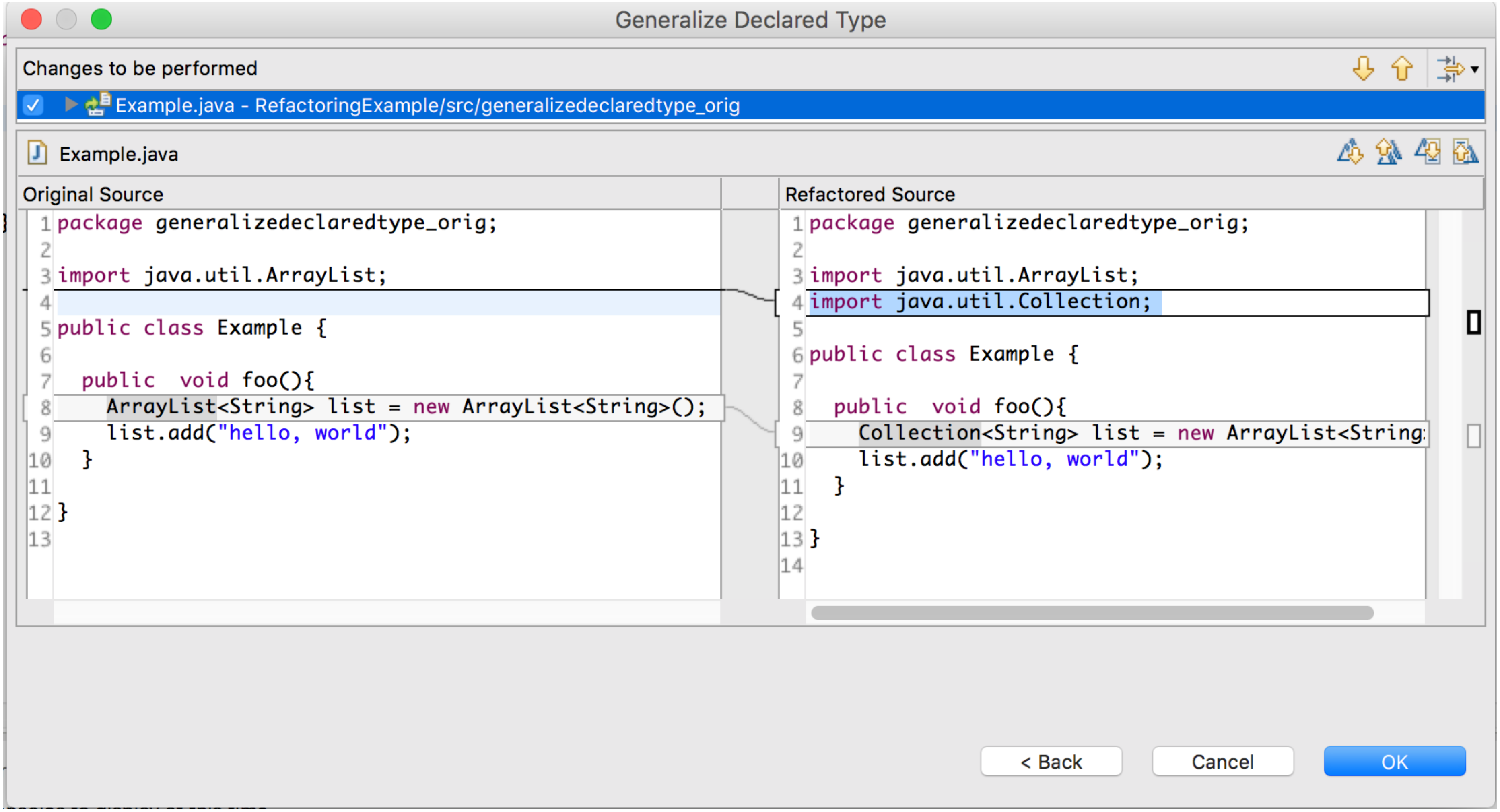


```java
package generalizedeclaredtype_orig;

import java.util.ArrayList;

public class Example {

  public  void foo(){
    ArrayList<String> list = new ArrayList<String>();
    list.add("hello, world");
  }

}
```

Generalize Declared Type

Choose new type for 'ArrayList<String> list':

GENERALIZE DECLARED TYPE



☑ Example.java - RefactoringExample/src/generalizedeclaredtype_orig

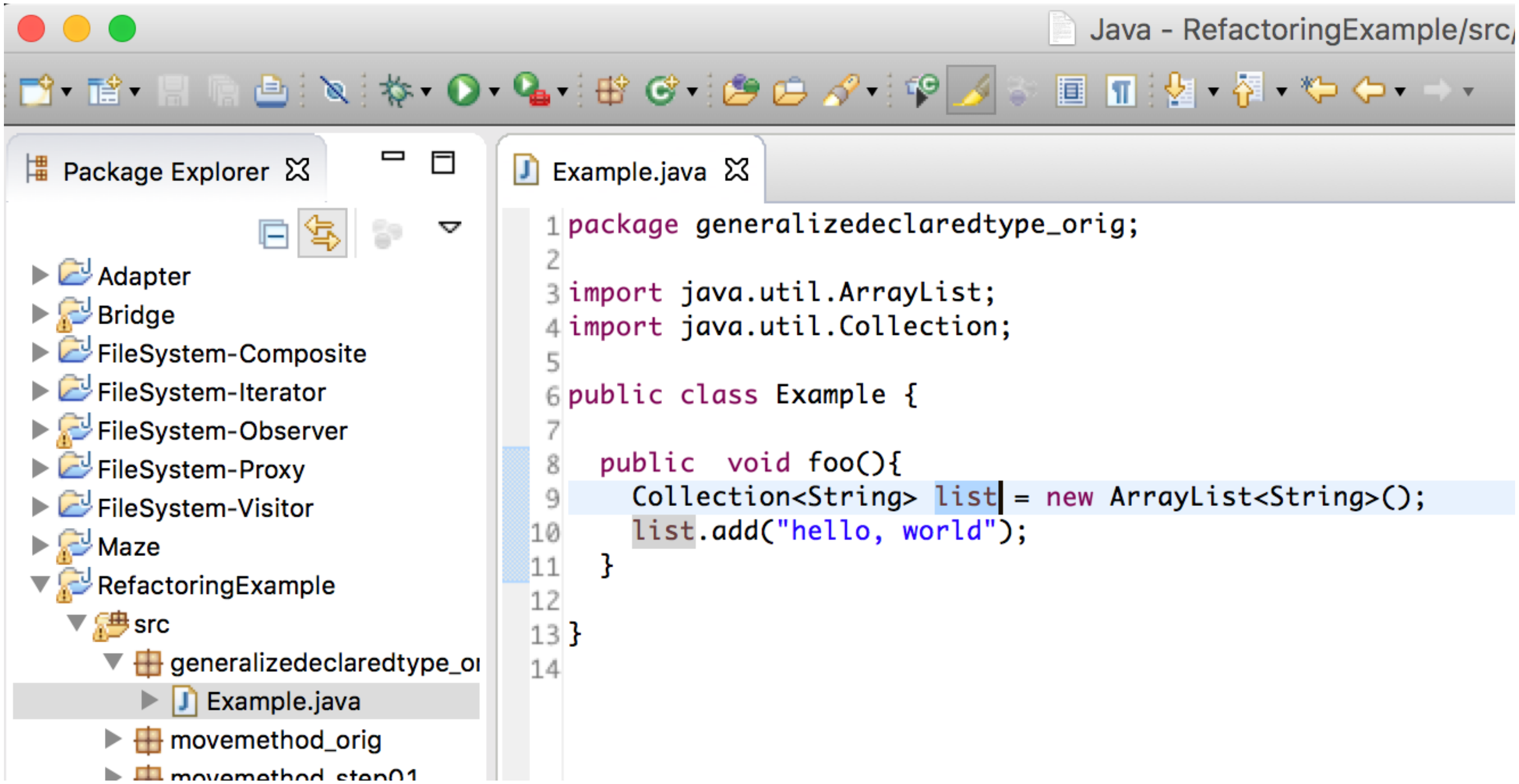
Original Source

```
package generalizedeclaredtype_orig;

import java.util.ArrayList;

public class Example {

  public  void foo(){
    ArrayList<String> list = new ArrayList<String>();
    list.add("hello, world");
  }

}

```

Refactored Source

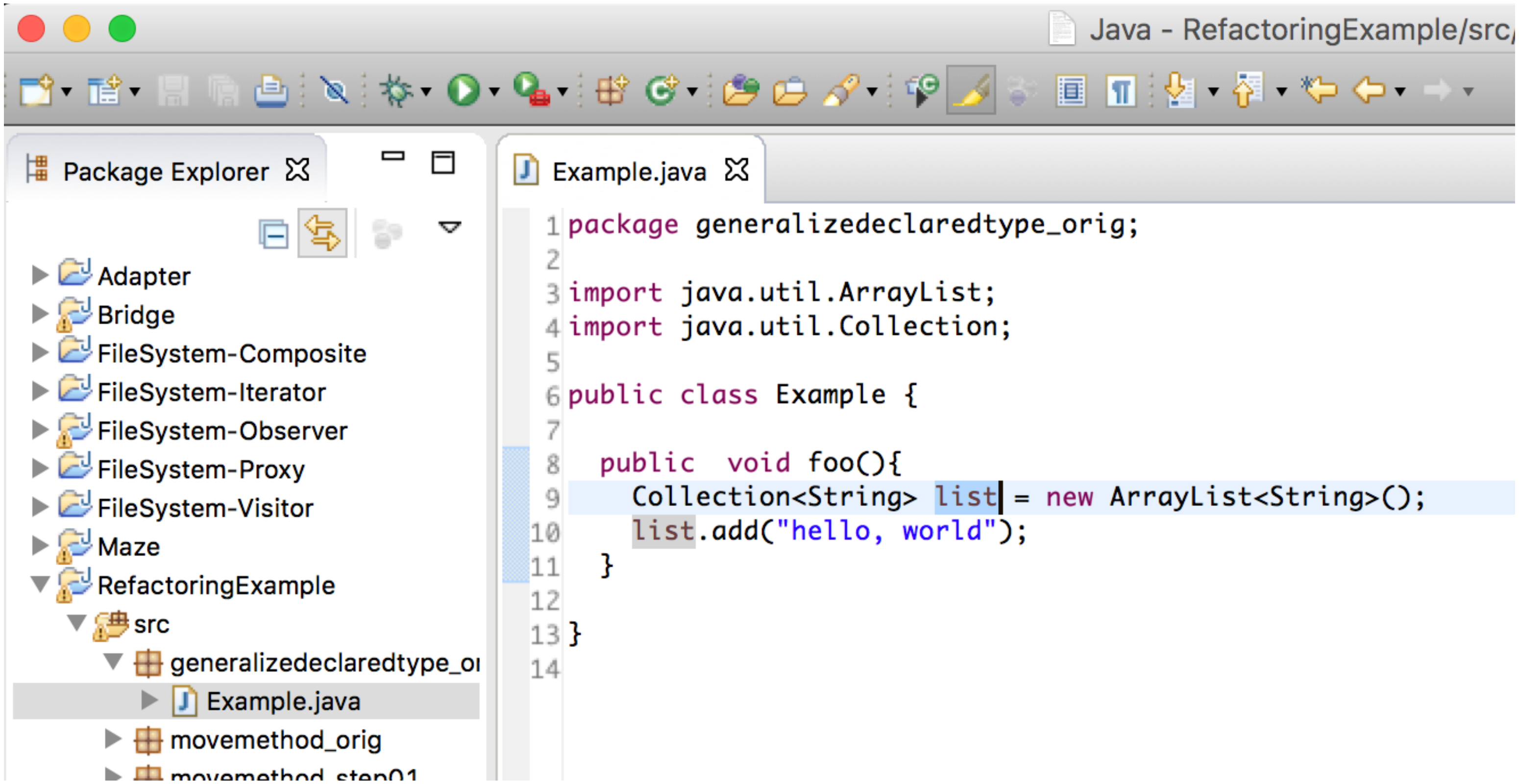
```
package generalizedeclaredtype_orig;

import java.util.ArrayList;
import java.util.Collection;

public class Example {

  public  void foo(){
    Collection<String> list = new ArrayList<String
    list.add("hello, world");
  }

}

```

GENERALIZE DECLARED TYPE



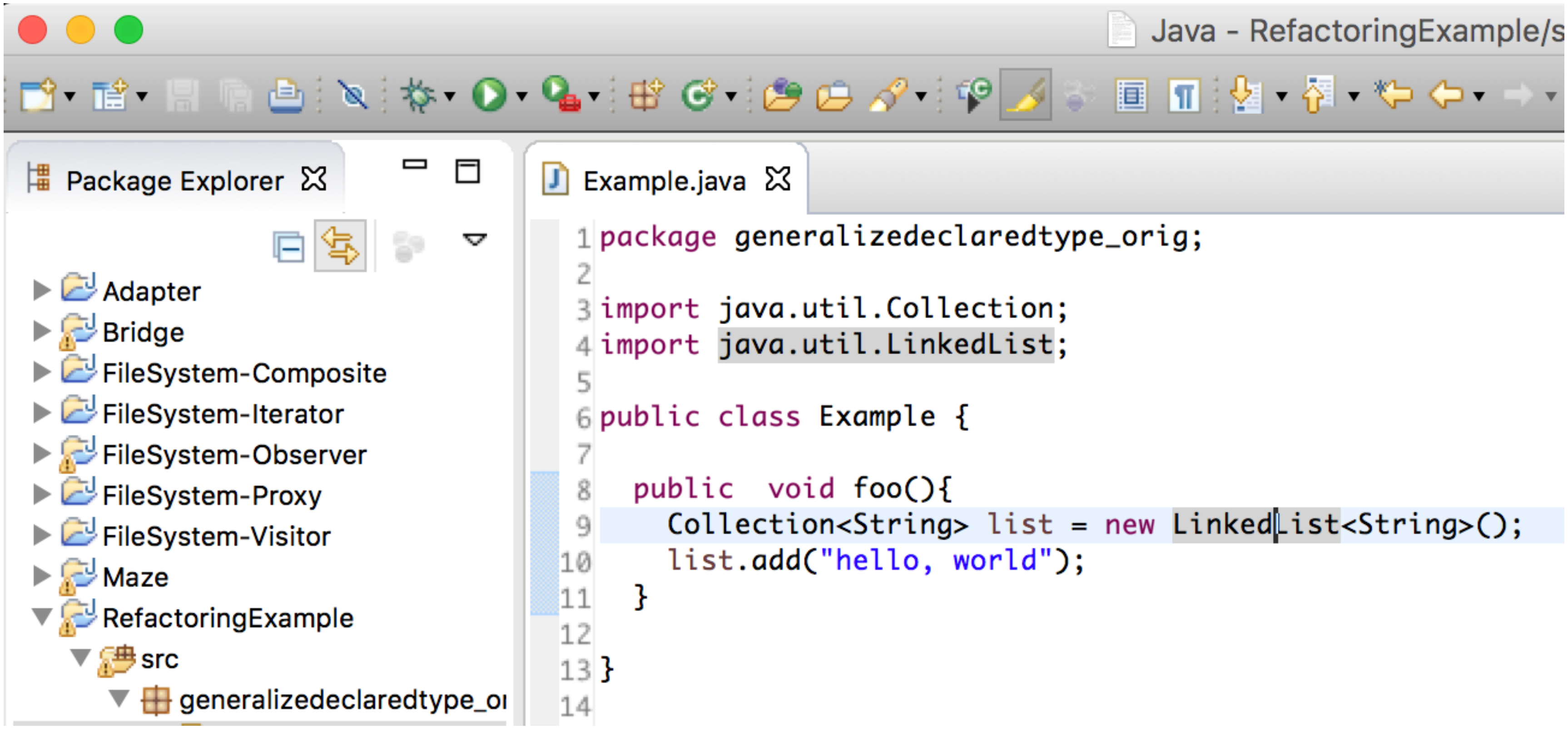
```java
package generalizedeclaredtype_orig;

import java.util.ArrayList;
import java.util.Collection;

public class Example {

  public  void foo(){
    Collection<String> list = new ArrayList<String>();
    list.add("hello, world");
  }

}

```

GENERALIZE DECLARED TYPE



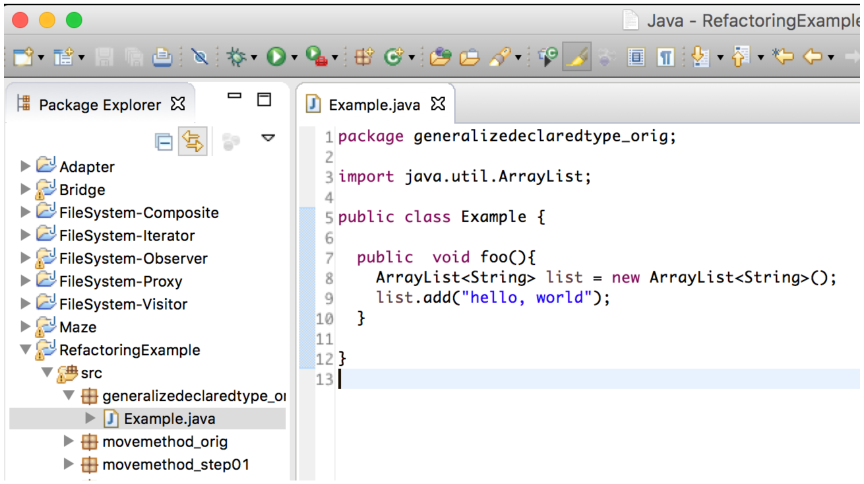
```java
package generalizedeclaredtype_orig;

import java.util.ArrayList;
import java.util.Collection;

public class Example {

  public  void foo(){
    Collection<String> list = new ArrayList<String>();
    list.add("hello, world");
  }

}
```

GENERALIZE DECLARED TYPE

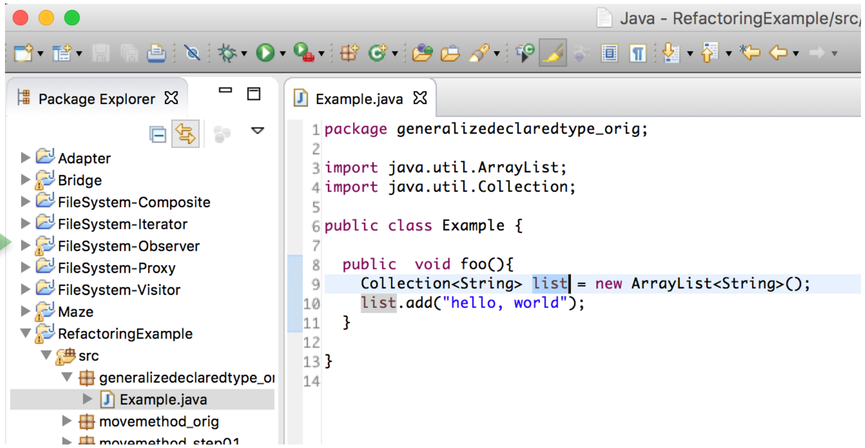


```
package generalizedeclaredtype_orig;

import java.util.Collection;
import java.util.LinkedList;

public class Example {

  public  void foo(){
    Collection<String> list = new LinkedList<String>();
    list.add("hello, world");
  }

}

```

GENERALIZE DECLARED TYPE



```java
package generalizedeclaredtype_orig;

import java.util.ArrayList;

public class Example {

  public  void foo(){
    ArrayList<String> list = new ArrayList<String>();
    list.add("hello, world");
  }

}
```



```java
package generalizedeclaredtype_orig;

import java.util.ArrayList;
import java.util.Collection;

public class Example {

  public  void foo(){
    Collection<String> list = new ArrayList<String>();
    list.add("hello, world");
  }

}
```

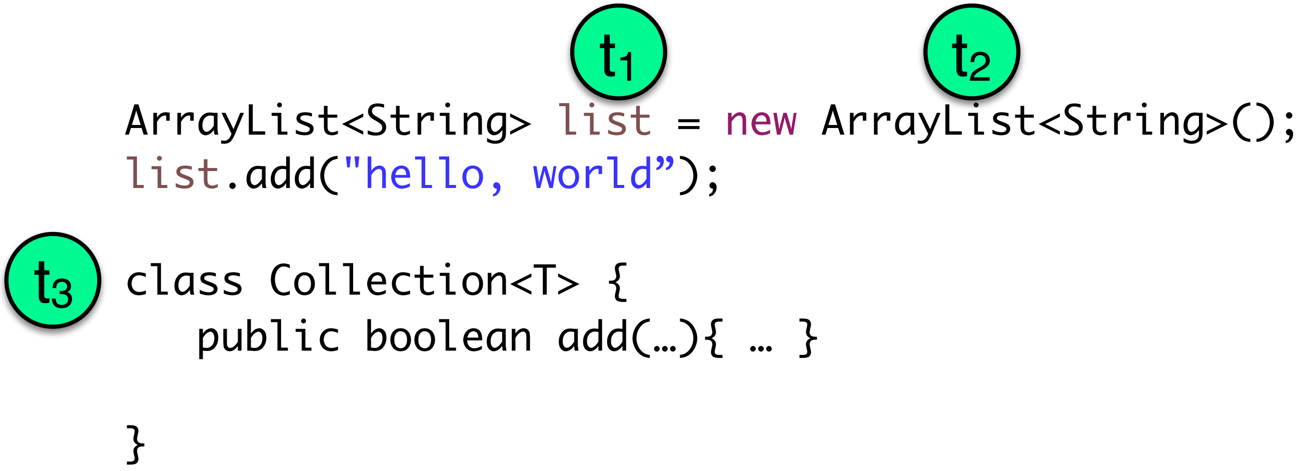
- So how does Eclipse figure out which types can be used?

TYPE CONSTRAINTS

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world");
```

- Eclipse applies **static analysis** to determine relationships between the types of variables and expressions that should be preserved

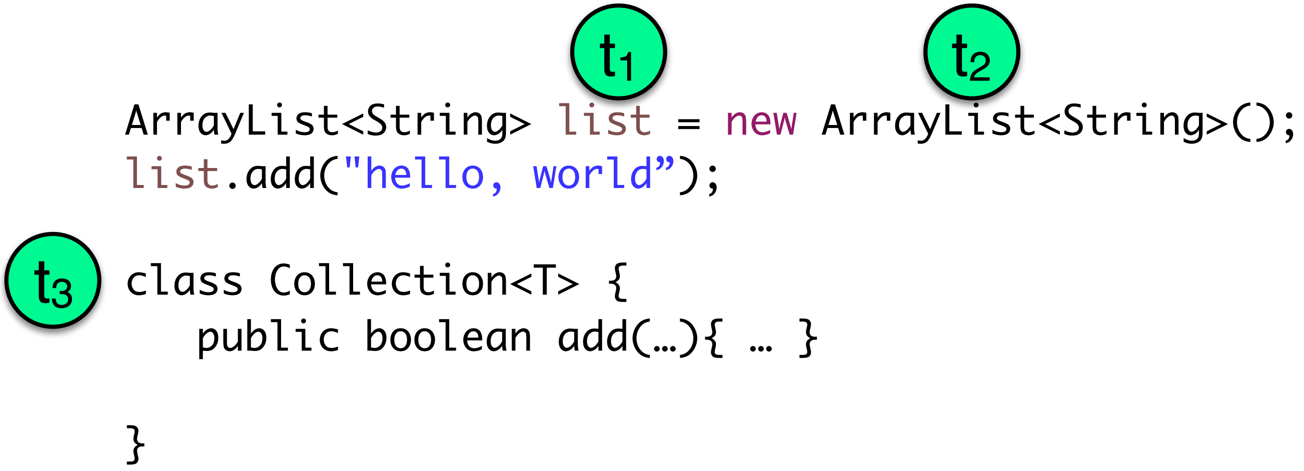
TYPE CONSTRAINTS



```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world”);

class Collection<T> {
   public boolean add(…){ … }

}
```

- Eclipse applies **static analysis** to determine relationships between the types of variables and expressions that should be preserved. Let’s say that:
  - **t_1** represents the type of variable list
  - **t_2** represents the type of expression `new ArrayList<String>()`
  - **t_3** represents the most general type in which `add()` is declared

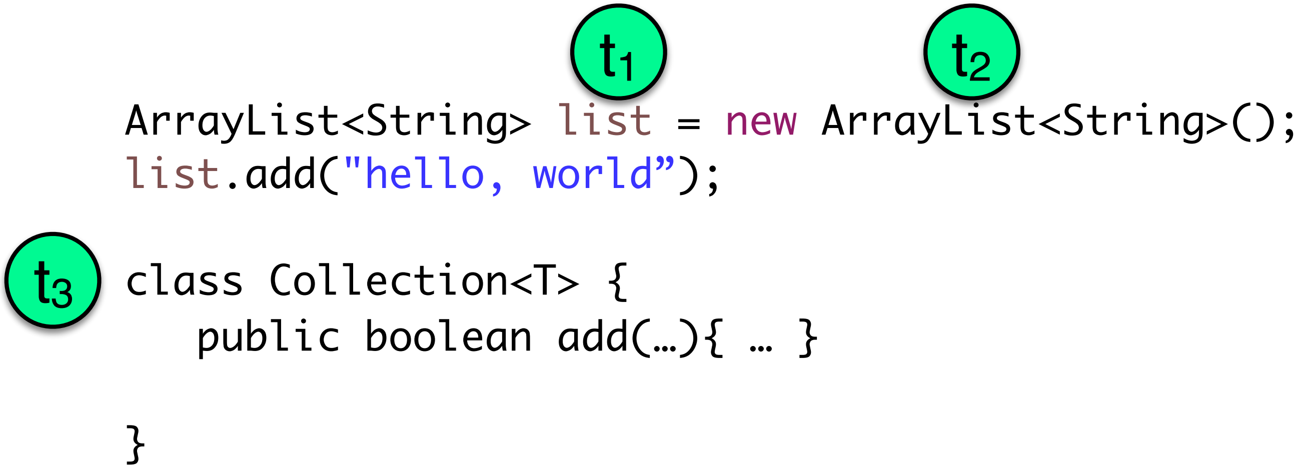
TYPE CONSTRAINTS



```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world”);

class Collection<T> {
   public boolean add(…){ … }

}
```

- Now, ignoring the specific type used for variable list, we can observe that this program is type correct if:
  - **t_2** is a subtype of **t_1**, and
  - **t_1** is a subtype of **t_3**

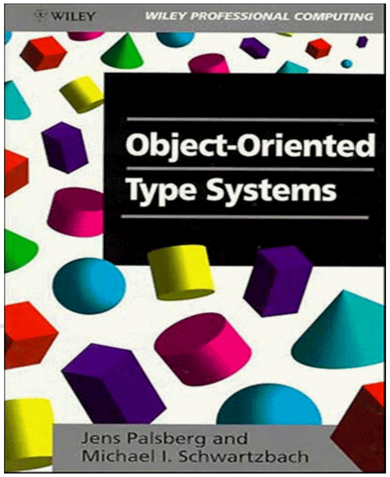
TYPE CONSTRAINTS



```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world”);

class Collection<T> {
   public boolean add(…){ … }

}
```

- Now, ignoring the specific type used for variable list, we can observe that this program is type correct if:
  - $t_2 \leq t_1$, and
  - $t_1 \leq t_3$

TYPE CONSTRAINTS



t_1 t_2

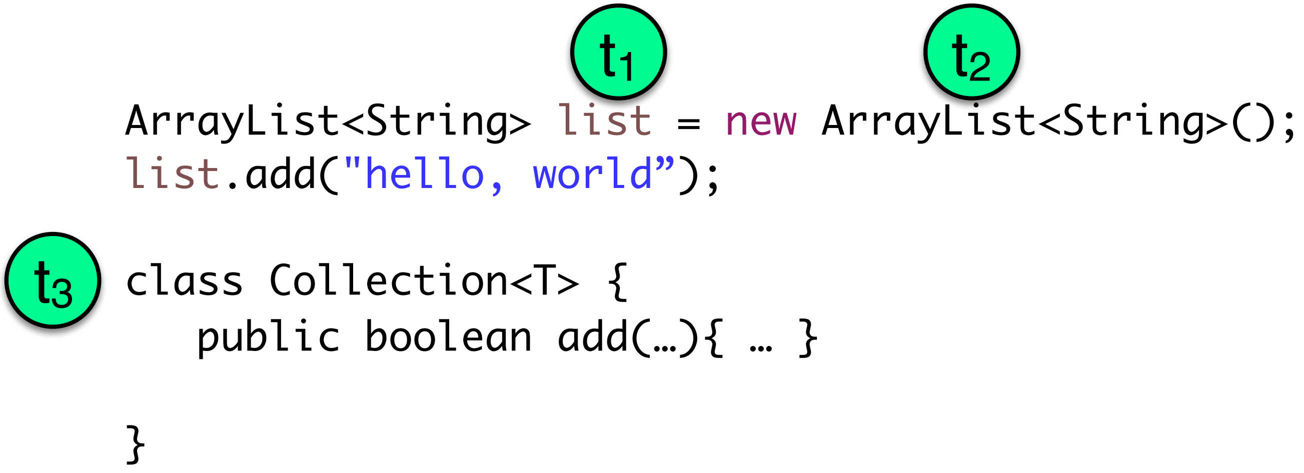
```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world”);
```

t_3

```
class Collection<T> {
   public boolean add(…){ … }

}
```

- Now, ignoring the specific type used for variable list, we can observe that this program is type correct if:
  - $t_2 \leq t_1$, and
  - $t_1 \leq t_3$
- From the program, we can see that:
  - t_2 = ArrayList
  - t_3 = Collection

CONSTRAINT SOLVING



```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world”);

class Collection<T> {
   public boolean add(…){ … }

}
```

- to find permissible types for **t_1**, Eclipse needs to **solve** the following system of constraints:
  - **$t_2 \leq t_1$**
  - **$t_1 \leq t_3$**
  - **t_2** = `ArrayList`
  - **t_3** = `Collection`

CONSTRAINT SOLVING

t_1

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world”);
```

solution must satisfy:

$\texttt{ArrayList} \leq t_1 \leq \texttt{Collection}$

CONSTRAINT SOLVING

t_1

```
ArrayList<String> list = new ArrayList<String>();
list.add("hello, world”);
```

solution must satisfy:

ArrayList $\leq t_1 \leq$ Collection

Collection

AbstractCollection

AbstractList

List

ArrayList

EXTRACT INTERFACE

- we'll consider an example where:
  - class **`Stack`** that defines methods like **`push()`**, **`pop(`**, etc.
  - class **`Client`** that refers to concrete type **`Stack`**
  - we'll use the **Extract Interface** refactoring to extract an interface **`IStack`** from **`Stack`**

EXTRACT INTERFACE

```
class Stack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    classreturn
  v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack  s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack  s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
             e.nextElement());
  }
}
```

```
class Client {
  public static
      void main(String[] args){
  Stack  s1 = new Stack();
  s1.push(new Integer(1));
  Stack s2 = new Stack();
  s2.push(new Float(2.2));
  s2.push(new Float(3.3));
  s1.moveFrom(s2);
  s2.moveTo(s1);
  Stack.print(s2);
  Vector v1 = new Vector();
  while (!s1.isEmpty()){
    Number n = (Number)s1.pop();
    v1.add(n);
  }
  // rest of code same as before..
}
Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    classreturn
  v2.remove(v2.size()-1);
  }
  public
    this
  }
  public
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
             e.nextElement());
  }
}
```

```
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(Stack  s6);
  public void moveTo(Stack  s7);
  public boolean isEmpty();
}
```

```
class Client {
  public static
      void main(String[] args){
    Stack  s1 = new Stack();
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Number n = (Number)s1.pop();
      v1.add(n);
    }
    // rest of code same as before..
}
```

Example: Extract interface IStack from Stack

- create interface IStack that declares all instance methods of Stack
- make IStack a supertype of Stack
- where possible, update declarations to refer to IStack instead of Stack

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack  s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack  s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
             e.nextElement());
  }
}
```

```
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(Stack  s6);
  public void moveTo(Stack  s7);
  public boolean isEmpty();
}

class Client {
  public static
      void main(String[] args){
  Stack  s1 = new Stack();
  s1.push(new Integer(1));
  Stack s2 = new Stack();
  s2.push(new Float(2.2));
  s2.push(new Float(3.3));
  s1.moveFrom(s2);
  s2.moveTo(s1);
  Stack.print(s2);
  Vector v1 = new Vector();
  while (!s1.isEmpty()){
    Number n = (Number)s1.pop();
    v1.add(n);
  }
  // ...
}
```

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack  s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack  s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
               e.nextElement());
  }
}
```

```
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(Stack  s6);
  public void moveTo(Stack  s7);
  public boolean isEmpty();
}

class Client {
  public static
      void main(String[] args){
  Stack  s1 = new Stack();
  s1.push(new Integer(1));
  Stack s2 = new Stack();
  s2.push(new Float(2.2));
  s2.push(new Float(3.3));
  s1.moveFrom(s2);
  s2.moveTo(s1);
  Stack.print(s2);
  Vector v1 = new Vector();
  while (!s1.isEmpty()){
    Number n = (Number)s1.pop();
    v1.add(n);
  }
  // ...
}
```

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack  s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack  s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
             e.nextElement());
  }
}
```

```
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(Stack  s6);
  public void moveTo(Stack  s7);
  public boolean isEmpty();
}

class Client {
  public static
      void main(String[] args){
  Stack  s1 = new Stack();
  s1.push(new Integer(1));
  Stack s2 = new Stack();
  s2.push(new Float(2.2));
  s2.push(new Float(3.3));
  s1.moveFrom(s2);
  s2.moveTo(s1);
  Stack.print(s2);
  Vector v1 = new Vector();
  while (!s1.isEmpty()){
    Number n = (Number)s1.pop();
    v1.add(n);
  }
  // ...
}
```

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack  s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack  s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
             e.nextElement());
  }
}
```

```
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(Stack  s6);
  public void moveTo(Stack  s7);
  public boolean isEmpty();
}

class Client {
  public static
      void main(String[] args){
  Stack  s1 = new Stack();
  s1.push(new Integer(1));
  Stack s2 = new Stack();
  s2.push(new Float(2.2));
  s2.push(new Float(3.3));
  s1.moveFrom(s2);
  s2.moveTo(s1);
  Stack.print(s2);
  Vector v1 = new Vector();
  while (!s1.isEmpty()){
    Number n = (Number)s1.pop();
    v1.add(n);
  }
  // ...
}
```

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(IStack s3){
    this.push(s3.pop());
  }
  public void moveTo(IStack s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
    return v2.isEmpty();
  }
  public static void print(Stack s5){
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
             e.nextElement());
  }
}
```

```
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(IStack s6);
  public void moveTo(IStack s7);
  public boolean isEmpty();
}

class Client {
  public static
      void main(String[] args){
  IStack s1 = new Stack();
  s1.push(new Integer(1));
  Stack s2 = new Stack();
  s2.push(new Float(2.2));
  s2.push(new Float(3.3));
  s1.moveFrom(s2);
  s2.moveTo(s1);
  Stack.print(s2);
  Vector v1 = new Vector();
  while (!s1.isEmpty()){
    Number n = (Number)s1.pop();
    v1.add(n);
  }
  // ...
}
```

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v
  }
  public void moveFrom
    this.push(s3.pop()
  }
  public void moveTo(S
    s4.push(this.pop()
  }
  public boolean isEmp
    return v2.isEmpty(
  }
  public static void p
    Enumeration e = s5.v2.elements();
    for ( ; e.hasMoreElements(); )
      System.out.println(
             e.nextElement());
  }
}
```

```
interface IStack {
  public void push(Object o);
  public Object pop();
  public void moveFrom(Stack  s6);
  public void moveTo(Stack  s7);
  public boolean isEmpty();
}

          nt {
          tatic
           main(String[] args){
          1 = new Stack();
          new Integer(1));
           = new Stack();
          new Float(2.2));
          new Float(3.3));
          rom(s2);
          o(s1);
          int(s2);
          1 = new Vector();
          s1.isEmpty()){
    Number n = (Number)s1.pop();
    v1.add(n);
  }
  // rest of code same as before..
}
```

Stack ≤ [s1]

[s1] ≤ IStack

Stack ≤ [s2]

[s2] ≤ IStack

[s2] ≤ [s3]

[s1] ≤ [s4]

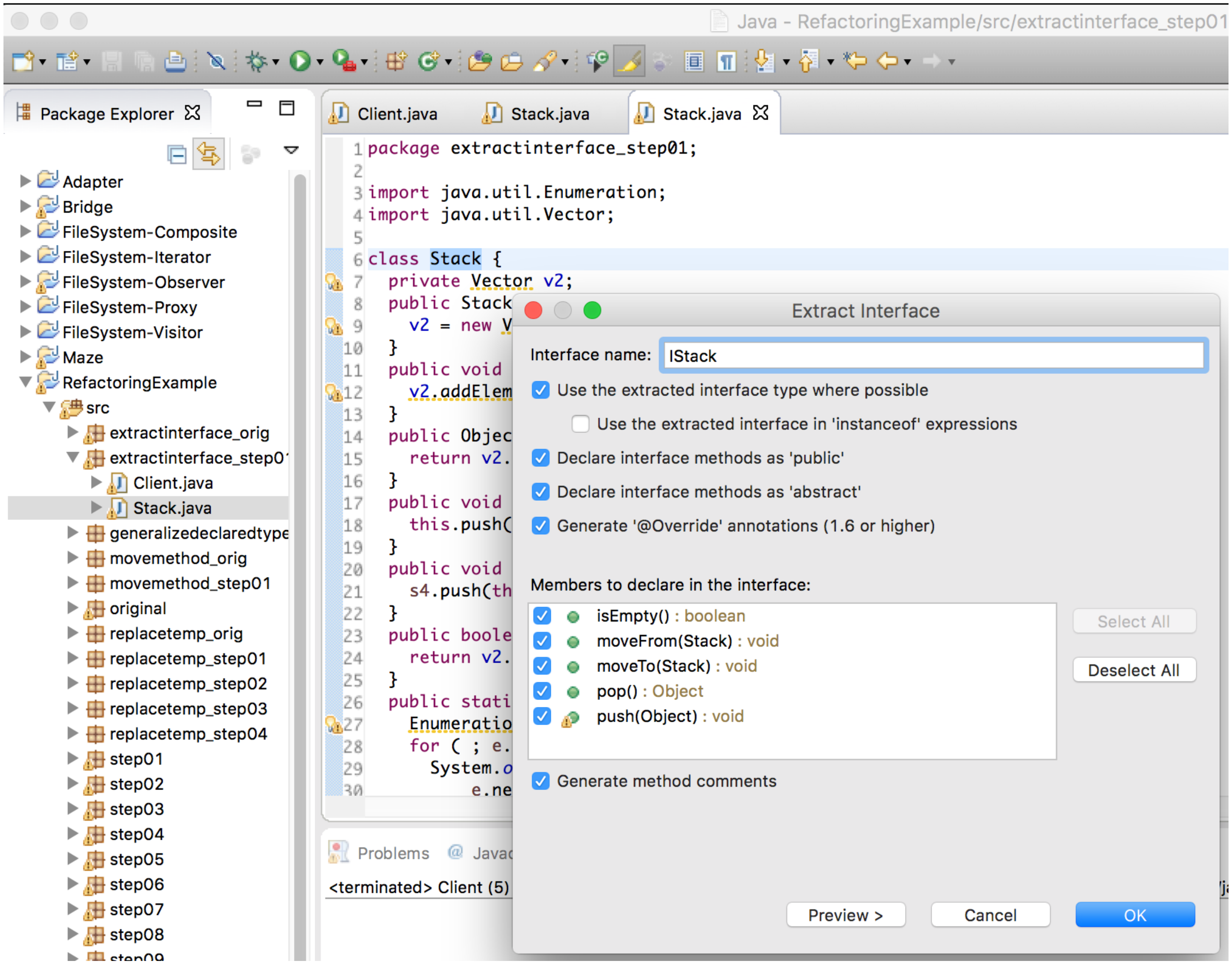
[s2] ≤ [s5]

[s3] ≤ IStack

[s4] ≤ IStack

[s5] ≤ Stack

EXTRACT INTERFACE



```
package extractinterface_step01;

import java.util.Enumeration;
import java.util.Vector;

class Stack {
  private Vector v2;
  public Stack
    v2 = new V
  }
  public void
    v2.addElem
  }
  public Objec
    return v2.
  }
  public void
    this.push(
  }
  public void
    s4.push(th
  }
  public boole
    return v2.
  }
  public stati
    Enumeratio
    for ( ; e.
      System.o
        e.ne
```

<terminated> Client (5)

Extract Interface

Interface name: IStack

☑ Use the extracted interface type where possible

☐ Use the extracted interface in 'instanceof' expressions

☑ Declare interface methods as 'public'

☑ Declare interface methods as 'abstract'

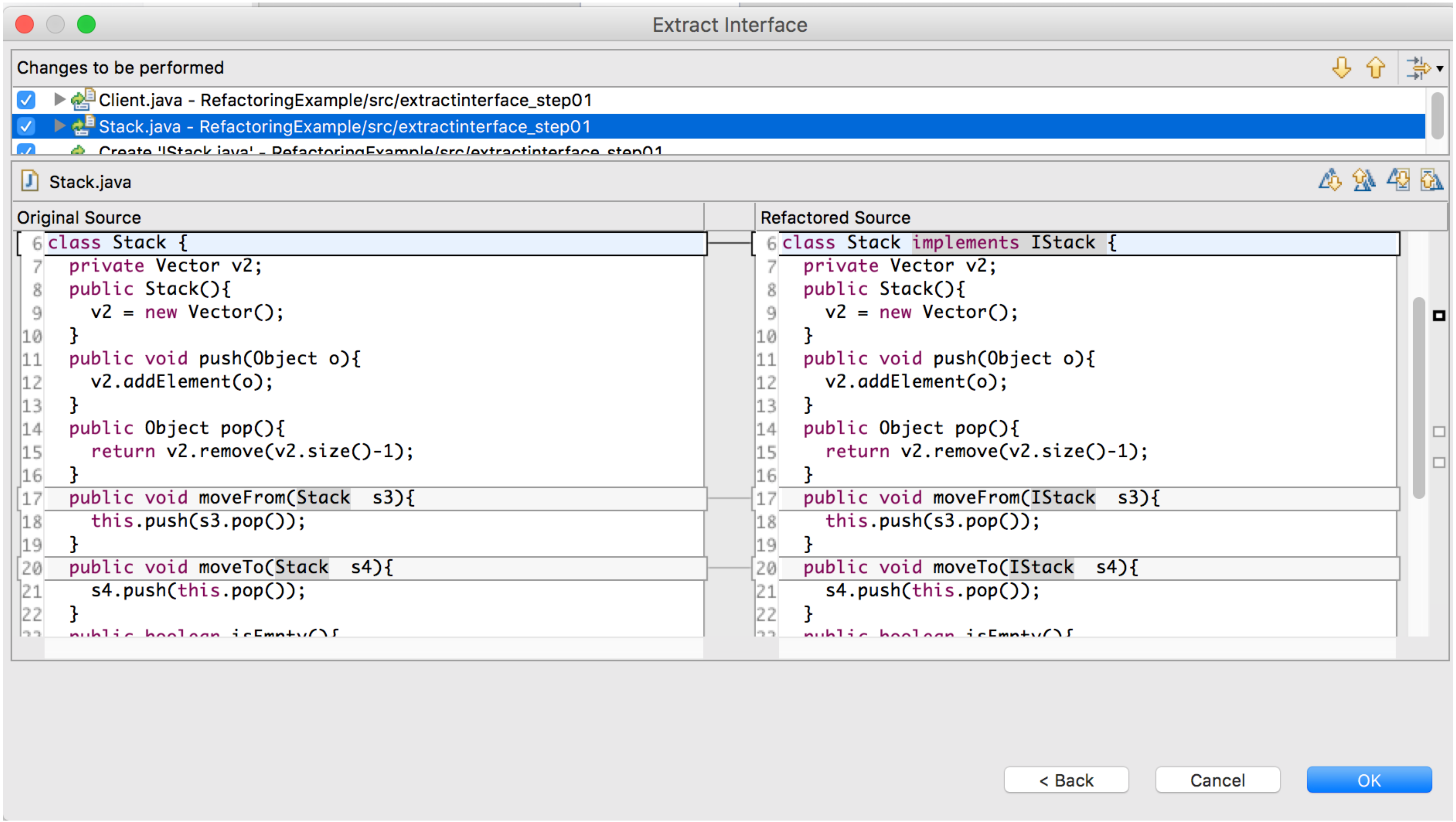
☑ Generate '@Override' annotations (1.6 or higher)

Members to declare in the interface:

- ☑ isEmpty() : boolean
- ☑ moveFrom(Stack) : void
- ☑ moveTo(Stack) : void
- ☑ pop() : Object
- ☑ push(Object) : void

☑ Generate method comments

EXTRACT INTERFACE



☑ Client.java - RefactoringExample/src/extractinterface_step01
☑ Stack.java - RefactoringExample/src/extractinterface_step01
☑ Create 'IStack.java' - RefactoringExample/src/extractinterface_step01

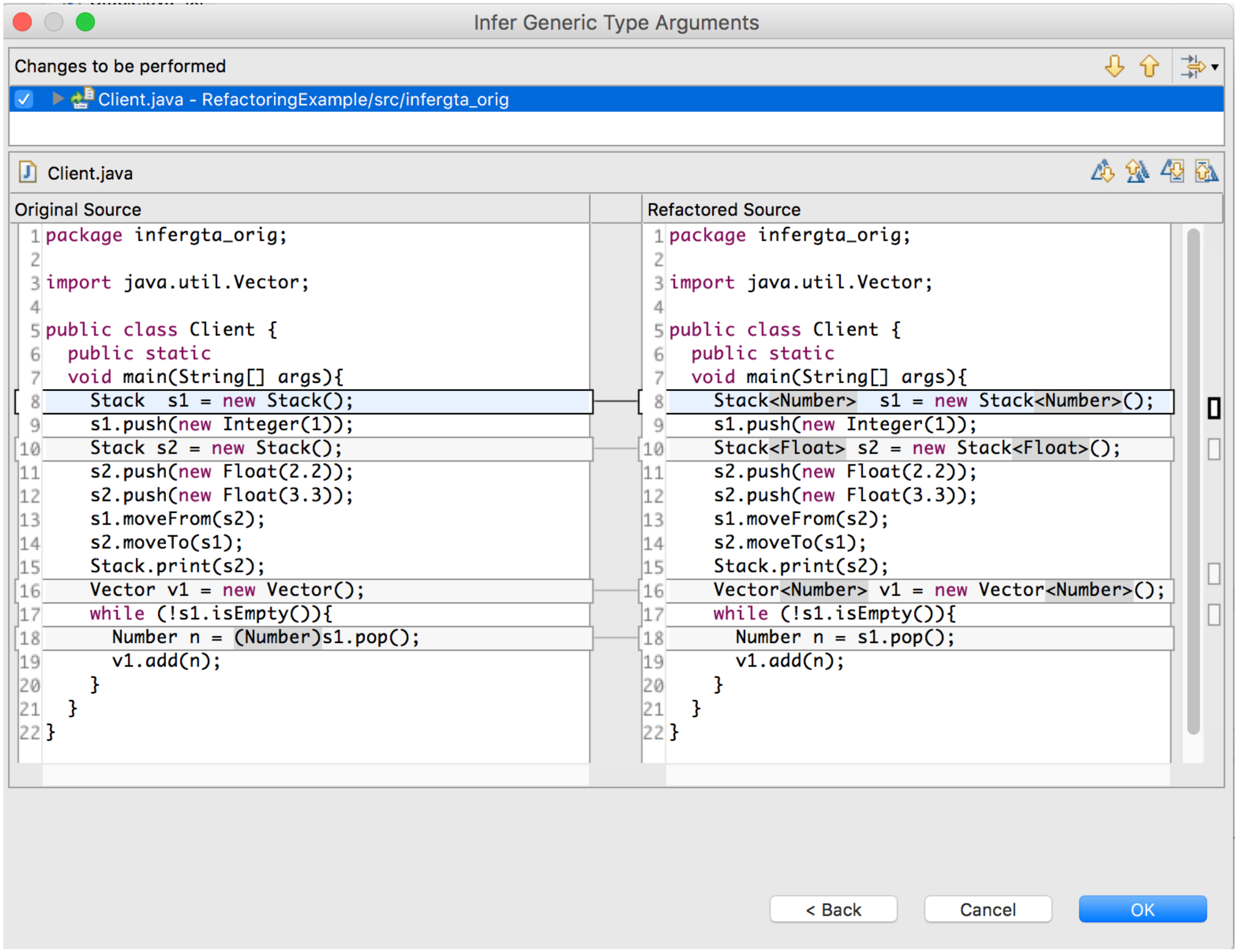
Original Source

```
class Stack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(Stack  s3){
    this.push(s3.pop());
  }
  public void moveTo(Stack  s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
```

Refactored Source

```
class Stack implements IStack {
  private Vector v2;
  public Stack(){
    v2 = new Vector();
  }
  public void push(Object o){
    v2.addElement(o);
  }
  public Object pop(){
    return v2.remove(v2.size()-1);
  }
  public void moveFrom(IStack  s3){
    this.push(s3.pop());
  }
  public void moveTo(IStack  s4){
    s4.push(this.pop());
  }
  public boolean isEmpty(){
```

INFER GENERIC TYPE ARGUMENTS


Infer Generic Type Arguments
Changes to be performed
☑ Client.java - RefactoringExample/src/infergta_orig
Client.java
Original Source
Refactored Source
< Back
Cancel
OK

Original Source

```java
package infergta_orig;

import java.util.Vector;

public class Client {
  public static
  void main(String[] args){
    Stack  s1 = new Stack();
    s1.push(new Integer(1));
    Stack s2 = new Stack();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector v1 = new Vector();
    while (!s1.isEmpty()){
      Number n = (Number)s1.pop();
      v1.add(n);
    }
  }
}
```

Refactored Source

```java
package infergta_orig;

import java.util.Vector;

public class Client {
  public static
  void main(String[] args){
    Stack<Number>  s1 = new Stack<Number>();
    s1.push(new Integer(1));
    Stack<Float> s2 = new Stack<Float>();
    s2.push(new Float(2.2));
    s2.push(new Float(3.3));
    s1.moveFrom(s2);
    s2.moveTo(s1);
    Stack.print(s2);
    Vector<Number> v1 = new Vector<Number>();
    while (!s1.isEmpty()){
      Number n = s1.pop();
      v1.add(n);
    }
  }
}
```

RECOMMENDATIONS

- continuously refactor to keep your code readable, understandable, and maintainable
    - by eliminating small problems soon, you can avoid big trouble later
- familiarize yourself with the automated refactoring support
    - this will save you time in the long run