On the automation of challenging refactorings through advanced method extraction techniques

Ran Ettinger (RanE@il.ibm.com), IBM Haifa Research Lab
PL/SE talk @ IBM WRC
11 June 2010
Ongoing joint work with Aharon Abadi, Yishai Feldman, and Mati Shomrat
Extract Computation: A Slice-Extraction Refactoring

• Make a slice of code reusable
  – Not merely copying and pasting it
  – Update the original code too, like in the “Extract Method” refactoring

• Turn a non-contiguous slice into a contiguous fragment of code, before applying “Extract Method”
  – Rearrange the rest of the code
    • Prepare parameters
    • Use slice results
    • Prevent unwanted side-effects
    • Compute further results
  – A kind of defragmentation
int start = page * 20;
int end = start + 20;
end = Math.min(end, album.getPictures().size());
for (int i = start; i < end; i++) {
  Picture picture = album.getPicture(i);
  printPicture(out, picture);
}
out.println("</table>");

void display(PrintStream out, int start, int end, Queue<Picture> pictures) {
  out.println("<table border=0">
               
  for (int i = start; i < end; i++) {
    Picture picture = album.getPicture(i);
    pictures.add(picture);
    printPicture(out, pictures.remove());
  }
  out.println("</table>");
}
A new slicing technique to assist in automated computation extraction

- Informally, a traditional (backward) slice of a given program with respect to selected “interesting” variables is a subprogram that computes the same values as the original program for the selected variables.

- Accordingly, a (backward) fine slice of a given program with respect to selected “interesting” variables and other “oracle” variables is a subprogram that computes the same values as the original program for the selected variables, given values for the oracle variables.
Fine Slicing

- A generalization of traditional program slicing
- Fine slices can be precisely bounded
  - Slicing criteria include set of data and control dependences to ignore
- Fine slices are **executable** and **extractable**
- Oracle-based semantics for fine slices
- Algorithm for computing data-structure representing the oracle
- Forward fine slices are executable
  - Might be larger than traditional forward slices
Extract Computation

• A new refactoring
• Extracts a fine slice into contiguous code
• Computes the co-slice
• Computation can then be extracted into a separate method using Extract Method
• Passes necessary “oracle” variables between slice and co-slice
• Generates new containers if series of values need to be passed
Extract Computation as a Building Block

• Examples of basic refactorings from Martin Fowler’s catalog
  – Split Loop
  – Replace Temp with Query
  – Separate Query from Modifier
  – Extract Method (non-contiguous flavor, e.g. near-clone elimination)

• Examples of refactorings to patterns from Joshua Kerievsky’s catalog
  – Compose Method
  – Form Template Method

• Examples of big refactorings from Martin Fowler and Kent Beck
  – Separate Domain from Presentation
  – Convert Procedural Design to Objects
Split Loop [Fowler]

• **Situation**: “You have a loop that is doing two things.”

• **Recommendation**: “Duplicate the loop.”

• **Link**: [http://www.refactoring.com/catalog/splitLoop.html](http://www.refactoring.com/catalog/splitLoop.html)
Before Split Loop on `averageAge`

```java
void printValues() {
    double averageAge = 0;
    double totalSalary = 0;
    for (int i = 0; i < people.length; i++) {
        averageAge += people[i].age;
        totalSalary += people[i].salary;
    }
    averageAge = averageAge / people.length;
    System.out.println(averageAge);
    System.out.println(totalSalary);
}
```
After Split Loop on `averageAge`

```java
void printValues() {
    double totalSalary = 0;
    for (int i = 0; i < people.length; i++) {
        totalSalary += people[i].salary;
    }
    double averageAge = 0;
    for (int i = 0; i < people.length; i++) {
        averageAge += people[i].age;
    }
    averageAge = averageAge / people.length;
    System.out.println(averageAge);
    System.out.println(totalSalary);
}
```
Split Loop: Mechanics

1. Identify the fragments in and above a loop statement that constitute a single computation and perform Extract Computation on those fragments
   - Choose between placing the extracting computation above or below its complement, based on whether it needs input from the complement or provides output to it
Split Loop: Challenges

• Advanced version: introduce new collections to store (and later retrieve) intermediate values
• Initialization code (above the loop)
• UI: Mechanism to select values of interest
• Move selected computation ahead or below?
• Exiting jumps and syntax preservation
Advanced Split Loop Example

```java
for (int i = start; i < end; i++) {
    Picture picture = album.getPicture(i);
    printPicture(out, picture);
}

Queue<Picture> pictures = new LinkedList<Picture>();
for (int i = start; I < end; i++) {
    Picture picture = album.getPicture(i);
    pictures.add(picture);
}
for (int i = start; i < end; i++) {
    printPicture(out, pictures.remove());
}
```
Replace Temp with Query (RTwQ) [Fowler]

• **Situation**: “You are using a temporary variable to hold the result of an expression.”

• **Recommendation**: “Extract the expression into a method. Replace all references to the temp with the expression. The new method can then be used in other methods.”

• **Link**: [http://www.refactoring.com/catalog/replaceTempWithQuery.html](http://www.refactoring.com/catalog/replaceTempWithQuery.html)
class Customer ...  
public String statement() {  
    double totalAmount = 0;  
    int frequentRenterPoints = 0;  
    Enumeration rentals = _rentals.elements();  
    String result = “Rental Record for “ + getName() + “
”;
    while (rentals.hasMoreElements()) {  
      Rental each = (Rental) rentals.nextElement();  
      frequentRenterPoints += each.getFrequentRenterPoints();  
      // show figures for this rental  
      result += “\t” + each.getMovie().getTitle() + “\t” +  
                String.valueOf(each.getCharge()) + “\n”;
      totalAmount += each.getCharge();  
    }  
    // add footer lines  
    result += “Amount owed is “ +  
               String.valueOf(totalAmount) + “\n”;
    result += “You earned “ +  
               String.valueOf(frequentRenterPoints) + “frequent renter points\n”;
    return result;
}
After RTwQ on totalAmount

class Customer ...  
public String statement() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        frequentRenterPoints += each.getFrequentRenterPoints();
        //show figures for this rental
        result += "\t" + each.getMovie().getTitle() + "\t" +
            String.valueOf(each.getCharge()) + "\n";
    }
    // add footer lines
    result += "Amount owed is " +
           String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " +
            String.valueOf(frequentRenterPoints) +
            "frequent renter points\n";
    return result;
}
After RTwQ on totalAmount

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;
}
class Customer ...
public String statement() {
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = “Rental Record for “ + getName() + “
;while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        frequentRenterPoints += each.getFrequentRenterPoints();
        //show figures for this rental
        result += “\t” + each.getMovie().getTitle()+ “\t” +
            String.valueOf(each.getCharge()) + “\n”;
    }
    // add footer lines
    result += “Amount owed is “ +
        String.valueOf(getTotalCharge()) + “\n”;
    result += “You earned “ +
        String.valueOf(frequentRenterPoints) +
        “frequent renter points\n”;
    return result;
}
class Customer ...
public String statement() { 
    Enumeration rentals = _rentals.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” +
                  String.valueOf(each.getCharge()) + “\n”;
    }
    // add footer lines
    result += “Amount owed is “ +
              String.valueOf(getTotalCharge()) + “\n”;
    result += “You earned “ +
              String.valueOf(getFrequentRenterPoints()) +
              “frequent renter points\n”;
    return result;
}
After RTwQ on `frequentRenterPoints`

class Customer ...  
private double getFrequentRenterPoints() {
    int result = 0;
    Enumeration rentals = _rentals.elements();
    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += each.getFrequentRenterPoints();
    }
    return result;
}
RTwQ: Mechanics

1. Extract Computation for the temp starting from all points of final-use
2. Extract Method on the temp’s slice
3. Inline Temp
RTwQ: Challenges

• Allow side effects in query? Prevent them?
• Distinguish intermediate from final references
  – Only the latter will be replaced with a query
  – The former will be extracted (Why?)
• Scope for extraction
  – Follow scope of the extracted temp?
• Beware of Eclipse’s Inline Temp
Separate Query from Modifier (SQfM) [Fowler]

• **Situation:** “You have a method that returns a value but also changes the state of an object.”

• **Recommendation:** “Create two methods, one for the query and one for the modification.”

• **Link:**
Before SQfM on `foundMiscreant`

```java
String foundMiscreant(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            return "Don";
        }
        if (people[i].equals("John")) {
            sendAlert();
            return "John";
        }
    }
    return "";
}

void checkSecurity(String[] people) {
    String found = foundMiscreant(people);
    someLaterCode(found);
}
```
After SQfM on foundMiscreant

```java
void sendAlert(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            sendAlert();
            return;
        }
        if (people[i].equals("John")) {
            sendAlert();
            return;
        }
    }
}
```
String foundPerson(String[] people) {
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals("Don")) {
            return "Don";
        }
        if (people[i].equals("John")) {
            return "John";
        }
    }
    return "";
}

void checkSecurity(String[] people) {
    sendAlert(people);
    String found = foundPerson(people);
    someLaterCode(found);
}
SQfM: Mechanics

1. Extract Computation on returned value(s)
2. Extract Method twice (Q and M)
3. Inline Method (on the original method)
SQfM: Challenges

• Exiting jumps: Illegal Java in intermediate steps (need arbitrary jumps)
• Q-before-M vs. M-before-Q
• Meyer’s Command-Query-Separation
  – Storing Q’s result in M
• Original call might be embedded inconveniently (e.g. in a loop’s condition)
  – Inline Method must be improved!
Extract Method [Fowler]

• **Situation**: “You have a code fragment that can be grouped together.”

• **Recommendation**: “Turn the fragment into a method whose name explains the purpose of the method.”

• **Link**: [http://www.refactoring.com/catalog/extractMethod.html](http://www.refactoring.com/catalog/extractMethod.html)
Extract Method: Challenges

• Extract multiple fragments (into a single method)
  – Aka. non-contiguous code
  – Where to place the call?
  – Which statements to delete?
  – Parameters?
  – Backup variables and/or renaming?
• Clone elimination
• Extract incomplete fragments (i.e. with non-extracted holes)
• Loop untangling with second loop reusing intermediate values
• Exiting jumps
• MVC examples (see backup slides)
Compose Method [Kerievsky]

- **Situation**: “You can't rapidly understand a method's logic.”
- **Recommendation**: “Transform the logic into a small number of intention-revealing steps at the same level of detail.”
class List...
public void add(Object element) {
    if (!readOnly) {
        int newSize = size + 1;
        if (newSize > elements.length) {
            Object[] newElements = new Object[elements.length+10];
            for (int i = 0; i < size; i++)
                newElements[i] = elements[i];
            elements = newElements;
        }
        elements[size++] = element;
    }
}
class List...
public void add(Object element) {
    if (!readOnly) {
        return;
        if (atCapacity())
            grow();
        addElement(element);
    }
}

private void addElement(Object element) {
    elements[size++] = element;
}

private void grow() {
    Object[] newElements =
    new Object[elements.length + GROWTH_INCREMENT];
    for (int i = 0; i < size; i++)
        newElements[i] = elements[i];
    elements = newElements;
}

private boolean atCapacity() {
    return (size + 1) > elements.length;
}
Compose Method: Challenges

• No mechanics!
  – Kerievsky on Page 125: "This is one of the most important refactoring I know. Conceptually, it is also one of the simplest—so you’d think that this refactoring would lead to a simple set of mechanics. In fact, it’s just the opposite. While the steps themselves aren’t complex, there is no simple, repeatable set of steps. Instead, there are guidelines for refactoring to Composed Method, some of which include the following..."

• We should be able to come up with some mechanics...
  – Using Extract Computation as a building block
  – But what’s “same level of detail”?
  – Let the user choose the key variables/values whose computation should be made contiguous and then extracted
Form Template Method (FTM)

• Fowler
  – **Problem**: “You have two methods in subclasses that perform similar steps in the same order, yet the steps are different.”
  – **Recommendation**: “Get the steps into methods with the same signature, so that the original methods become the same. Then you can pull them up.”

• Kerievsky
  – **Situation**: “Two methods in subclasses perform similar steps in the same order, yet the steps are different.”
  – **Recommendation**: “Generalize the methods by extracting their steps into methods with identical signatures, then pull up the generalized methods to form a Template Method.”
FTM: Challenges

• Extract Computation as a preparatory step
  – A recurring theme in Kerievsky’s refactorings

• FTM vs. Extract Computation
  – No need for Template Method in simple cases
    • Avoid introducing new classes and inheritance
  – Similarity with aspect oriented programming
    (before, after or around)
  – Harder cases: Could try Extract Computation with collections (to pass sequences of intermediate values)
Convert Procedural Design to Objects
[Fowler & Beck]

• **Situation:** “You have code written in a procedural style.”

• **Recommendation:** “Turn the data records into objects, break up the behavior, and move the behavior to the objects.”

• **Link:** [http://sourcemaking.com/refactoring/convert-procedural-design-to-objects](http://sourcemaking.com/refactoring/convert-procedural-design-to-objects)
Separate Domain from Presentation
[Fowler & Beck]

• **Situation:** “You have GUI classes that contain domain logic.”

• **Recommendation:** “Separate the domain logic into separate domain classes.”

• **Link:** [http://sourcemaking.com/refactoring/separate-domain-from-presentation](http://sourcemaking.com/refactoring/separate-domain-from-presentation)
Previous Work on Behavior Preserving Procedure Extraction

- **Contiguous code**
  - Bill Opdyke's thesis (UIUC, 92): for C++
  - Griswold&Notkin (ToSE93): for Scheme

- **Arbitrary selections of (not necessarily contiguous) statements**
  - Tucking (Lakhotia&Deprez, IST98): the complement is a slice too (from all non-extracted points; no dataflow from the extracted slice to its complement yields over-duplication; strong preconditions (e.g., no global variables involved and no live-on-exit variable defined on both the slice and complement)
  - Semantics-Preserving Procedure Extraction (Komondoor&Horwitz, POPL00): considers all permutations of selected and surrounding statements; no duplication allowed; not practical (exponential time complexity)
  - Effective Automatic Procedure Extraction (Komondoor&Horwitz, IWPC03): improves on their POPL00 algorithm by being more practical (cubic time and space), allowing some duplication (of conditionals and jumps); might miss some correct permutations though; no duplication of assignments or loops; allows dataflow from complementary to extracted code and from extracted code to (a second portion of) complementary code; supports exiting jumps
  - Extraction of block-based slices (Maruyama, SSR01): extracts a slice of one variable only; restricted to structured code; no proof given
  - My thesis (Sliding, 2006): sliding transformation, sequentially composes a slice and its complement, allowing dataflow from the former to the latter; supports loop untangling and duplication of assignments; but restricted to slicing from the end, and only final values from the extracted slice can be reused in the complement; proof for toy language
Conclusion

• Program slicing can help in building automated tools for refactoring
• Fine slicing is a generalization of program slicing
  – It can be used to compute meaningful sub-programs
    • For extraction, in automated refactoring tools
    • For program understanding too
  – Can be used to compute the complementary code required for correct refactoring
• This work is part of a long-term research project focusing on advanced enterprise refactoring tools, aiming to assist in daily software development on the one hand and in legacy modernization on the other
• The new Extract Computation refactoring for isolating fine slices is a crucial building block in this endeavor
  – It will be used to enhance the automation for complex code-motion refactorings in order to support enterprise transformations such as the move to MVC [WRT08-09]
  – As our prototype matures, it will be possible to evaluate to what extent such enterprise transformations can be automated
• Present and future work
  – Refactoring tools for Cobol in RDz
  – Refactoring tools for Java in Eclipse
  – Fine slicing tool to assist in program comprehension
  – Big refactorings too: composition; meaningful error reporting
References

- Abadi, Ettinger, and Feldman, WRT08: Re-approaching the refactoring rubicon. *Workshop on Refactoring Tools @ OOPSLA08*.
- Abadi, Ettinger, and Feldman, WRT09: Fine slicing for advanced method extraction. *Workshop on Refactoring Tools @ OOPSLA09*.
- Griswold&Notkin, ToSE93: Automated assistance for program restructuring.
- Komondoor&Horwitz, POPL00: Semantics-preserving procedure extraction.
- Komondoor&Horwitz, IWPC03: Effective automatic procedure extraction.
- Lakhotia&Deprez, IST98: Restructuring programs by tucking statements into functions.
- Maruyama, SSR01: Automated method-extraction refactoring by using block-based slicing.
Backup
A Case Study in Enterprise Refactoring [WRT08-09]

- Converted a Java Servlet to use the MVC pattern, using a series of small refactoring steps*
- Inadequate automation for most steps
- Most significant deficit in Extract Method
  - (a) Extract multiple fragments
  - (b) Extract a partial fragment
  - (c) Extract loop with partial body
  - (d) Extract code with conditional exits
- **All 4 cases involve the extraction of fine slices**

* Based on: Alex Chaffee, “Refactoring to Model-View-Controller”
  (http://www.purpletech.com/articles/mvc/refactoring-to-mvc.html)
(a) Extract multiple fragments

User user = getCurrentUser(request);
if (user == null) {
    response.sendRedirect(LOGIN_PAGE_URL);
    return;
}

response.setContentType("text/html");
disableCache(response);

String albumName = request.getParameter("album");

PrintWriter out = response.getWriter();
(b) Extract a partial fragment

```java
out.println(DOCTYPE_HTML);
out.println("<html>");
out.println("<head>");
out.println("<title>Error</title>");
out.println("</head>");
out.print("<body><p class='error'>");
out.print("Could not load album ", albumName + "}");
out.println("</p></body>");
out.println("</html>" costs};
```
out.println("<table border=0>");

int start = page * 20;
int end = start + 20;
end = Math.min(end, 
    album.getPictures().size());

for (int i = start; i < end; i++) {
    Picture picture = album.getPictures(i);
    printPicture(out, picture);
}

out.println("</table>");
int start = page * 20;
int end = start + 20;
end = Math.min(end,
    album.getPictures().size());

Queue<Picture> pictures =
    new LinkedList<Picture>();
for (int i = start; i < end; i++) {
    Picture picture = album.getPicutre(i);
pictures.add(picture);
}

out.println("<table border=0>");
for (int i = start; i < end; i++)
    printPicture(out, pictures.remove());
out.println("</table>");
(d) Extract code with conditional exits

```java
if (album == null) {
    new ErrorPage("Could not load album '",
    + album.getName() + ",").printMessage(out);
    return;
}

//...
if (invalidAlbum(album, out))
    return;
}
//...

boolean invalidAlbum(Album album,
    PrintWriter out) {
    boolean invalid = album == null;
    if (invalid) {
        new ErrorPage("Could not load album \\
            + album.getName() + ")
            .printMessage(out);
    }
    return invalid;
}