Multistaging to Understand: Distilling the Essence of Java Code Examples

Huascar Sanchez
SRI International
Email: huascar.sanchez@sri.com

Jim Whitehead
University of California Santa Cruz
Email: ejw@soe.ucsc.com

***Abstract*—Programmers commonly search the Web to find code examples that can help them solve a specific programming task. While some novice programmers may be willing to spend as much time as needed to understand a found code example, more experienced ones want to spend as little time as possible. They want to get a quick overview of their operation, so they can start working on them immediately. In this paper, we introduce a technique that helps them accomplish that. Our technique automatically reveals interesting segments of code in an easily understood sequence. These segments can be explored non-sequentially; enabling exploratory learning. We discuss the key components of our technique and describe empirical results based on actual code examples on StackOverflow.**

I. Introduction

We define the *essence* of a code example as a set of cohesive chunks of behavior that convey the most important aspects of the example's intended function. The essence of a code example is related to the example's decomposition, and thus, a key factor in overall code understanding. We call the decomposition of a code example into these chunks, *distillation*. Distilling a code example's essence is a process likely to play a significant part in code foraging activities.

Programmers who forage code on the Web have some intuitive notion of this distillation concept, and frequently use it to get an overview of *unfamiliar* code examples' operation [1]. While this act of distilling code examples' essence can be invaluable to programmers, it is still a cumbersome act [2]. This represents a problem for programmers wishing to get a quick overview of code examples' operation. We address this problem in this paper.

A. Accuracy versus Efficiency

When programmers find code examples online, they often go outside the Web browser to experiment with them in an editor. They do that before fully understanding the included code samples [2]. This premature form of reuse might negatively affect programmers in multiple ways [3], such as the loss of the example's context and understanding.

Although there are tools that try to bridge the gap between the Web browser and the editor to re-establish context [4], [5], they all critically rely on humans for making distilling decisions. The rationale is that humans are better and more accurate at making these types of decisions. The downside of relying entirely on humans is that the distillation process is still time-consuming. This clashes with our goal of making the distillation process quick and easy. If we can automate the distillation process, right at the original Web page, and then present the distilled code, then we can help programmers complete their code understanding tasks quickly and accurately.

One way to speed up the distillation process is through multistage code examples [6]. Multistage code examples have their functionality broken into discrete chunks of behavior, called code stages. Each code stage is self-contained and builds upon, and in relation to, preceding code stages. Together, they provide an optional and yet accessible roadmap of the code example. By following this roadmap, their learners can complete their code understanding tasks quickly and accurately.

B. Our approach

We approach the distillation process by automatically multistaging code examples. Specifically, we decompose a code example into a series of discrete chunks of behavior. This decomposition continues to the point where a chunk cannot be further decomposed. Non-Decomposing chunks are called *prime* subsets of behavior. The resulting chunks are self-contained and thus can be accessed non-sequentially.

Following the distilled code suggests a form of code inspection we call *Multistaging to Understand* (MTU). By adopting this form of inspection, programmers are directed to specific units of functionality that might be of interest. Programmers can explore these units in any order; enabling some form of exploratory learning [7]. Similar to other code reading techniques [8], MTU is useful if the code is poorly specified, as it is often the case for online code examples [9]. Unlike these techniques, the identification of the prime subsets of behavior is done automatically, based on the source code's content.

The key problem in content-based multistaging is determining the minimal set of declarations [1] that are required for each code stage to compile properly. This problem is magnified by the ambiguities in code examples [10]. We use an algorithm called *MethodSlicing* for decomposing code examples into an ordered set of code stages. The algorithm doesn't place any limitations on the completeness of code examples, as its input code is *packed* with the code elements needed for its proper operation (see IV-B for details). The only limitation is that the input code must be written in Java.

[1]Declarations bind identifiers to elements in the source code; e.g., variables, methods, and class hierarchy declarations.

Whereas previous work has considered authoring multistage code examples using either direct editing and editable code histories [5], record and replay of code-based tutorials [11], or annotated code tours that highlight important code locations [12], we propose something different. We propose that code stages can be algorithmically discovered by statically analyzing the examples' Java code.

Our multistaging approach is implemented atop the *Vesperin* system [9]. *Vesperin* is a system for curating online Java source code. At a high level, *Vesperin* consists of two main components: a Chrome extension (named *Violette*) for allowing developers to actually modify code examples in their original Web page, and a RESTful service (named *Kiwi*) for managing curation and snippet parsing operations. Together, they provide a mechanism by which developers can examine Java code examples through source code curation.

C. Contributions

The technical contributions of our work are as follows:

1) We introduce two algorithms for distilling Java code examples' essence, called *MethodSlicing*, and *MethodSlicing with Reduction*.
2) We introduce a form of code inspection, called *Multistaging to Understand*.
3) We implement a prototype atop the *Vesperin* system. This prototype implements our technique and assists programmers with their code understanding tasks.
4) We explore our technique's effectiveness experimentally.

The remainder of the paper is organized as follows: Section II describes a *MTU* session. Section III introduces the *Multistaging Problem* and how to solve it. Section IV describes its architecture. Section V presents its evaluation. Section VI summarizes related work. Section VII concludes.

II. Multistaging to Understand

The idea behind MTU is that, as the programmers non-sequentially inspect a few of the generated code stages, their functionality is mentally abstracted, and then combined to understand the intended function of a code example. This section walks the reader through a MTU session (summarized in Figure 5). In the remainder of this section, we will refer to the programmer using our technique simply as Bella.

To facilitate presentation, we make certain assumptions on the used code example; e.g., complete code and compiling. Any deviation from these assumptions will not affect the applicability of our technique, as indicated in Section IV-B.

Figure 1 shows the first step. This step takes as input a query describing Bella's information need. Bella then issues this query to a Web search engine. The Web search engine returns a set of generated results (see Figure 2). At this point, Bella selects the first result. She does that to examine the result's code example [2] (see Figure 3).

Bella uses *Violette* to create a new scratch space for the found code example. This is only possible due to it being

[2]Available at http://stackoverflow.com/q/29802290#29802635.

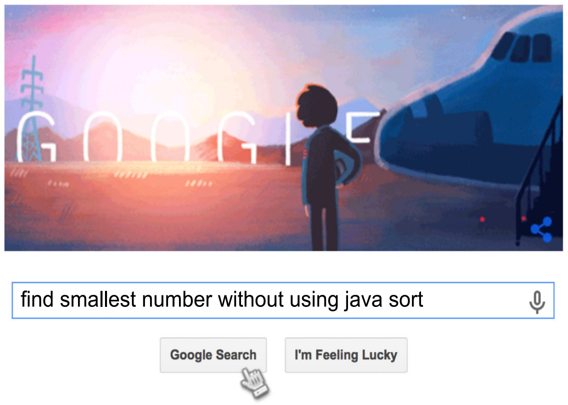
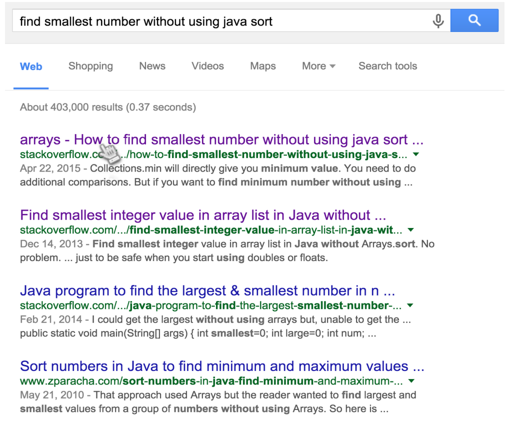


Fig. 1. Search for a code example that finds the smallest number in an array without using sorting.



find smallest number without using java sort

Web Shopping News Videos Maps More Search tools

About 403,000 results (0.37 seconds)

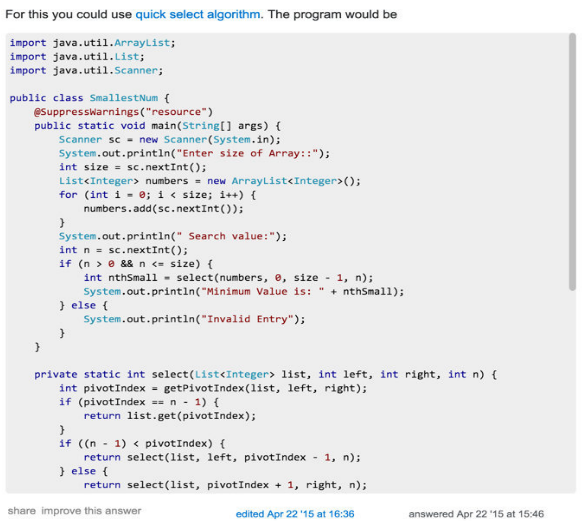
arrays - How to find smallest number without using java sort ...
stackoverflow.com/.../how-to-find-smallest-number-without-using-java-s...
Apr 22, 2015 - Collections.min will directly give you minimum value. You need to do additional comparisons. But if you want to find minimum number without using ...

Find smallest integer value in array list in Java without ...
stackoverflow.com/.../find-smallest-integer-value-in-array-list-in-java-wit...
Dec 14, 2013 - Find smallest integer value in array list in Java without Arrays.sort. No problem. ... just to be safe when you start using doubles or floats.

Java program to find the largest & smallest number in n ...
stackoverflow.com/.../java-program-to-find-the-largest-smallest-number-...
Feb 21, 2014 - I could get the largest without using arrays but, unable to get the ... public static void main(String[] args) { int smallest=0; int large=0; int num; ...

Sort numbers in Java to find minimum and maximum values ...
www.zparacha.com/sort-numbers-in-java-find-minimum-and-maximum-...
May 21, 2010 - That approach used Arrays but the reader wanted to find largest and smallest values from a group of numbers without using Arrays. So here is ...

Fig. 2. Generated results by Web search engine.



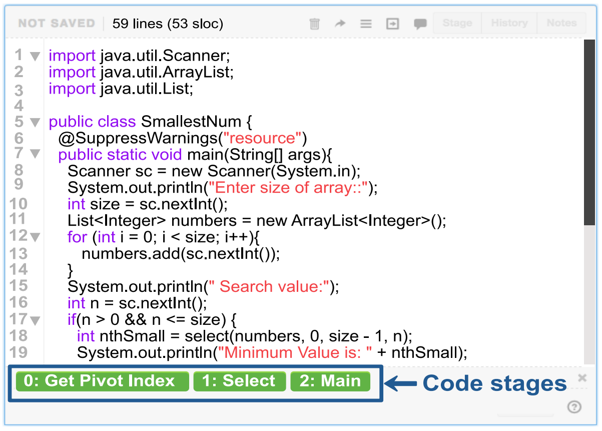
For this you could use quick select algorithm. The program would be

```java
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;

public class SmallestNum {
    @SuppressWarnings("resource")
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.println("Enter size of Array::");
        int size = sc.nextInt();
        List<Integer> numbers = new ArrayList<Integer>();
        for (int i = 0; i < size; i++) {
            numbers.add(sc.nextInt());
        }
        System.out.println(" Search value:");
        int n = sc.nextInt();
        if (n > 0 && n <= size) {
            int nthSmall = select(numbers, 0, size - 1, n);
            System.out.println("Minimum Value is: " + nthSmall);
        } else {
            System.out.println("Invalid Entry");
        }
    }

    private static int select(List<Integer> list, int left, int right, int n) {
        int pivotIndex = getPivotIndex(list, left, right);
        if (pivotIndex == n - 1) {
            return list.get(pivotIndex);
        }
        if ((n - 1) < pivotIndex) {
            return select(list, left, pivotIndex - 1, n);
        } else {
            return select(list, pivotIndex + 1, right, n);
```

share improve this answer edited Apr 22 '15 at 16:36 answered Apr 22 '15 at 15:46

Fig. 3. Smallest number in array code example.

hosted on StackOverflow. Then, she presses the `Stage` button. *Violette* responds to this action by asking *Kiwi* to multistage the code example. *Kiwi* returns 3 generated code stages (illustrated in Figure 4).



```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;

public class SmallestNum {
  @SuppressWarnings("resource")
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter size of array::");
    int size = sc.nextInt();
    List<Integer> numbers = new ArrayList<Integer>();
    for (int i = 0; i < size; i++){
       numbers.add(sc.nextInt());
    }
    System.out.println(" Search value:");
    int n = sc.nextInt();
    if(n > 0 && n <= size) {
      int nthSmall = select(numbers, 0, size - 1, n);
      System.out.println("Minimum Value is: " + nthSmall);
```

Fig. 4. Generated code stages (green buttons).

After reading the labels of the generated code stages, Bella builds a general hypothesis about the nature of the code example. Using the fact that code stages are self-contained and can be accessed in any order, Bella clicks the first code stage that comes to her mind. This is the `Get Pivot Index` code stage (see Figure 5a).

Bella then skims `getPivotIndex`'s method signature and starts refining her general hypothesis about the code example. She hypothesizes that `getPivotIndex`'s function is to get an index between two indices (i.e., left and right).

With this new hypothesis in mind, she inspects the visible code blocks in the code stage (see Figure 5a), opening any hidden code blocks as she goes along. The first hidden block she opens is the one located at line 7 in Figure 5b. The code inside this block is responsible for incrementing the left index as long as this index is less than the pivot value (calculated in line 5). She uses this information to deduce the function of the next hidden code block, located at line 8 in Figure 5b. The elements in this block are responsible for navigating a list of integers from right to left.

After having verified the function of the two code blocks, Bella notices certain similarities between the `getPivotIndex` method and the Quicksort's `Partition` method. She uses this information to guess the function of the next hidden code block, which starts at line 11 in Figure 5c. The function of this hidden code block is to swap elements in a list of integers only if the left index is less than the right index.

At this point, Bella is getting the hang of inspecting code stages. Consequently, she approaches the `Select` code stage in the same way (see Figures 5d, 5e, and 5f).

After having inspected the `Select` code stage, and learned its function, Bella feels she has achieved her desired compression level. As a result, she skips the `Main` code stage and then combines all her gained knowledge to determine the code example's function. The function is to find the kth smallest element in a list using continuous list partitioning and careful recursive calls. She now thinks she can use this example in her own work. This ends the MTU session.

III. The Multistaging Problem

The goal of multistaging code examples is to reveal segments of code in an easily understood sequence. We generalize this problem as follows:

Problem 3.1: **The Multistaging Problem.** Given a code example's abstract syntax tree (AST), with a set of n method declarations $D = D_1 \cup D_2 \cdots \cup D_n$, compute an *ordered* set of code stages $\{ S \mid S \subseteq D \times D \}$, such that each code stage $s \in S$ builds upon, and in relation to, preceding code stages; i.e., $s_i \leq s_j, s_i$ precedes s_j, where $i, j = 1, \ldots, |S|$.

In this problem, the first code stage of a Java code example always lacks a preceding code stage. As such, without loss of generality, we add a special `null` code stage to the set S; called $s_\emptyset$. The preceding code stage of $s_\emptyset$ is itself.

Unlike the work in [5], we consider a code stage as a group of code fragments that captures a prime subset of the behavior found in the Java code example. Whenever a code stage is generated, a *composition* relationship is established between the new code stage and a previous code stage. For example, the `Select` code stage (see Figure 5f) contains a method from the `Get Pivot Index` code stage (see Figure 5a). We generalize these insights using the notion of Code Stages (See Definition 3.1). This definition simplifies the multistaging process, as we describe it later in this section.

Definition 3.1: **Code stages.** Code stages are a set of subsets of behavior found in a code example, such that

- Each subset contains a method or a set of collaborating methods [3].
- Each subset builds upon, and in relation to preceding subsets.
- Code stages enumerate all the subsets with the above properties.

We describe an algorithm for multistaging Java code examples in Section III-A. To facilitate presentation, we consider the same Java code example used in Section II.

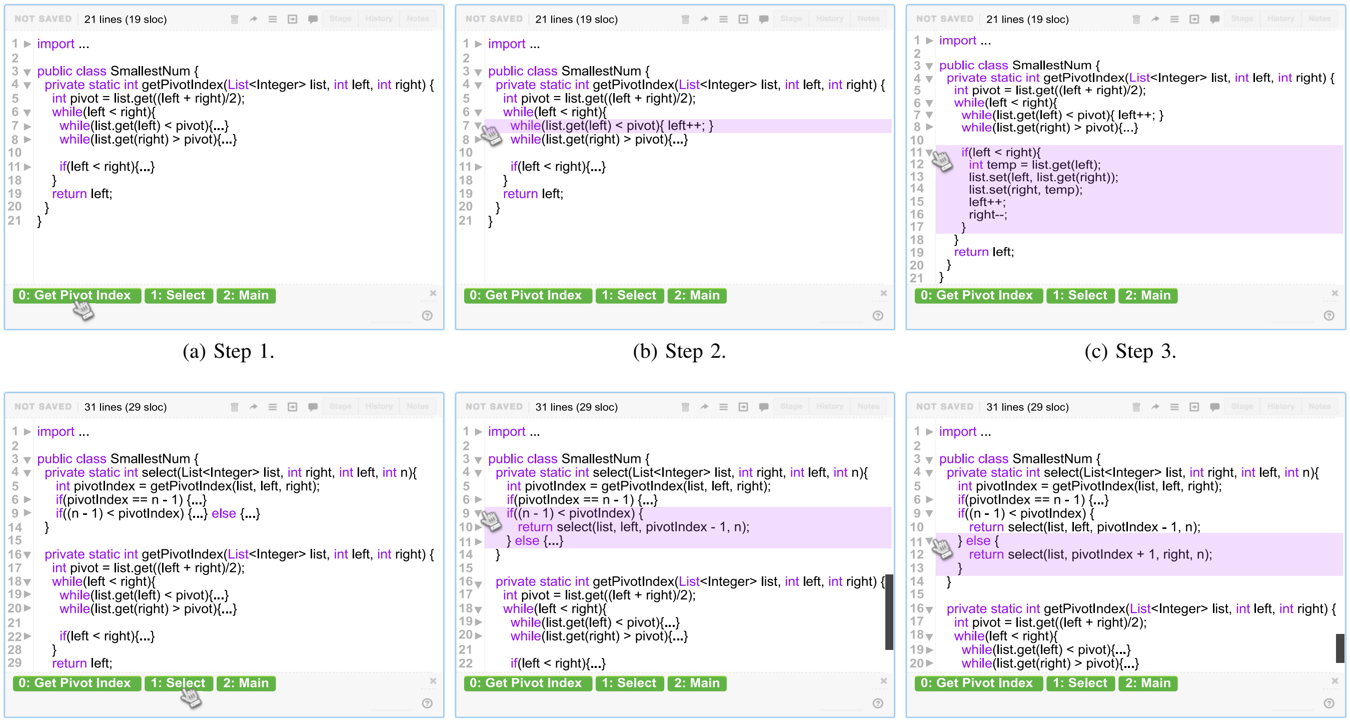
Figure 7 illustrates the computed Code Stages, sorted in ascending order by code stage length. The length of a code stage is determined by its number of lines of code (LOC). In the next section, we show how to compute these code stages.

A. MethodSlicing

In this section, we introduce MethodSlicing. MethodSlicing is an algorithm for solving the Multistaging Problem (described in Problem 3.1). At a high level, MethodSlicing takes an *AST* as input, statically analyzes it, and then slices it into Code Stages. Based on the Definition 3.1, the Code Stages are modeled as a set of tuples containing one or more collaborating methods. Figure 6 describes MethodSlicing.

While the general algorithm may apply to any object-oriented language, some of the details (and used libraries) are tied to the specifics of the Java language. For example, we use

[3]Collaboration is established through method invocations.



(a) Step 1.

```
21 lines (19 sloc)
import ...

public class SmallestNum {
  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){...}
      while(list.get(right) > pivot){...}

      if(left < right){...}
    }
    return left;
  }
}
```

(b) Step 2.

```
21 lines (19 sloc)
import ...

public class SmallestNum {
  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){ left++; }
      while(list.get(right) > pivot){...}

      if(left < right){...}
    }
    return left;
  }
}
```

(c) Step 3.

```
21 lines (19 sloc)
import ...

public class SmallestNum {
  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){ left++; }
      while(list.get(right) > pivot){...}

      if(left < right){
        int temp = list.get(left);
        list.set(left, list.get(right));
        list.set(right, temp);
        left++;
        right--;
      }
    }
    return left;
  }
}
```

(d) Step 4.

```
31 lines (29 sloc)
import ...

public class SmallestNum {
  private static int select(List<Integer> list, int right, int left, int n){
     int pivotIndex = getPivotIndex(list, left, right);
     if(pivotIndex == n - 1) {...}
     if((n - 1) < pivotIndex) {...} else {...}
  }

  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){...}
      while(list.get(right) > pivot){...}

      if(left < right){...}
    }
    return left;
```

(e) Step 5.

```
31 lines (29 sloc)
import ...

public class SmallestNum {
  private static int select(List<Integer> list, int right, int left, int n){
     int pivotIndex = getPivotIndex(list, left, right);
     if(pivotIndex == n - 1) {...}
     if((n - 1) < pivotIndex) {
        return select(list, left, pivotIndex - 1, n);
     } else {...}
  }

  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){...}
      while(list.get(right) > pivot){...}

      if(left < right){...}
```

(f) Step 6.

```
31 lines (29 sloc)
import ...

public class SmallestNum {
  private static int select(List<Integer> list, int right, int left, int n){
     int pivotIndex = getPivotIndex(list, left, right);
     if(pivotIndex == n - 1) {...}
     if((n - 1) < pivotIndex) {
        return select(list, left, pivotIndex - 1, n);
     } else {
        return select(list, pivotIndex + 1, right, n);
     }
  }

  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){...}
      while(list.get(right) > pivot){...}
```

Fig. 5. Multistaging to Understand session. Unfolded code blocks are highlighted.

```
Algorithm MethodSlicing($p$)  // AST p
    $S \leftarrow \emptyset \cup \{s_\emptyset\}$  // Code stages
    for each method $m \in p$ do
        $B \leftarrow \emptyset \cup$ GetAllAstNodeBindings($m$)
        $d \leftarrow \emptyset$  // declarations set
        for each binding $b \in B$ do
            $d[b] \leftarrow p[b]$ declaring node
        end for
        $s \leftarrow$ ReconstructSourceCode($p, d$)
        $S \cup \{s\}$
    end for
    sort $S$ in ascending order
    return sorted $S$
end Algorithm
```

Fig. 6. Pseudocode for `MethodSlicing`.

Vesperin's source code curation utilities to parse, manipulate, and reconstruct Java code.

MethodSlicing uses two subroutines (illustrated in Figure 8, and Figure 9). `GetAllAstNodeBindings` subroutine (Figure 8) collects binding information of *AST* nodes. This subroutine uses *Vesperin*'s traversal utilities to walk the *AST* node representing a method declaration. During this walk, it collects the binding information for variables, fields, parameters, packages, methods, and inner class (or nested) declarations used within that method declaration. The declarations corresponding to these bindings are then obtained for later use in the algorithm.

We model binding information as a set of n tuples made of name-and-ASTNodeType pairs; e.g., *(name, nodetype)*$_1$ ... *(name, nodetype)*$_n$, where n is the number of named entities in a Java code example. For example, a Java package `java.util.Scanner` is modeled as `(Scanner,package)`.

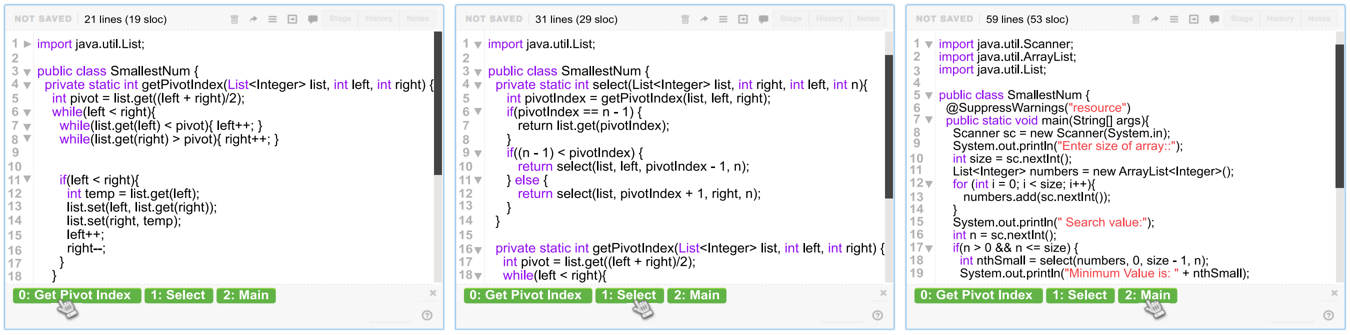
`ReconstructSourceCode` subroutine (Figure 9) uses *Vesperin*'s code transformation utilities to reconstruct a code stage's source code based on the obtained declarations and the *AST*.

MethodSlicing iterates over the methods in an *AST* in the exact order [4] they were written in the code example. For each method, it collects the binding information of its children, as well as the binding information of collaborating methods. It then specifies the content of a code stage (see Figure 6) by getting the *AST* node declarations of each method's local bindings B. For example, `getPivotIndex`'s B is: `{(List,package), (SmallestNum,type), (getPivotIndex,method), (left',parameter), (right',parameter), (list',parameter), (pivot,variable), (temp,variable)`.

The above elements represent the content of `Get Pivot Index` code stage. This code stage includes the `SmallestNum` class declaration, the `getPivotIndex` method declaration, and `getPivotIndex`'s children. See the reconstructed source code of the `Get Pivot Index` code stage in Figure 7.

MethodSlicing's divide and conquer approach for distilling code examples' essence can be beneficial during code comprehension tasks. However, there is one caveat that can hinder

[4]The ordering of the methods in the *AST* is irrelevant to the algorithm, as it always converge to unique Code stages.



```
import java.util.List;

public class SmallestNum {
  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){ left++; }
      while(list.get(right) > pivot){ right++; }


      if(left < right){
        int temp = list.get(left);
        list.set(left, list.get(right));
        list.set(right, temp);
        left++;
        right--;
      }
    }
```

```
import java.util.List;

public class SmallestNum {
  private static int select(List<Integer> list, int right, int left, int n){
    int pivotIndex = getPivotIndex(list, left, right);
    if(pivotIndex == n - 1) {
      return list.get(pivotIndex);
    }
    if((n - 1) < pivotIndex) {
      return select(list, left, pivotIndex - 1, n);
    } else {
      return select(list, pivotIndex + 1, right, n);
    }
  }

  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
```

```
import java.util.Scanner;
import java.util.ArrayList;
import java.util.List;

public class SmallestNum {
  @SuppressWarnings("resource")
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter size of array::");
    int size = sc.nextInt();
    List<Integer> numbers = new ArrayList<Integer>();
    for (int i = 0; i < size; i++){
      numbers.add(sc.nextInt());
    }
    System.out.println(" Search value:");
    int n = sc.nextInt();
    if(n > 0 && n <= size) {
      int nthSmall = select(numbers, 0, size - 1, n);
      System.out.println("Minimum Value is: " + nthSmall);
```

(a) Code stage 1. (b) Code stage 2. (c) Code stage 3.

Fig. 7. One application of `MethodSlicing` against the `SmallestNum` code example.

```
function GETALLASTNODEBINDINGS($p$)  // AST $p$
    $\mathcal{V}, \mathcal{S}, \mathcal{R} \leftarrow \emptyset$
    $\mathcal{W} \leftarrow$ {target node types}
    $\mathcal{S} \cup p.root$
    while $\mathcal{S}$ is not empty do
        $u \leftarrow$ pop $\mathcal{S}$
        continue unless $u \notin \mathcal{V}$
        $\mathcal{V} \cup \{u\}$
        for each child node $w \in u$ do
            $\mathcal{R} \cup$ {binding of $w$} if $w \in \mathcal{W}$
            $\mathcal{S} \cup \{w\}$
        end for
    end while
    return $\mathcal{R}$  // Set of bindings in $p$
end function
```

Fig. 8. `GetAllAstNodeBindings` subroutine.

```
function RECONSTRUCTSOURCECODE($p, d$)
    // delete declaration nodes $\{p \setminus d\}$ from AST $p$
    $p' \leftarrow p \setminus \{p \setminus d\}$
    return source code for $p'$
end function
```

Fig. 9. `ReconstructSourceCode` subroutine.

its effectiveness: produced code stages might consist of long methods [5]. Long methods tend to take more time to understand than small methods [13]. Consequently, code stages with long methods (i.e., large code stages) might be difficult to digest by those programmers wishing to obtain a quick overview of their operation.

To address this issue, we investigated the obstacles programmers faced when inspecting the large code stages produced by MethodSlicing. We discovered that most of the issues were related to the navigation of the code stages' content. One way to deal with these issues is via code folding. The efficiency of code folding on browsing tasks was validated by Cockburn et al. [14].

In line with our findings, we extended MethodSlicing to automatically reduce large code stages (via code folding) whenever possible. Reduction in MethodSlicing shows the code elements that are most informative (i.e., with high usage score) and hides (i.e., folds) the ones that are less informative in each code stage. However, these hidden elements are easily accessible if one chooses to see them. This technique is described in the next section.

[5]Methods with size close to or beyond 15 LOC.

B. *MethodSlicing with Reduction*

Programmers dealing with large code stages are often confronted with the consequent information overload problem. We can reduce this problem by automatically reducing them. The rationale is that reduced code stages can be easily digested by programmers wishing to get a quick overview of their operation.

We make reduction decisions in MethodSlicing entirely based on examples' source code structure. Our approach is consistent with how human abstractors approach inspecting unfamiliar source code [1]. When inspecting an unfamiliar source code, they extract code fragments according to the hierarchical structure of control flow units present in the source code [15], [16]. This structure can be described as a series of interconnected code blocks. Each code block has an associated usage score. We compute the usage score of a code block using Equation 1. The usage score of a code block is representative of the demand of its elements throughout the code example. The usage frequency of each element in a code block is the number of times this element appears in a code stage. As a result, we use the code blocks' usage score to show the blocks with a higher demand and hide those with a lesser demand.

$$UsageScore(B) = \frac{\sum_{elem \in B} UsageFreq(elem)}{TotalChildren(B)} \tag{1}$$

For example, given a nested code block at line 11 in Figure 5c, we first collect its children: temp, list, left, and right. Second, we compute each child's usage frequency: 2, 7, 10, and 9. Lastly, we put it all together and calculate the nested code block's usage score: $(2 + 7 + 10 + 9)/4 = 7$.

We cast the problem of reducing large code stages as an instance of the Precedence Constrained Knapsack Problem or PCKP [17]. This problem is specified herein.

Problem 3.2: **Code Stage Reduction.** Given a set of code blocks $\mathcal{B}$ (with weight w_b and profit p_b per block $b \in \mathcal{B}$), a

Knapsack capacity $\mathcal{W}$, a precedence order $\mathcal{O} \subseteq \mathcal{B} \times \mathcal{B}$, and a set of constraints $\mathcal{C}$, find $\mathcal{H}^*$ such that $\mathcal{H}^* = \mathcal{B} \setminus \mathcal{X}^*$, where w_b = number of lines of code in b, $p_b = UsageScore(b)$, $\mathcal{X}^*$ = arg max $\{\sum_{b \in \mathcal{B}} p_b\}$, and $\mathcal{X}^*$ satisfies the constraints in $\mathcal{C}$. The constraints in $\mathcal{C}$ include: $\sum_{b_j \in \mathcal{B}} w_{b_j} \leq \mathcal{W}$, where $b_i \rightsquigarrow b_j$ (b_i precedes b_j) $\in \mathcal{O}$, and $i, j = 1, \ldots, |\mathcal{B}|$.

Similar to Samphaiboon et al. [17], we solve this problem by using dynamic programming. Our solution generalizes the code stage reduction problem, also taking into account a precedence relation between code blocks in a code stage. We build a Directed Acyclic Graph (DAG) to represent such a relation, where nodes correspond to code blocks in a one–to–one fashion. This relation is expressed as a composition relation between code blocks. For instance, a code block $k-1$ precedes a code block k, if code block $k-1$ contains the code block k. We build this DAG when traversing a code stage's *AST*. Specifically, we visit all the code block nodes in the *AST*, add them to the DAG, and then return this DAG.

In this DAG, each visited code block has a profit (i.e., the usage score of a code block) and a weight [6] (i.e., the number of lines of code in the code block). Our solution's Knapsack has a fixed capacity (i.e., total number of lines of code to be displayed in the reduced code stage). So, given a code stage and a capacity, our solution automatically reduces a code stage. It does it by identifying the location of non essential code blocks; i.e., those code blocks that if added to the solution would exceed the fixed Knapsack's capacity (see $\mathcal{H}^*$ in Problem 3.2). The value of this capacity is tunable. We selected it based on feedback from our user studies.

```
function ReconstructSourceCode($p, d$)
    // delete declaration nodes $\{p \setminus d\}$ from AST $p$
    $p' \leftarrow p \setminus \{p \setminus d\}$
    $DAG_{p'} \leftarrow$ traverse $p'$ and then get built DAG
    $\mathcal{H}^* \leftarrow$ computes $\mathcal{B}_{p'} \setminus \mathcal{X}^*_{p'}$ using $DAG_{p'}$ and a given $\mathcal{W}$
    return source code for $p'$
end function
```

Fig. 10. Updated `ReconstructSourceCode`.

We extend `ReconstructSourceCode` to include the code stage reduction step. Figure 10 sketches this step. Our solution is based on the following recursive formula:

$$\mathcal{X}^*\text{[k,w]} = \begin{cases} \mathcal{X}^*\text{[k - 1, w]} & w_k > w \\ \max(\mathcal{X}^*\text{[k - 1, w]}, & \\ \mathcal{X}^*\text{[k - 1, w - } w_k\text{]} + p_k) & w_k \leq w \wedge k-1 \rightsquigarrow k \end{cases} \quad (2)$$

This recurrence (see Equation 2) determines the set $\mathcal{X}^*$, which has a total weight w, where $w \leq \mathcal{W}$. In the first case, a block k cannot be part of the solution since the total weight will be greater than w. In the second case, a block k can be in the solution, and we choose the case with greater value only if there is an edge between a previously chosen block $k-1$ and the current block k.

[6]Code blocks enclosing other code blocks have their weight calculated and distributed among their enclosing code blocks. For example, if a code block A surrounds two code blocks B and C, then $w_A = w_{A_{original}} - (w_B + w_C)$.

We use the same code example of Section III as input for MethodSlicing with Reduction. The Knapsack capacity is 15 LOC. We illustrate its output in Figure 11. This figure shows smaller and nicely decomposed code stages. In the next section we describe our code example multistager's architecture.

IV. Multistager Architecture

Figure 12 shows the architecture of our code example multistager. This multistager implements the MethodSlicing with Reduction algorithm. This figure shows two execution paths of the multistager, and its relation with *Vesperin* [9] and its two main components: *Violette* and *Kiwi*.

A. Multistaging Requests

Consider the architecture illustrated in Figure 12. At any time, during a code example's inspection, programmers may use this multistager through *Violette*'s interface. Programmers can press the `Stage` button. *Violette* follows this action by issuing a remote call to *Kiwi* requesting the example's Java code to be decomposed into an ordered set of code stages. *Kiwi* reacts to this call by first *packing* the example to generate its *AST*. Then, it multistages it with code stage reduction in mind. Lastly, it ships the generated code stages back to the caller. In a non-error path, *Kiwi*'s reply contains a set of code stages (including $\mathcal{H}^*$). Otherwise, it contains a set of warnings describing a failed multistaging attempt.

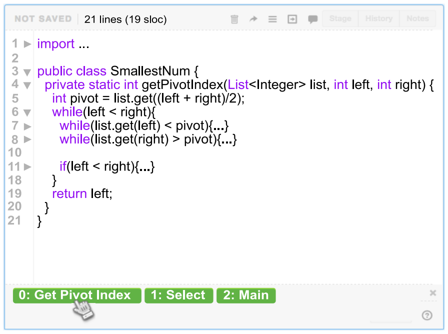
B. Packing Code examples

To correctly multistage Java code examples, we must produce an *AST* representation of their code. Code examples are often partial and non-compiling programs. Consequently, producing an *AST* of a partial program that can be multistaged is difficult. Especially if the local information present in their code is incomplete, and thus insufficient for resolving all their bindings at compile time.

To address this problem and thus guarantee multistaging, the multistager performs a four step process, called *Codepacking*. *Codepacking*'s goal is not about achieving program correctness but approximating program completeness. First, it surrounds a code example with the appropriate body declarations (class, method, or both), if needed. Second, it cross references missing type information of existing ill-typed expressions [18] with a repository of pre-acquired API information and then suggests their well-typed corrections. Third, it uses *Vesperin*'s code transformation utilities to add the suggested corrections to the example's code. Fourth, it produces the desired *AST*.

Briefly, we model APIs in the repository as tuples in $T \times L \times T \times \cdots \times T$, where the first T represents the return type of a public method, the L represents its labels, the second T represents the type of its receiver object [7], and the rest of the elements represent the types of its arguments. Labels are strings of characters that we used to refer to either a particular type or method in the repository. For example, a static Java method `static double ceil(double x)` of `java.lang.Math` is modeled as `(double,` **`(math,`**

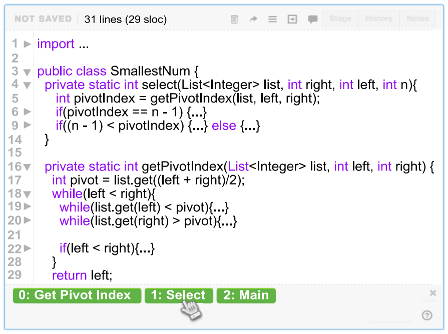
[7] Public static methods' receiver object is prepended with **static:** keyword.



NOT SAVED | 21 lines (19 sloc)

```
import ...

public class SmallestNum {
  private static int getPivotIndex(List<Integer> list, int left, int right) {
    int pivot = list.get((left + right)/2);
    while(left < right){
      while(list.get(left) < pivot){...}
      while(list.get(right) > pivot){...}

      if(left < right){...}
    }
    return left;
  }
}
```

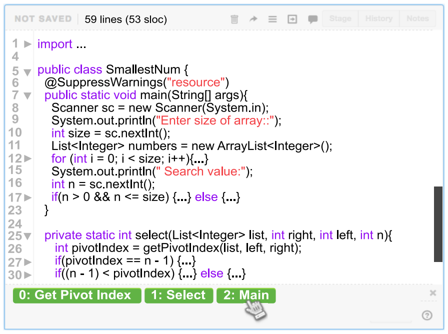
(a) Reduced Code stage 1.



NOT SAVED | 31 lines (29 sloc)

```
import ...

public class SmallestNum {
  private static int select(List<Integer> list, int right, int left, int n){
    int pivotIndex = getPivotIndex(list, left, right);
    if(pivotIndex == n - 1) {...}
    if((n - 1) < pivotIndex) {...} else {...}
  }

  private static int getPivotIndex(List<Integer> list, int left, int right) {
   int pivot = list.get((left + right)/2);
   while(left < right){
     while(list.get(left) < pivot){...}
     while(list.get(right) > pivot){...}

     if(left < right){...}
   }
   return left;
```

(b) Reduced Code stage 2.

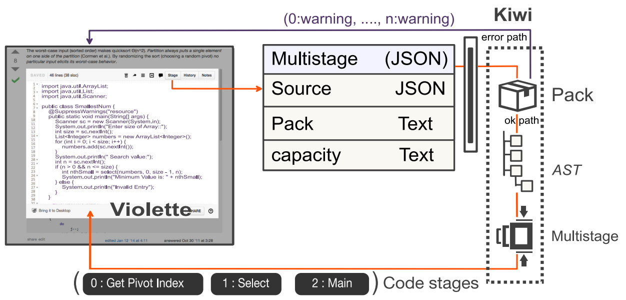


NOT SAVED | 59 lines (53 sloc)

```
import ...

public class SmallestNum {
  @SuppressWarnings("resource")
  public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    System.out.println("Enter size of array::");
    int size = sc.nextInt();
    List<Integer> numbers = new ArrayList<Integer>();
    for (int i = 0; i < size; i++){...}
    System.out.println(" Search value:");
    int n = sc.nextInt();
    if(n > 0 && n <= size) {...} else {...}
  }

  private static int select(List<Integer> list, int right, int left, int n){
    int pivotIndex = getPivotIndex(list, left, right);
    if(pivotIndex == n - 1) {...}
    if((n - 1) < pivotIndex) {...} else {...}
```

(c) Reduced Code stage 3.

Fig. 11. Reduced large code stages. Knapsack capacity is 15. The . . . symbol represents folded areas.



Multistage	(JSON)
Source	JSON
Pack	Text
capacity	Text

Fig. 12. Code example multistager's architecture.

ceil), static:java.lang.Math, (double)). We used information about the code elements in 4 popular APIs on StackOverflow to bootstrap the repository: (1) Java 1.6, (2) Guice, (3) Apache commons, and (4) Guava. This repository contains $6,436$ types, $42,164$ methods, and $751,856$ labels.

In what follows, we describe MTU's experimental evaluation. We describe the used methodology, discuss the results, and then describe how we mitigated threats to the validity.

V. Experimental Evaluation

In this section, we evaluate MTU. We assume that one of the factors that influences comprehension time and accuracy in Java code examples is code size. Code size dictates how long a programmer spends inspecting an example's Java code. Consequently, it's important to test MTU using code examples with varying sizes.

We have two *null* research hypotheses:

$H_{0,accuracy,X,Y}$. *MTU* does not increase the accuracy of answers to questions regarding abstractions X for code examples of size Y.

$H_{0,reviewing-time,Y}$. *MTU* does not reduce the reviewing time of code examples of size Y.

The rejection of $H_{0,accuracy,X,Y}$ and $H_{0,reviewing-time,Y}$ hypotheses would lead to the acceptance of two alternative hypotheses, which state: (1) *MTU* increases comprehension accuracy with regards to the creation of abstraction X for code example of size Y; and (2) *MTU* reduces reviewing time of code examples of size Y.

Our experimental set-up is exploratory. Based on [19], we assume a significance criterion α of 0.10. This indicates a 10% probability of concluding that a group difference exists when there is no actual difference.

A. Experimental Design

12 participants were recruited using Upwork, a popular crowdsourcing service. Subjects had previous programming experience with Java, used code foraging, used StackOverflow, and had limited experience with code inspections.

The subjects were split into two groups (control and treatment) of 6 participants, where each group contained subjects with the same mix of abilities (per their Java experience stated in their professional profiles at Upwork). Each group then focused on one technique, and was aware of the other technique, since the experimental processing was previously explained to them. Specifically, the treatment group applied MTU, while the control group applied the Read to Understand (RTU) technique–a technique that involves reading a given source code to steer understanding.

We used a crossed factorial design with two factors to test MTU. This design is balanced. Table I illustrates the details of this design. We use MTU and RTU as the between-subjects factor. Then, we use the Java code examples' size (short, medium, long) as the within-subjects factor, where (1) the size of a short code example is between 35 and 70 LOC; (2) the size of a medium code example is between 70 and 140 LOC; and (3) the size of a long code example is greater than 140 LOC. We exposed the participants in each group to all code example sizes, and applied the assigned inspection technique during each run.

We have two independent variables and two dependent variables. Our independent variables are the program comprehension technique, and the size of Java code examples (short, medium, long). Our dependent variables are the response accuracy, and the code example reviewing time.

To control extraneous factors, such as systematic performance differences (e.g., Java programming experience) between the treatment and control groups, we formed blocks of participants with similar levels of Java programming experience. Specifically, each group had 2 novice, 2 proficient, and

2 expert Java programmers. This setting assures two things: (1) the overall performance of the two groups is expected to be equal, and (2) each participant in a group has a counterpart with similar abilities in the other group.

TABLE I
CROSSED FACTORIAL DESIGN

Size	MTU	RTU
Short	Group 1 – run 1	Group 2 – run 1
Medium	Group 1 – run 2	Group 2 – run 2
Long	Group 1 – run 3	Group 2 – run 3

B. Experimental Materials

All the materials used for this experiment are available at the experiment's Web page: `huascarsanchez.com/experiments/multistaging`.

1) Code Examples: Java was the language chosen for the code examples, being both the language with which the subjects had the most experience and the only language supported by *Vesperin*. We used 3 code examples:

(1) `stackoverflow.com/q/26818478#26819260`,
(2) `stackoverflow.com/q/14210307#14210519`, and
(3) `stackoverflow.com/q/5317329#5843759`

We sorted a pool of 50,000 question and answer (Q&A) pages from StackOverflow [8] into 3 size categories: short ($\sim$50 LOC), medium ($\sim$135 LOC), and long ($\sim$200 LOC). Then, we randomly selected the examples from each size category.

2) Program Comprehension Questions: Our comprehension questions are open-ended questions and are variants of closed-ended questions proposed by Pennington [15]; one for each program comprehension aspect: (1) Function, (2) Operations, (3) Control Flow, (4) Data Flow, and (5) State. Please refer to the experiment's Web page for access to our program comprehension questions.

Table II presents a list of generic questions for evaluating the different abstractions of Pennington's model (label (c)), and their open-ended variants respectively (label (o)).

TABLE II
CLOSED AND OPEN ENDED QUESTIONS

Abstraction	Program Comprehension Questions
Control Flow	(c) What is the sequence of execution? (o) Describe in pseudo-code an execution sequence.
State	(c) What is the content of a data object at some point of execution? (o) Describe the composition of the data object at some point of execution.
Operations	(c) What does the code compute? (o) Describe the need of data object in a sequence of execution.
Data flow	(c) What does the code compute? (o) Describe when a data object gets updated in an execution sequence.
Function	(c) What is the overall functionality? (o) Describe what functionality is implemented in an execution sequence.

[8]Please refer to the experiment's Web page for access to this Q&A dataset.

3) Response Accuracy Rating Scheme: We use Du Bois's rating scheme [19] to evaluate answers to our open-ended questions. We chose this rating scheme because it could identify objective differences in response accuracy of open-ended questions. We rated the response accuracy of our open-ended questions in four categories:

- *Correct Answer*. Included all pertinent information.
- *Almost Correct Answer*. Missed minor details.
- *Right Idea*. Missed important details.
- *Wrong Answer*. Missed all pertinent information.

There is a clear difference in distance among the above types of answers. Consequently, Du Bois's rating scheme rated a correct answer as 10, an almost correct answer as 8, a right idea as 5, and a wrong answer as 0 respectively. Moreover, the rating scheme makes no distinction among responses in the same category as this could lead to highly subjective differences.

Participants were graded randomly and anonymously in order to minimize bias. Once all the assessments were completed, the results were sorted back into the respected groups.

C. Experimental Procedure

We evaluated our technique based on the program comprehension aspects mentioned in Section V-B2.

Participants were introduced to our experimental procedure at the start of the experiment. We also gave them an overview of the goals and guidelines for their assigned comprehension technique. We asked them to spend their time wisely, stay only at the code example's Q&A page, and stick to the assigned task. Please refer to the experiment's Web page for additional information about goals and guidelines.

The experiment comprises three program comprehension tasks. Each task was divided into two parts: A multistaging/code reading part and an answering comprehension questions part. After the introduction and overview, we asked the subjects to start the tasks.

D. Discussion of Results

We discuss in this section the results of our experiment. Specifically, the group differences with respect to our two research hypotheses.

1) $H_{0,accuracy,X,Y}$, Response Accuracy: Contained within Table III are the results of our analysis concerning the response accuracy for both groups. We use "*number* – *number* p-value=*number*" as the format to represent the results in each cell, where the first number represents the average response accuracy of the MTU group, the second number the average response accuracy of the RTU group, and the third number the p-value for the one-sided paired t-test. Since our *null* hypotheses are directional, we use these t-tests to verify whether the MTU group shows higher accuracy in their answers to our comprehension questions than the RTU group.

Also illustrated in Table III are the rejection of our *null* hypothesis (H_0 columns). R stands for Rejected ($p < .10$), and AR stands for Almost Rejected ($.10 < p < .12$).

TABLE III
AVERAGE RESPONSE ACCURACY

	Short	H_0	Medium	H_0	Long	H_0
Function	6.83 - 3.33 p=0.0037	R	7.17 - 3.83 p=0.0509	R	7.67 - 5.00 p=0.0534	R
Control Flow	8.50 - 6.83 p=0.0525	R	7.17 - 4.33 p=0.1984		8.17 - 4.33 p=0.0204	R
Data Flow	8.67 - 6.17 p=0.0462	R	5.33 - 3.00 p=0.2308		8.50 - 6.00 p=0.1199	AR
State	8.67 - 7.00 p=0.0873	R	7.67 - 5.67 p=0.1594		9.00 - 6.50 p=0.0971	R
Operations	7.33 - 3.33 p=0.0595	R	7.83 - 4.83 p=0.0609	R	6.50 - 3.00 p=0.0549	R

Table III shows notable differences in response accuracy averages between the groups, completely favoring the MTU group. Since a critical performance component of our tool is multistaging, the results also imply that the quality of the produced code stages is equivalent (or better) to what humans would produce. However, we noticed that some of the $p - values$ were still high for the medium code example. Information gathered from subjects showed there were multiple wrong answers in both groups. Moreover, it showed there was a higher frequency of wrong answers in RTU group than in the MTU group. This higher frequency caused a significant variation, resulting in these high *p-values*.

We investigated the obstacles subjects faced when inspecting the medium code example. By interviewing the subjects, we discovered that most of the experienced issues were related to deducing the intent of a few delocalized methods in the code example [20]. The reason why this appears in the medium code example is likely by chance. Delocalization appears when a particular goal is implemented by lines of code that appear in spatially disparate areas of the program. It has been shown by [20] that the likelihood of a programmer correctly recognizing a plan or intention in a program decreases as the lines of code that realize it are spread out or delocalized in the program. Reading code examples with delocalized plans or intentions can be difficult to understand as it is time consuming to find all the parts of the plan or intention and then figure out what they do. Consequently, understanding is attempted based on purely local information, resulting in confusion [21].

2) $H_{0,reviewing-time,Y}$, *Reviewing Time:* Differences in reviewing-time between both groups are reported in Table IV. We use the format mentioned in Section V-D1 to represent the results in each cell. We also use t-tests to verify whether the MTU group shows shorter reviewing times than the RTU group.

TABLE IV
AVERAGE REVIEWING TIME

	Short	H_0	Medium	H_0	Long	H_0
Reviewing time(sec)	475 - 745 p=0.0995	R	655 - 1022 p=0.0446	R	465 - 912 p=0.0284	R

Table IV shows shorter reviewing times (close to 50% average reduction) by the MTU group, irrespective of delocalization. This table also includes the rejection of our *null* hypothesis (H_0 columns).

We gathered reviewing times from two sources. We used Upwork's time tracker to time the RTU group and *Violette*'s time tracker to time the MTU group.

E. Threats to Validity

Threats to internal validity typically include the presence of confounding factors that comprise the integrity of the experimental results. We made every effort to minimize these factors, but possibilities in this case can occur:

- Selection effects. These can occur through variations in the natural performance of individual subjects. Subjects were split into two groups of equal abilities (Java experience) in an attempt to minimize these effects.
- Maturation effects. These can occur when subjects react differently as time passes. Tasks were similar and were sorted by code example' size in ascending order. Since positive (e.g., learning) and negative (e.g., fatigue) effects could not be ruled out, we introduced the tasks at the beginning of the experiment to try to counteract these effects.
- Loss of enthusiasm. Since subjects were involved in the experiment for a total of two hours, it is possible that subjects found this action repetitive, and thus their interest dropped towards the end. We informed subjects of the context of our research in advance in an attempt to minimize these effects.

Threats to external validity limit the ability to generalize any results from an experiment to a wider population. Our efforts to minimize threats to external validity were minimal, as the nature of our experiment was exploratory. Consequently, possibilities of threats to external validity include:

- The subjects of the experiment (professional programmers recruited at Upwork) may not be representative of the general software engineering population.
- The Java code examples may not be representative (in complexity or length) of the general Java code examples found on StackOverflow. Interested readers can access these examples online at `stackoverflow.com/questions/tagged/java`

VI. RELATED WORK

Our work builds on two primary areas of prior work; tools for using code examples, and tools for inspecting them.

A. Using Code Examples

Tools for authoring multistage code examples typically assume that a complete and correct set of code stages is not available [5]. They rely on humans to massage a source code (via direct editing and editable histories) and then turn it into a multistage code example. Our work is quite different. We assume that code stages are available and they can be extracted based on the examples' existing source code. We algorithmically generate all the necessary code stages to turn an existing Java code example into a multistage one; all with minimal human intervention. In addition, our technique does not require a complete program to work. It can handle partial and non-compiling programs.

Like our work, *JTourBus* provides a mechanism for incremental navigation of important source code locations [12]. It leads programmers directly to relevant details about the source code, but does not offer an automatic way for identifying these prime locations. In contrast, our *MethodSlicing with Reduction* technique focuses on carefully slicing the example into a series of cohesive chunks of functionality, reducing long chunks whenever possible.

B. Inspecting Java Programs

MTU shares similarities with code reading by stepwise abstraction [8]. Code reading by stepwise abstraction calls for inspectors to identify prime subprograms in the software, determine their function and use them to determine a function for the entire program. The effectiveness and efficiency of this technique for structured programs was validated by [22], and validated for object-oriented programs by [23]. In contrast, we focus on algorithmically generating all the prime subsets of behavior in advance for the programmers. We can generate these prime subsets based entirely on the source code's content.

Another similar approach involves using static program slicing approaches to aid with code inspection. For example, *CodeSurfer* [24] is advertised as a great companion for code inspections. Tools of this sort tend to be inherently conservative. Consequently, they tend to produce large slices that are often too large to be of practical use. In contrast, we reduce this information-overloading problem by automatically reducing large generated slices (i.e., large code stages). A reduced code stage shows its most informative code elements (i.e., code elements with high usage score) and hides (i.e., folds) its less informative ones.

VII. Conclusions

In this paper we present a practical and automated approach for distilling the essence of Java code examples. We show a motivating example and explain how our technique distills its essence. This approach is based on the formulation of the *Multistaging Problem*. Our experimental results suggests that our technique is a valuable tool for understanding code examples where most of the code is non-localized, but has only minor benefits when code is partially or fully delocalized. The technique provides consistent speed improvements, irrespective of delocalization.

References

[1] C. L. Corritore and S. Wiedenbeck, "An exploratory study of program comprehension strategies of procedural and object-oriented programmers," *Int. J. Hum.-Comput. Stud.*, vol. 54, no. 1, pp. 1–23, 2001.

[2] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2009, pp. 1589–1598.

[3] C. Parnin and C. Görg, "Building Usage Contexts During Program Comprehension," in *Proceedings of the 14th International Conference on Program Comprehension*, ser. ICPC '06, 2006, pp. 13–22.

[4] B. Hartmann, M. Dhillon, and M. K. Chan, "HyperSource: Bridging the Gap Between Source and Code-Related Web Sites," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, 2011, pp. 2207–2210.

[5] S. Ginosar, D. Pombo, L. Fernando, M. Agrawala, and B. Hartmann, "Authoring Multi-stage Code Examples with Editable Code Histories," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, 2013, pp. 485–494.

[6] A. Robins, J. Rountree, and N. Rountree, "Learning and Teaching Programming: A Review and Discussion," *Computer Science Education*, vol. 13, no. 2, pp. 137–172, 2003.

[7] J. M. Carroll, *The Nurnberg Funnel: Designing Minimalist Instruction for Practical Computer Skill*. MIT Press Cambridge, MA, 1990.

[8] R. C. Linger, B. I. Witt, and H. D. Mills, *Structured Programming; Theory and Practice the Systems Programming Series*. Boston, MA: Addison-Wesley Longman Publishing Co., Inc., 1979.

[9] H. Sanchez and J. Whitehead, "Source Code Curation on Stackoverflow: The Vesperin System," in *Proceedings of the 37th International Conference on Software Engineering*, ser. ICSE '15, 2015, pp. 661–664.

[10] B. Dagenais and M. P. Robillard, "Recovering Traceability Links between an API and its Learning Resources," in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE '13, 2012, pp. 47–57.

[11] C. Kojouharov, A. Solodovnik, and G. Naumovich, "JTutor: An Eclipse Plug-in Suite for Creation and Replay of Code-based Tutorials," in *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology eXchange*, ser. Eclipse '04, New York, NY, 2004, pp. 27–31.

[12] C. Oezbek and L. Prechelt, "JTourBus: Simplifying Program Understanding by Documentation that Provides Tours Through the Source Code," in *Proceedings of the 23rd IEEE International Conference on Software Maintenance*, ser. ICSM 2007, 2007, pp. 64–73.

[13] M. Mäntylä, J. Vanhanen, and C. Lassenius, "A Taxonomy and an Initial Empirical Study of Bad Smells in Code," in *Proceedings of the International Conference on Software Maintenance*, ser. ICSM '03, 2003, pp. 381–384.

[14] A. Cockburn and M. Smith, "Hidden Messages: Evaluating the Efficiency of Code Elision in Program Navigation," *Interacting with Computers*, vol. 15, no. 3, pp. 387–407, 2003.

[15] N. Pennington, "Comprehension Strategies in Programming," in *Empirical Studies of Programmers: Second Workshop*, G. M. Olson, S. Sheppard, and E. Soloway, Eds. Norwood, NJ: Ablex Publishing Corp., 1987, pp. 100–113.

[16] F. Détienne and F. Bott, *Software Design–Cognitive Aspect*. Springer Science & Business Media, 2002.

[17] Samphaiboon, Natthawut and Yamada, Y, "Heuristic and Exact Algorithms for the Precedence-Constrained Knapsack Problem," *Journal of Optimization Theory and Applications*, vol. 105, no. 3, pp. 659–676, 2000.

[18] M. Lee, B.-Z. Barta, and P. Juliff, *Software Quality and Productivity: Theory, Practice, Education and Training*. Springer, 2013.

[19] B. Du Bois, S. Demeyer, and J. Verelst, "Does the "Refactor to Understand" Reverse Engineering Pattern Improve Program Comprehension?" in *Proceedings of the 9th European Conference on Software Maintenance and Reengineering*, ser. CSMR 2005, 2005, pp. 334–343.

[20] S. Letovsky and E. Soloway, "Delocalized Plans and Program Comprehension," *IEEE Software*, vol. 3, no. 3, p. 41, 1986.

[21] C. A. Welty, "An Integrated Representation for Software Development and Discovery," Ph.D. dissertation, PhD thesis, Rensselaer Polytechnic Institute, Department of Computer Science, 1995.

[22] V. R. Basili and R. W. Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Trans. Software Eng.*, no. 12, pp. 1278–1296, 1987.

[23] A. Dunsmore, M. Roper, and M. Wood, "Practical Code Inspection Techniques for Object-Oriented Systems: An Experimental Comparison," *IEEE Software*, no. 4, pp. 21–29, 2003.

[24] P. Anderson and T. Teitelbaum, "Software Inspection using CodeSurfer," in *Proceedings of the 1st Workshop on Inspection in Software Engineering*, ser. WISE 2001, Jul. 2001, pp. 4–11.