A Catalog of While Loop Specification Patterns

Aditi Barua and Yoonsik Cheon

TR #14-65
September 2014

Keywords: functional program verification, intended functions, program specification, specification pattern, while statement.

1998 CR Categories: D.2.1 [*Software Engineering*] Requirements/Specifications—languages; D.2.4 [*Software Engineering*] Software/Program Verification—correctness proofs, formal methods; D.3.3 [*Programming Languages*] Language Constructs and Features—control structures; F.3.1 [*Logics and Meanings of Programs*] Specifying and Verifying and Reasoning about Programs—logics of programs, specification techniques.

Department of Computer Science
The University of Texas at El Paso
500 West University Avenue
El Paso, Texas 79968-0518, U.S.A

TABLE OF CONTENTS

INTRODUCTION

This document provides a catalog of while loop patterns along with their skeletal specifications. The specifications are written in a functional form known as *intended functions* [3] [6]. The catalog can be used to derive specifications of while loops by first matching the loops to the cataloged patterns and then instantiating the skeletal specifications of the matched patterns [1]. Once their specifications are formulated and written, the correctness of while loops can be proved rigorously or formally using the *functional program verification technique* in which a program is viewed as a mathematical function from one program state to another [1] [5] [6].

This document describes seven different patterns to capture the most commonly used while loops, and some of the patterns are specializations or sub-patterns of other more general ones. The documented patterns are language-neutral in that they can be applied to a wide range of programming languages, from imperative, procedural languages to object-oriented languages. For example, the patterns can be matched to while loops that iterate over different implementations of index-based collections like arrays, strings, and sequences, as well as iterator-based collections like linked list and pointer or reference-based collection data structures commonly found in programming languages such as C, C++, and Java.

Below we first explain briefly functional program verification because its notation is used to document our patterns. We next describe the conceptual framework used to identify and classify different uses of while loops and the corresponding patterns. The framework can also be used to systematically analyze while loops to find matching patterns. We then describe the format that we use to document our patterns. Finally we suggest a general process for applying the patterns.

Functional program verification

The patterns in this document are specified using a notation for functional program verification. In functional program verification, a program is viewed as a mathematical function from one program state to another [1] [5] [6]. In essence, functional program verification involves calculating the function computed by code, called a *code function*, and comparing it with the intention of the code written also as a function, called an *intended function* [3] [6]. For the verification, each section of code is documented with its intended function (see below for an example of annotated code). A *concurrent assignment notation* of the form [x_1, x_2, ..., x_n := e_1, e_2, ..., e_n] is used to express these functions by only stating changes that happen. It states that x_i's new value is e_i, evaluated concurrently in the initial state—the state just before executing the code; the value of a state variable that doesn't appear in the left-hand side remains the same. For example, [x, y := y, x] is a function that swaps the values of two variables x and y. There is a formal annotation language for writing intended functions [3], but in this document we will write intended functions semi-formally but rigorously (see below).

```
// [r : = ∑i=o…a.length-1(a[i] > 0 ? 1 : 0)]
   // [r, i := 0, 0]
   r = 0;
   int i = 0;
   // [r, i := r + ∑j=i…a.length-1(a[j] > 0 ? 1 : 0), anything]
   while (i < a.length) {
      // [r , i := a[i] > 0 ? r + 1 : r, i + 1]
         // [r := a[i] > 0 ? r + 1 : r]
            if (a[i] > k)
               // [r := r + 1]
               r++;
         // [i := i + 1]
         i++;
   }
```

The above code snippet calculates the number of positive values contained in an array *a*, and its intended functions are written semi-formally by using the expression syntax of programming languages like Java (e.g., array indexing and conditional expression E_1 ? E_2: E_3) and mathematical symbols such as $\sum$. The keyword **anything** in concurrent assignments indicates that one doesn't care about its value; this is frequently the case for a local variable including a loop variable.

Framework for analyzing while loops

A while loop can be analyzed by considering four different dimensions: (a) how it acquires the values to be manipulated in the loop body, (b) what operation or manipulation it performs on the acquired values, (c) where the result of the manipulation is stored, and (d) when it terminates its iterations [1]. These dimensions along with some possible values are shown in Figure 1.

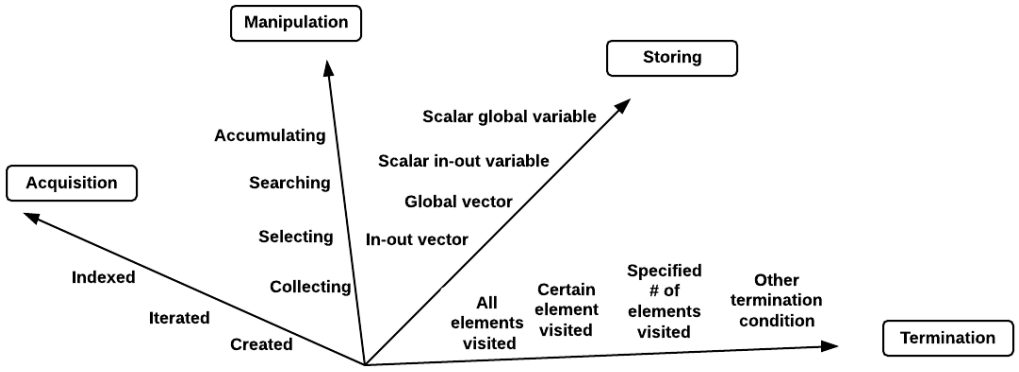


Figure 1: Analysis of while loops

One common use of while loops is to iterate over a sequence of values. The *acquisition* dimension tells how the loop acquires the next value of the sequence to iterate over. The sequence is often stored explicitly, and its elements can be accessed by using an index or an iterator. It is also possible to create the elements on-the-fly without storing them explicitly. The *manipulation* dimension determines the functionality of a loop by telling how the acquired values are manipulated or what operations are performed on them. Some common types of manipulations are accumulating, searching, selecting, and collecting [1]. The *storing* dimension tells where the results are stored. There are many possibilities here, including storing the results to variables different from the sequence being iterated over or mutating the input sequence. For accumulation and searching the results are typically stored in scalar variables whereas for selecting and collecting they are stored in vector or collection variables. The *termination* dimension specifies the termination condition of a loop—a condition that must be true for the loop to terminate its iterations. As in the manipulation dimension, a loop termination condition can differ in many ways, e.g., completing iterations over all elements, iterating over certain elements, completing a certain number of iterations, and satisfying other termination conditions.

We used these four dimensions to analyze many different while loops to come up with the patterns documented in this catalog. These dimensions are also recommended for a systematic analysis of a while loop to find a matching pattern for it.

Pattern documentation

Following the documentation of design patterns [4], each pattern in this catalog is documented by addressing the following major points.

Name: gives a short name for the pattern. A pattern name usually consists of only one or two words.

Purpose: describes the main purpose of the pattern, including the kind of while loops that can be matched to the pattern.

Description: provides more detailed information about the pattern including its skeletal intended function. For example, it provides descriptions of main elements of skeletal intended function such as result variables and the collection being iterated over by the loop.

Structure: explains in detail the structure of the pattern. Each pattern consists of two structural elements: a skeletal intended function (f_1) from which an actual intended function of a matching while loop can be derived and an intended function for the loop body (f_2).

```
[f1]
while (C) {
   [f2]
   …
}
```

The intended function f_1 capture the behavior of the whole loop in terms of f_2 that specifies the behavior of the loop body only. As shown, the loop body is not given in skeletal code but is abstracted to an intended function so that any code segment that correctly implements the intended function can be matched to the pattern.

Example: shows sample code along with intended functions that can be matched to the described pattern.

Application: suggests a general process for applying the described pattern. It also shows a sample application of the pattern to illustrate in a step-by-step fashion how the pattern can be used.

Variations and related patterns: lists variations possible for the described pattern. Some of the variations are named and catalogued as separate patterns, especially when they occur commonly.

Pattern application

We suggest the following general process consisting of four steps for applying the catalogued patterns.

Step 1: Formulate the intended function of the loop body. The first step is to formulate and specify the behavior of the loop body because patterns in this document are specified in terms of the intended function of the loop body, not its code structure. If the code of the loop body doesn't contain any nested loops, its code function can be systematically calculated using a technique like *trace tables* [6]. If the loop body contains another loop, however, the intended function of the nested loop may be found first by applying one of the patterns in this catalog. The intended function or code function of the loop body should document the side effect of the code, i.e., state changes caused by a single iteration of the loop.

Step 2: Find a matching pattern. The next step is to match the loop to one of the patterns documented in this catalog. For this we suggest one to analyze the loop along the four dimensions described in the introduction of this document: (a) how it acquires the values to be manipulated in the loop body, (b) what operation or manipulation it performs on the acquired values, (c) where the result of the manipulation is stored, and (d) when it terminates the iteration [1]. It is likely that most of the analysis, especially acquisition, manipulation, and storing are already performed and documented in the intended function of the loop body. The intended function of the loop body will have the following general form:

$$[s_1, s_2, \ldots, s_n := M_1(e, s_1), M_2(e, s_2), \ldots, M_n(e, s_n)]$$

where s_i is a state variable whose value may be changed in the loop body, e is the current element of the collection being iterated, M_i is a manipulation function defining the new value of s_i possibly in terms of its current value and the current element of the collection. The state variable s_i can be either a result variable or an iterator, and the current element e is typically given in terms of an iterator. To find a match pattern, one can compare the intended function of the loop body of the code with that of the pattern.

Step 3: Unify intended functions of the code and the matching pattern. Once a matching pattern is found, the next step is to define a mapping or correspondence between variables, symbols, and expressions appearing in the intended functions of the loop body of the code and the matched pattern. This mapping will allow one to derive an intended function of the code from the skeletal intended function given by the pattern (see Step 4 below).

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to derive an intended function of the code by instantiating the skeletal intended function given in the pattern. For this, one needs to replace variables, symbols, and expressions appearing in the skeletal intended function with the actual ones of the code using the binding defined in Step 3 above.

ACCUMULATING

1. Purpose

This pattern provides a skeleton intended function for those while loops that iterate over a collection of values and combine some of the values to a single value, e.g., a while loop that adds all positive numbers of an array.

2. Description

One common use of a while loop is to combine certain elements of a collection to a single value by applying various accumulation operators such as addition, multiplication, and concatenation. The Accumulating pattern captures this use of while loops. The type of the result, i.e., the accumulated value is often the same as that of the elements of the collection being iterated over. The intended function of the loop is defined by referring to the accumulated value, the criterion for selecting elements, and the iterator used to access the elements of the collection.

3. Structure

This pattern has the following code structure, where the intended function f_1 capture the behavior of the whole loop and f_2 specifies the behavior of the loop body only.

```
[f1]
while (C) {
   [f2]
   …
}
```

As shown, the loop body is not given in skeletal code but is abstracted to an intended function so that any code segment that implements the intended function correctly can be matched to the pattern. Below we first explain the intended function of the loop body (f_2) and then describe the intended function of the whole loop (f_1) which is written in terms of the components of f_2.

3.1. The loop

The while loop for the Accumulating pattern has the structure shown below. We use a symbol *s* to denote the collection whose elements are to be accumulated. Since its elements are accessed in a certain order in a loop, it can be logically viewed as a sequence, and thus its elements can be denoted by their positions in the sequence. For this we introduce an abstract variable *i*—an abstraction of the iterator to access the elements of the collection—and we use a notation *s*@*i* to denote the *i*-th element of the collection *s*.

```
while (C) {
  [r, i  := P(s@i) ? (r ♢ s@i) : r, E(i)]
  ...
}
```

As specified by the intended function, the loop body may change the values of two state variables, *r* and *i*. The variable *r* stores the result, i.e., the accumulated value, and as explained before *i* is an abstraction of the iterator to access the elements of the collection *s*. The new value of *r* is defined by using a conditional expression of the form E_1 ? E_2 : E_3, denoting either E_2 or E_3 depending on the value of a Boolean expression E_1. The following symbols and notation are used to define the values of *r* and *i*.

P(*x*): a predicate defined on the elements of the sequence *s*. It specifies the criterion for selecting the elements to be accumulated; for an element *x* of the sequence *s*, it tells whether *x* should be accumulated or not. It's a function of signature T → Boolean, where T is the element type of the collection *s*.

$\Diamond$**:** a binary operator of signature T × T → T, where T is the element type of the collection *s*. It's the accumulation operator used to combine elements of the sequence *s*; common examples are addition, multiplication, and string concatenation.

E(*i*) **:** an expression written in terms of the abstract variable *i*. It represents an advancement of the iterator *i* to the next or another element. Examples include *i* + 1 for an index-based collection such as arrays, and *i.next*() for an iterator-based collection.

The new value of *r* is its old value with the current element of *s* (i.e., *s*@*i*) accumulated using the accumulation operator $\Diamond$ if the current element (*s*@*i*) satisfies the selection criterion (*P*(*s*@*i*)); otherwise, it's its old value. The new value of *i* is *E*(*i*) that represents an advancement of the iterator *i* to the next element.

3.2. The intended function

The intended function of the whole loop is defined below. In the intended function, the keyword **anything** indicates that one doesn't care about the final value of the iterator *i*; in general, this is the case for the iterator for it will be a local or incidental variable used only for accessing the elements of a collection.

[r, i := $\overline{\Diamond}$(r, s@i..)**, anything**]
while (*C*) {
  [r, i := *P*(s@i) ? (r $\Diamond$ s@i) : r, *E*(i)]
}

The final value of *r* is defined using the following symbols and notations.

s@*i*..: a subsequence of *s* starting at *i* and selected using the advancement expression *E*(*i*). It is a sequence consisting of elements *s*@*i*, s@*E*(*i*), s@*E*(*E*(*i*)), s@*E*(*E*(*E*(*i*))), etc, and represents the elements of *s* that are iterated over by the loop. The last element of *s*@*i*.. is determined by the loop termination condition *C*, and if the condition fails at the first iteration, *s*@*i*.. is an empty sequence. The sequence is defined recursively as follows.

s@i..	≡ s@i ⊦ s@*E*(i)..	if *i* denotes a valid index or position of *s*
	⟨⟩	otherwise

where ⟨⟩ denotes an empty sequence and ⊦ denotes concatenation of an element to a sequence.

$\overline{\Diamond}$: a promotion of the $\Diamond$ binary operator to a sequence. It is a function of signature T × Seq(T) → T, where T is the element type of the sequence s and Seq(T) is a sequence of elements of type T. It accumulates the elements of the given sequence to the given element using the $\Diamond$ accumulation operator, and can be defined recursively as follows.

$\overline{\Diamond}$(e, ⟨⟩) ≡ e
$\overline{\Diamond}$(e, h ⊦ t) ≡ *P*(h) ? $\overline{\Diamond}$ (e $\Diamond$ h, t) : $\overline{\Diamond}$ (e, t)

where ⟨⟩ denotes an empty sequence and ⊦ denotes concatenation of an element to a sequence. When the given sequence is empty, it returns the given element. When the given sequence is not empty and the first element (*h*) satisfies the selection criterion (*P*), the first element is accumulated to the given element (*e*) using the accumulation operator ($\Diamond$) and the function is recursively applied to the rest of the sequence. If the first element fails the selection criterion, it is ignored and the function is recursively applied to the rest of the sequence.

The intended function states that the final value of *r* is $\overline{\Diamond}$(*r*, *s*@*i*..), an accumulation of elements of *s*—only those elements of *s* at positions *i*, *E*(*i*), *E*(*E*(*i*)), and so on that satisfy the selection criterion *P*—to the initial value *r* using the $\Diamond$ operator.

4. Example Code

The while loop below adds all positive elements of an array *a* starting at index *i* and stores the result to *sum*. Later we will show how one can derive the intended function of the loop by using the Accumulating pattern.

```
// [sum, i := sum + ∑ j=i..a.length-1 (a[j] > 0 ? a[j] : 0) , anything]
while (i < a.length) {
  // [sum, i  := a[i] > 0 ? a[i] : 0, i + 1]
  if (a[i] > 0) {
     sum = sum + a[i];
  }
  i++;
}
```

5. Application

5.1. Suggested Process

We suggest the following general process consisting of four steps for applying the pattern.

Step 1: Formulate the intended function of the loop body. The first step is to formulate and specify the behavior of the loop body because the pattern is specified in terms of the intended function of the loop body, not its code structure. If the code of the loop body doesn't contain any nested loops, its code function can be systematically calculated using a technique like *trace tables* [6]. If the loop body contains another loop, however, the intended function of the nested loop may be found first by applying one of the patterns in this catalog. The intended function or code function of the loop body should document the side effect of the code, i.e., state changes caused by a single iteration of the loop.

Step 2: Find a matching pattern. The next step is to match the loop to one of the patterns documented in this catalog. For this we suggest one to analyze the loop along the four dimensions described in the introduction of this document: (a) how it acquires the values to be manipulated in the loop body, (b) what operation or manipulation it performs on the acquired values, (c) where the result of the manipulation is stored, and (d) when it terminates the iteration [1]. It is likely that most of the analysis, especially acquisition, manipulation, and storing are already performed and documented in the intended function of the loop body. The intended function of the loop body will have the following general form:

$$[s_1, s_2, \ldots, s_n := M_1(e, s_1), M_2(e, s_2), \ldots, M_n(e, s_n)]$$

where s_i is a state variable whose value may be changed in the loop body, e is the current element of the collection being iterated, M_i is a manipulation function defining the new value of s_i possibly in terms of its current value and the current element of the collection. The state variable s_i can be either a result variable or an iterator, and the current element e is typically given in terms of an iterator. If the intended function has the following specific form, the loop can be matched to the Accumulating pattern.

$$[r, i := P(e) \ ? \ F(e, r) : r, E(i)]$$

where r is a result variable whose type is the same as that of the element, P is a predicate defined on the elements of the collection, F is an accumulation function of signature $T \times T \rightarrow T$, where T is the element type, and i is an iterator used to access the elements of the collection, i.e., the current element e is specified in terms of i.

Step 3: Unify intended functions of the code and the matching pattern. Once a matching pattern is found, the next step is to define a mapping or correspondence between variables, symbols, and expressions appearing in the intended functions of the loop body of the code and the matched pattern. This mapping will allow one to derive an intended function of the code from the skeletal intended function given by the pattern (see Step 4 below).

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to derive an intended function of the code by instantiating the skeletal intended function given in the pattern. For this, one needs to replace

variables, symbols, and expressions appearing in the skeletal intended function with the actual ones of the code using the binding defined in Step 3 above.

5.2. Example

In this section we illustrate in detail the application of the Accumulating pattern using the example code described in the previous section.

```
while (i < a.length) {
  if (a[i] > 0) {
     sum = sum + a[i];
  }
  i++;
}
```

Step 1: Formulate the intended function of the loop body. The code function of the loop body can be written straightforwardly; *a*[*i*] is added to *sum* only if it is positive, and *i* is always incremented by 1.

[sum, i := a[i] > 0 ? sum + a[i] : sum, i + 1]

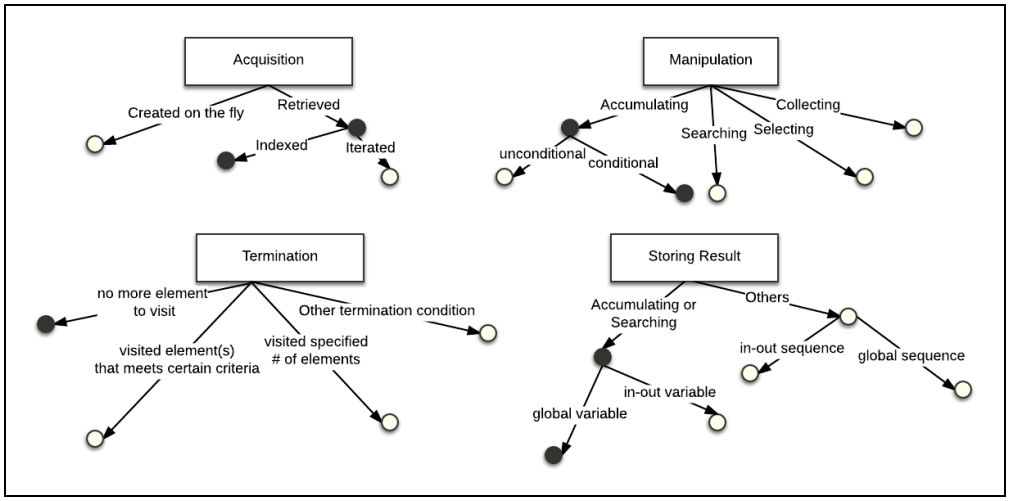
Step 2: Find a matching pattern. It is easy to see that the loop has the following characteristics along the four dimensions of analysis.

Acquisition: index-based (*i, a*[*i*])
Manipulation: addition (+)
Storing: scalar variable (*sum*)
Termination: all elements accessed (*i* < *a.length*)

And the intended function of the loop body matches that of the Accumulating pattern with the binding {$r \rightarrow sum$, $i \rightarrow i$, $e \rightarrow a[i]$, $P(e) \rightarrow e > 0$, $F(e, r) \rightarrow r + e$, $E(i) \rightarrow i + 1$}.

pattern: [r, i := *P*(e) ? *F*(e, r) : r, *E*(i)]
code: [sum, i := a[i] > 0 ? sum + a[i] : sum, i + 1]

Thus, the code matches the Accumulating pattern. We can also use decision trees to find a matching pattern as shown below. For each analysis dimension, we determine its value to find a matching pattern, in this case an index-based, conditional accumulation.



Step 3: Unify intended functions of the code and the matching pattern. We map variables, symbols, and expressions from the matching pattern to those of code, and the result is summarized below.

pattern: [r, i := *P*(s@i) ? (r ◇ s@i) : r, *E*(i)], [r, i := $\overline{◇}$(r, s@i..)**, anything**]
code: [sum, i := a[i] > 0 ? sum + a[i] : sum, i + 1]

Pattern	Code
s	a
r	sum
i	i
P(*x*)	x > 0
x@i	x[i]
x ◇ y	x + y
E(*x*)	x + 1
x@i..	x[i..a.length-1]

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to instantiate the skeletal intended function given by the pattern using the binding defined in the previous step.

[r, i := $\overline{◇}$ (r, s@i..), **anything**]

≡ [sum, i := $\overline{◇}$ (sum, a[i..a.length-1]), **anything**] where $\overline{◇}$ is now defined as follows.

$\overline{◇}$(e, ⟨⟩) ≡ e
$\overline{◇}$(e, h ⊦ t) ≡ e > 0 ? $\overline{◇}$ (e + h, t) : $\overline{◇}$ (e, t)

Note that $\overline{◇}$ denotes the sum of all positive values of the given array plus the given value, and thus it can be rewritten using a more familar mathematical notation: $\overline{◇}$ (*sum*, *a*[*i..a.length*]) ≡ *sum* + $\sum_{j=i..a.length-1}$ (*a*[*j*] > 0 ? *a*[*j*] : 0). Therefore, the derived intended function can be rewritten to:

[r, i := sum + $\sum_{j=i..a.length-1}$ (a[j] > 0 ? a[j] : 0), **anything**]

which matches the intention of the loop, i.e., calculating the sum of all positive numbers stored in the array *a*.

6. Variations and Related Patterns

There are many variations possible for the Accumulating pattern. For example, each possible value of the four analysis dimensions can be a variation, e.g., index-based vs. iterator-based acquisition. Below we describe some of the variations that are not mentioned in the description of the four analysis dimensions in the introduction of this document.

Selection: The loop body of the Accumulating pattern has a general form of [*r, i* := *P*(*e*) ? *F*(*e*, *r*) : *r*, *E*(*i*)], and one possible variation is the case where the condition *P* is always true; that is, there is no constraint and thus all elements are accumulated. In fact, this is so common we name it a separate pattern, *Unconditionally Accumulating* (see the following chapter). Another possible variation is the case where the condition *P* is written in terms of the iterator *i*, not the current element *e*. An example is to accumulate every even-indexed element of a collection; the predicate can be written as *P*(*i*) = *i* % 2 == 0.

Manipulation: It is frequently the case that an element is manipulated prior to its accumulation. To incorporate this into the pattern, the intended function of the loop body can be generalized to: [*r, i* := *P*(*e*) ? *A*(*M*(*e*), *r*) : *r*, *E*(*i*)]. The accumulation function *F*(*e*, *r*) is now rewritten to *A*(*M*(*e*), *r*), where *M*: T → S maps an element to another value, and A: S×S → S accumulates the mapped or manipulated value to the result. As shown, the type of the accumulated value (S) can be different from that of the element (T). An example is to count the number of positive values contained in an array, whose manipulation can be defined as a constant function that always returns 1.

Acquisition: Beside various ways of acquiring elements described in the introduction of this document, a loop can accumulate elements of more than one collection using either a single or mulitple iterators. For example, a loop can accumulate all values of two arrays using a single iterator, e.g., [*r*, *i* := *r* + *a*[*i*] + *b*[*i*], *i* + 1] or using two iterators, e.g., [*r*, *i*, *j* := *r* + *a*[*i*] + *b*[*j*], *i* + 1, *j* + 1].

Storage: It is possible for the Accumulating pattern to produce more than one accumulated value, meaning that it can have more than one result variable. An example is to sum all positive values as well as all negative values of an array; the loop body will have an intended function of the form [*pos*, *neg*, *i* := *pos* + (*a*[*i*] > 0 ? *a*[*i*] : *0*), *neg* + (*a*[*i*] < 0 ? *a*[*i*] : *0*), *i* + 1].

UNCONDITIONALLY ACCUMULATING

This is a special case of the Accumulating pattern (refer to the previous chapter) in which all elements are accumulated unconditionally. Because of its frequent occurrence, it is catalogued as a separate pattern. The pattern has the following structure.

```
[r, i := ◇̄(r, s@i..), anything]
while (C) {
  [r, i  := r ◇ s@i, E(i)]
  ...
}
```

As shown, the loop body accumulates every element of the collection *s*; that is, there is no predicate that selects or filters the elements to be accumulated. As in the Accumulating pattern, the final value of the result (or accumulation) variable is defined in terms of the promotion of the accumulation operator ◇ to a collection, as shown below.

$$\overline{\diamond}(e, \langle\rangle) \equiv e$$
$$\overline{\diamond}(e, h \vdash t) \equiv \overline{\diamond}\,(e \diamond h, t)$$

Below is an example loop along with its intended function that can be matched to the pattern. It adds all the elements of an array *a* starting at index *i* to a result variable *sum.*

```
// [sum, i := sum + ∑ j=i..a.length-1 a[j] , anything]
while (i < a.length) {
  // [sum, i  := sum + a[i], i + 1]
 sum = sum + a[i];
  i++;
}
```

This pattern can have several variations similar to those of the Accumulating pattern (refer to the previous chapter).

SEARCHING

1. Purpose

This pattern provides a skeleton intended function for those while loops that search for a particular element in a collection. The result of such a loop is typically the element found, but other results are possible including the position or index of the element found and a flag indicating whether an element is found or not.

2. Description

A while loop is commonly used to find an element in a collection, e.g., a maximum value of an array of numbers. The Searching pattern captures this use of while loops that iterate over elements of a collection and find a particular element or search for an element that meets a certain condition. The result of such a while loop is typically the element being searched. However, other possibilities are the position or index of the element being searched and a flag indicating whether an element is found or not. The intended function of the loop is defined by referring to the search criterion and the iterator used to access the elements of the collection.

3. Structure

This pattern has the following code structure, where the intended function f_1 capture the behavior of the whole loop and f_2 specifies the behavior of the loop body only.

```
[f1]
while (C) {
   [f2]
   …
}
```

As shown, the loop body is not given in skeletal code but is abstracted to an intended function so that any code segment that correctly implements the intended function can be matched to the pattern. Below we first explain the intended function of the loop body (f_2) and then describe the intended function of the whole loop (f_1) which is written in terms of the components of f_2.

3.1. The loop

The while loop for the Searching pattern has the structure shown below. We use a symbol *s* to denote the collection whose elements are to be searched. Since its elements are accessed in a certain order in a loop, it can be logically viewed as a sequence, and thus its elements can be denoted by their positions in the sequence. For this we introduce an abstract variable *i*—an abstraction of the iterator to access the elements of the collection—and we use a notation *s*@*i* to denote the *i*-th element of the collection *s*.

```
while (C) {
  [r, i  := P(r, s@i) ? M(s@i) : r, E(i)]
  ...
}
```

As specified by the intended function, the loop body may change the values of two state variables, *r* and *i*. The variable *r* stores the result, i.e., the searched value, and as explained previously *i* is an abstraction of the iterator to access the elements of the collection *s*. The new value of *r* is defined by using a conditional expression of the form E_1 ? E_2 : E_3, denoting either E_2 or E_3 depending on the value of a Boolean expression E_1. The following symbols and notation are used to define the values of *r* and *i*.

P(*r*, *e*): a predicate defined on a pair of the result and an element of the sequence *s*. It specifies the search criterion for elements contained in the sequence *s*, and is a function of signature S × T → Boolean, where S and T are the result type and the element type of the collection *s*.

M(*e*): a manipulation function of signature T → S, where T is the element type and S is the result type, that maps an element found to the result. Most often it is an identity function, meaning that the result is the same as the element found. However, the result doesn't have to be the same as the element found; for example, it can be a flag indicating the presence of an element in the collection, which can be modeled by defining *M* as a constant function that always returns true. Another common use of the manipulation function is to obtain only a certain part of a composite element.

E(*i*) **:** an expression written in terms of the abstract variable *i*. It represents an advancement of the iterator *i* to the next or another element. Examples include *i* + 1 for an index-based collection such as arrays and *i.next*() for an iterator-based collection.

The new value of *r* is the current element of *s* mapped by M (i.e., *M*(*s*@*i*)) if the current element (*s*@*i*) satisfies the search criterion (P(*r*, *s*@*i*)); otherwise, it's its old value. The new value of *i* is *E*(*i*) that represents an advancement of the iterator *i* to the next element.

3.2. The intended function

The intended function of the whole loop is defined below. In the intended function, the keyword **anything** indicates that one doesn't care about the final value of the iterator *i*; in general, this is the case for the iterator for it will be a local or incidental variable used only for accessing the elements of a collection.

[r, i := $\overline{\diamond}$(r, s@i..)**, anything**]
while (*C*) {
  [r, i := *P*(r, s@i) ? *M*(s@i) : r, *E*(i)]
  ...
}

The final value of *r* is defined using the following symbols and notations.

s@*i*..: a subsequence of *s* starting at *i* and selected using the advancement expression *E*(*i*). It is a sequence consisting of elements *s*@*i*, s@*E*(*i*), s@*E*(*E*(*i*)), s@*E*(*E*(*E*(*i*))), etc, and represents the elements of *s* that are iterated over by the loop. The last element of *s*@*i*.. is determined by the loop termination condition *C*, and if the condition fails at the first iteration, *s*@*i*.. is an empty sequence. The sequence is defined recursively as follows.

s@i.. ≡ s@i ⊦ s@*E*(i).. if *i* denotes a valid index or position of *s*
〈〉 otherwise

where 〈〉 denotes an empty sequence and ⊦ denotes concatenation of an element to a sequence.

$\overline{\diamond}$: a promotion of the manipulation function *M* to a sequence. It is a function of signature S × Seq(T) → S, where S is the result type, T is the element type of the sequence *s*, and Seq(T) is a sequence of elements of type T. It calculates the result from a given sequence using the manipulation function *M* and can be defined recursively as follows.

$\overline{\diamond}$(r, 〈〉) ≡ r
$\overline{\diamond}$(r, h ⊦ t) ≡ *P*(h) ? $\overline{\diamond}$ (*M*(h), t) : $\overline{\diamond}$ (r, t)

where 〈〉 denotes an empty sequence and ⊦ denotes concatenation of an element to a sequence. When the given sequence is empty, it returns the given result. When the given sequence is not empty and the first element (*h*) satisfies the search criterion (*P*), the first element is mapped using *M* and the function is recursively applied to the rest of the sequence. If the first element fails the search criterion, it is ignored and the function is recursively applied to the rest of the sequence.

The intended function states that the final value of *r* is $\overline{\diamond}$(*r*, *s*@*i*..), a mapped value (*M*) of the element of *s* at positions *i*, *E*(*i*), *E*(*E*(*i*)), and so on that satisfies the search criteria *P*.

4. Example Code

The while loop below finds a maximum value of an array *a* starting at index *i* and stores it to *r.* Later we will show how one can derive the intended function of the loop by using the Searching pattern.

```
// [r, i := MAX(r, a, i), anything]  where MAX(r,a,i) is i > a.length - 1 ? r : MAX(Math.max(r, a[i]), a, i+1)
while (i < a.length) {
  // [r, i  := a[i] > r ? a[i] : r, i + 1]
  if (a[i] > r) {
     r = a[i];
  }
  i++;
}
```

5. Application

5.1. Suggested Process

We suggest the following general process consisting of four steps for applying the pattern.

Step 1: Formulate the intended function of the loop body. The first step is to formulate and specify the behavior of the loop body because the pattern is specified in terms of the intended function of the loop body, not its code structure. If the code of the loop body doesn't contain any nested loops, its code function can be systematically calculated using a technique like *trace tables* [6]. If the loop body contains another loop, however, the intended function of the nested loop may be found first by applying one of the patterns in this catalog. The intended function or code function of the loop body should document the side effect of the code, i.e., state changes caused by a single iteration of the loop.

Step 2: Find a matching pattern. The next step is to match the loop to one of the patterns documented in this catalog. For this we suggest one to analyze the loop along the four dimensions described in the introduction of this document: (a) how it acquires the values to be manipulated in the loop body, (b) what operation or manipulation it performs on the acquired values, (c) where the result of the manipulation is stored, and (d) when it terminates the iteration [1]. It is likely that most of the analysis, especially acquisition, manipulation, and storing are already performed and documented in the intended function of the loop body. The intended function of the loop body will have the following general form:

$$[s_1, s_2, \ldots, s_n := M_1(e, s_1), M_2(e, s_2), \ldots, M_n(e, s_n)]$$

where s_i is a state variable whose value may be changed in the loop body, e is the current element of the collection being iterated, M_i is a manipulation function defining the new value of s_i possibly in terms of its current value and the current element of the collection. The state variable s_i can be either a result variable or an iterator, and the current element e is typically given in terms of an iterator. If the intended function has the following specific form, the loop can be matched to the Searching pattern.

$$[r, i := P(r, e) \; ? \; F(e) : r, E(i)]$$

where r is a result variable, P is a predicate defined on a pair of the result and an element of the collection, F is a manipulation function of signature T → S, where T and S are the element type and the result type, respectively, and i is an iterator used to access the elements of the collection, i.e., the current element e is specified in terms of i.

Step 3: Unify intended functions of the code and the matching pattern. Once a matching pattern is found, the next step is to define a mapping or correspondence between variables, symbols, and expressions appearing in the intended functions of the loop body of the code and the matched pattern. This mapping will allow one to derive an intended function of the code from the skeletal intended function given by the pattern (see Step 4 below).

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to derive an intended function of the code by instantiating the skeletal intended function given in the pattern. For this, one needs to replace

variables, symbols, and expressions appearing in the skeletal intended function with the actual ones of the code using the binding defined in Step 3 above.

5.2. Example

In this section we illustrate in detail the application of the Searching pattern using the example code described in the previous section.

```
while (i < a.length) {
  if (a[i] > r) {
     r = a[i];
  }
  i++;
}
```

Step 1: Formulate the intended function of the loop body. The code function of the loop body can be written straightforwardly; *r* is the maximum of *a*[*i*] and itself, and *i* is incremented by 1.

[r, i := a[i] > r ? a[i] : r, i + 1]

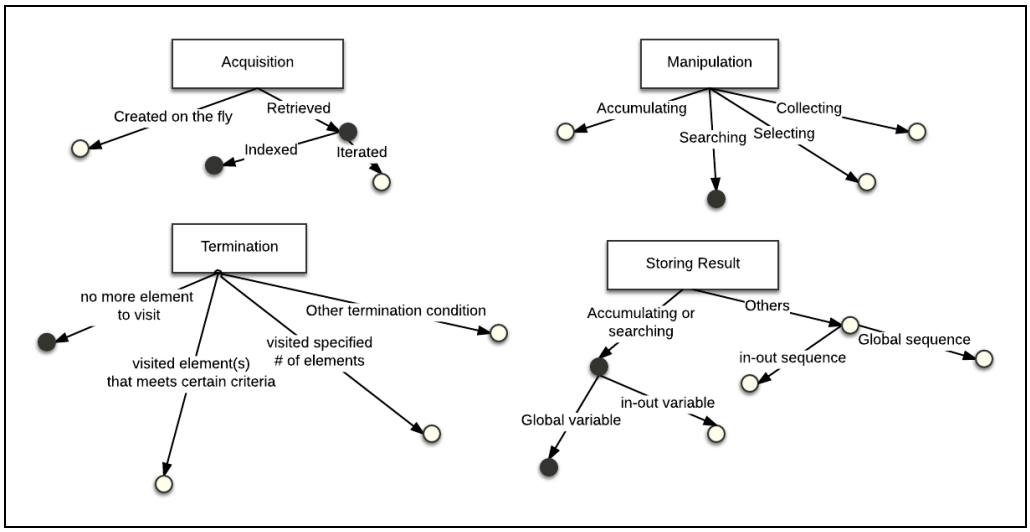
Step 2: Find a matching pattern. It is easy to see that the loop has the following characteristics along the four analysis dimensions.

Acquisition: index-based (*i, a*[*j*])
Manipulation: identity function (i.e., no manipulation)
Storing: scalar variable (*r*)
Termination: all elements accessed (*i* < *a.length*)

And the intended function of the loop body matches that of the Searching pattern with the binding {$r \rightarrow r$, $i \rightarrow i$, $e \rightarrow a[i]$, $P(r, e) \rightarrow e > r$, $M(e) \rightarrow e$, $E(i) \rightarrow i + 1$}.

pattern: [*r, i* := *P*(r, *e*) ? *M*(e,r): *r*, *E*(*i*)]
code: [r, i := a[i] > r ? a[i] : r, i + 1]

Thus, the code matches the Searching pattern. We can also use decision trees to find a matching pattern as shown below. For each analysis dimension, we determine its value to find a matching pattern, in this case an index-based searching.



Step 3: Unify intended functions of the code and the matching pattern. We map variables, symbols, and expressions from the matching pattern to those of code, and the result is summarized below.

pattern: [r, i := *P*(r, s@i) ? *M*(s@i) : r, *E*(i)], [r, i := $\overline{\diamond}$(r, s@i..)**, anything**]
code: [r, i := a[i] > r ? a[i] : r, i + 1]

Pattern	*Code*
s	a
r	r
i	i
P(r, *x*)	x > r
x@i	x[i]
M(x)	x
E(*x*)	x + 1
x@i..	x[i..a.length-1]

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to instantiate the skeletal intended function given by the pattern using the binding defined in the previous step.

[r, i := $\overline{\diamond}$ (r, s@i..), **anything**]

≡ [r, i := $\overline{\diamond}$ (r, a[i..a.length-1]), **anything**] where $\overline{\diamond}$ is now defined as follows.

$\overline{\diamond}$(e, ⟨⟩) ≡ e
$\overline{\diamond}$(e, h ⊦ t) ≡ e > 0 ? $\overline{\diamond}$ (e, t) : $\overline{\diamond}$ (e, t)

Note that $\overline{\diamond}$(*r*, *a*[*i*..*a*.*length*-1]) denotes the maximum of *r* and the elements of the array *a* starting at index i, and thus it can be rewritten in a more Java-like notation: *MAX*(*r*, *a*, *i*), where *MAX*(*r*, *a*, *i*) is defined recursively as *i* > *a*.*length* - 1 ? *r* : *MAX*(*Math*.*max*(*r*, *a*[*i*]), *a*, *i*+1). Therefore, the derived intended function can be rewritten as:

[r, i := MAX(r, a, i), **anything**], where MAX(r,a,i) is short for i > a.length - 1 ? r : MAX(Math.max(r, a[i]), a, i+1).

which matches the intention of the loop, i.e., finding a maximum of the array *a* starting at index *i*.

6. Variations and Related Patterns

There are many variations possible for the Searching pattern. For example, each possible value of the four analysis dimensions can be a variation, e.g., index-based vs. iterator-based acquisition. Below we describe some of the variations that are not mentioned in the description of the four analysis dimensions in the introduction of this document.

Selection: The loop body of the Searching pattern has a general form of [*r, i* := *P*(*r*, *e*) ? *M*(*e*) : *r*, *E*(*i*)], and one variation or specialization is the case where the condition *P* is written solely in terms of *e*, i.e., without referring to the result *r*. An example is to find a positive number contained in a collection, in which the current element of the collection doesn't have to be compared with the current result. The condition can also be generalized to include other variables, e.g., the iterator to restrict the search space to a sub-collection.

Manipulation: The intended function of the loop body accommodates manipulation or mapping of the element found prior to its storage in the result variable. However, it is frequently the case that no such manipulation is needed, and in this case the intended function can be simplified to [*r, i* := *P*(*r*, *e*) ? *e* : *r*, *E*(*i*)] by removing the manipulation function *M*; however, the element type should be assignment-compatible to the result type.

Acquisition: Beside various ways of acquiring elements described in the introduction of this document, a loop can search an element in more than one collection using either a single or mulitple iterators. . For example, a loop can

search a maximum value of two arrays using a single iterator, e.g., [*r*, *i* := *max*(*a*[*i*], *b*[*i*], r), *i* + 1] or using two iterators, e.g., [*r*, *i*, *j* := *max*(*a*[*i*], *b*[*j*], r), *i* + 1, *j* + 1], where *max*(*x*, *y*, *z*) denotes a maxium of *x*, *y*, and *z*.

Storage: It is possible for the Search pattern to produce more than one result. An example is to find both a positive value and a negative value in an array; the loop body will have an intended function of the form [*pos*, *neg*, *i* := *a*[*i*] > 0 ? *a*[*i*] : *pos*, *a*[*i*] < 0 ? *a*[*i*] : *neg*, *i* + 1].

Termintation: When searching an element in a collection, there are two possibilities for terminating the search, terminating as soon as an element is found and iterating to the last element of the collection. The first case is for finding the first occurrence of a matching element and the second for finding the last occurrence. In the pattern, this is somewhat implicity modeled by the sequence "*s*@*i*..". If needed, we can explicitly model the termination. For example, the first case can be captured by the following definition of $\overline{\diamond}$:

$\overline{\diamond}(e, \langle\rangle) \equiv e$
$\overline{\diamond}(e, h \vdash t) \equiv e > 0 \; ? \; M(h) \; : \; \overline{\diamond}(e, t)$

SELECTING

1. Purpose

This pattern provides a skeleton intended function for those while loops that select some elements of a collection and store the selected elements to the same or a different collection.

2. Description

A while loop is frequently used to select or filter some elements of a collection, and the Selecting pattern captures this use of while loops. The element type of the result collection is the same as that of the input collection. The intended function of the loop is defined by referring to the input and the result collections along with their iterators, and the criterion for selecting elements; iterators are used to access and store the elements of collections.

3. Structure

This pattern has the following code structure, where the intended function f_1 capture the behavior of the whole loop and f_2 specifies the behavior of the loop body only.

```
[f1]
while (C) {
   [f2]
   …
}
```

As shown, the loop body is not given in skeletal code but is abstracted to an intended function so that any code segment that correctly implements the intended function can be matched to the pattern. Below we first explain the intended function of the loop body (f_2) and then describe the intended function of the whole loop (f_1) which is written in terms of the components of f_2.

3.1. The loop

The while loop for the Selecting pattern has the structure shown below. We use a symbol *in* to denote the collection whose elements are to be selected or filtered. Since its elements are accessed in a certain order in a loop, it can be logically viewed as a sequence, and thus its elements can be denoted by their positions in the sequence. For this we introduce an abstract variable *i*—an abstraction of the iterator to access the elements of the collection—and we use a notation *in*@*i* to denote the *i*-th element of the collection *in*. Similarly we use symbols *out* and *j* to denote the result collection and its iterator, respectively.

```
while (C) {
  [out@j, i, j  := P(in@i) ? in@i : out@j, E1(i), E2(j)]
  ...
}
```

As specified by the intended function, the loop body may change the values of three state variables, *out*, *i* and *j*. The variable *out* contains the result, i.e., the selected elements, and as explained previously *i* and *j* are an abstraction of the iterators to access the elements of the collection *in* and *out,* respectively. The new value of *out* is defined by using a conditional expression of the form E_1 ? E_2 : E_3, denoting either E_2 or E_3 depending on the value of a Boolean expression E_1. The following symbols and notation are used to define the values of *out*, *i* and *j*.

P(*x*): a predicate defined on the elements of the sequence *in*. It's a function of signature T → Boolean, where T is the element type of the collection *in* and specifies the criterion for selecting an element. If *P*(*x*) is true for an element *x* of the sequence *in* then *x* should be selected.

$E_1(i)$**:** an expression written in terms of the abstract variable *i*. It represents an advancement of the iterator *i* to the next or another element. Examples include *i* + 1 for an index-based collection and *i*.*next*() for an iterator-based collection.

$E_2(j)$**:** an expression written in terms of the abstract variable *j*. It represents an advancement of the iterator *j* to the next or another element. Examples include *j* + 1 for an index-based collection and *j*.*next*() for an iterator-based collection.

The new value of *out*@*j* is the current element of *in* (i.e., *in*@*i*) if the current element satisfies the selection criterion (*P*(*in*@*i*)); otherwise, it's its old value. The new values of *i* and *j* are *E*(*i*) and E(*j*), respectively, that represent an advancement of the iterators *i* and *j* to the next elements. Operationally, it means that if the element in the sequence *in* at position *i* satisfies the condition *P*, it will be stored in the sequence *out* at position *j*; otherwise, the element of *out* in position *j* will not be changed. And *i* and *j* will be changed to advance their positions.

3.2. The intended function

The intended function specifies the behavior of a loop, e.g., for a while loop that alters the state of a collection *out*, it's intended function needs to specify which elements of the result collection *out* are changed. To express this in a representation-neutral way we view a collection abstractly as a partial function from its indices to elements. For example, a string sequence *s* consisting of two elements "Hello" and "World" can be viewed as a partial function from integers to strings defined as {0 → "Hello", 1 → "World"}. In this view our notation *s*@*i* can be interpreted as *s*(*i*), i.e., an application of *s* to *i*. We also introduce the following notation to manipulate collections abstractly.

dom: domain of a collection, e.g., dom *s* = {0, 1}. The result may be assumed to be an ordered set.

ran: range of a collection, e.g., ran s = {"Hello", "World"}. The result may be assumed to be an ordered bag.

⊕: function overriding. $c_1 \oplus c_2$ maps everything in the domain of c_2 the same value as c_2 does, and everything else in the domain of c_1 to the same value as c_1 does, e.g., *s* ⊕ {1 → "My", 2 → "Friend"} is {0 → "Hello", 1 → "My", 2 → "Friend"}.

Using this extended notation, the intended function of the whole loop is defined below. In the intended function, the keyword **anything** indicates that one doesn't care about the final value of the iterators *i* and *j*; in general, this is the case for an iterator for it will be a local or incidental variable used only for accessing the elements of a collection.

[out@D, i, j := R, **anything**, **anything**], where D and R are domain and range of $\overline{\diamond}$(in, out, i, j, {})
while (*C*) {
  [out@j, i, j := *P*(in@i) ? in@i : out@j, E_1(i), E_2(j)]
  ...
}

where {} denotes an empty function---a function with no mapping defined. The final value of *out* is defined using the following notations.

out@*D*: for a sequence *out* and an ordered index set *D*, a subsequence of *out* consisting of elements at positions *D*. It is a sequence consisting of elements projected by the index set *D* and can be defined recursively as follows.

s@⟨⟩ ≡ ⟨⟩
s@(h ⊢ t) ≡ s@h ⊢ s@t

where ⟨⟩ denotes an empty sequence and ⊢ denotes concatenation of an element to a sequence. The elements in set D can be defined recursively by the expression *j*, $E_2(j)$, $E_2(E_2(j))$, and so on.

$\overline{\diamondsuit}$: a function to determine all the elements to be selected along with their storage indices. The calculation is done by promoting the intended function of the loop body (specified at the element level) to collections (see Figure 2). That is, $\overline{\diamondsuit}(in, out, i, j, \{\})$ denotes the selected elements as a partial function from indices to values.

$$\overline{\diamondsuit}(\text{in, out, i, j, r}) \equiv$$

r	if $\neg C(\text{in, out, i, j})$
$\overline{\diamondsuit}(\text{in, out} \oplus \{j \rightarrow \text{in@i}\}, E_1(i), E_2(j), r \oplus \{j \rightarrow \text{in@i}\})$	if $C(\text{in, out, i, j}) \wedge P(\text{in@i})$
$\overline{\diamondsuit}(\text{in, out}, E_1(i), E_2(j), r)$	otherwise

where *C* is the loop termination condition written in terms of input collection *in,* output collection *out* and the iterator *i* and *j*, $\oplus$ denotes function overloading as described previously. The three cases represents (1) when the iteration is completed, (2) when the current element is selected, i.e., satisfies the selection criterion, and (3) when the current element is not collected, i.e., fails the selection criterion. The last argument *r* is an accumulator that stores index-value pairs for selected elements.

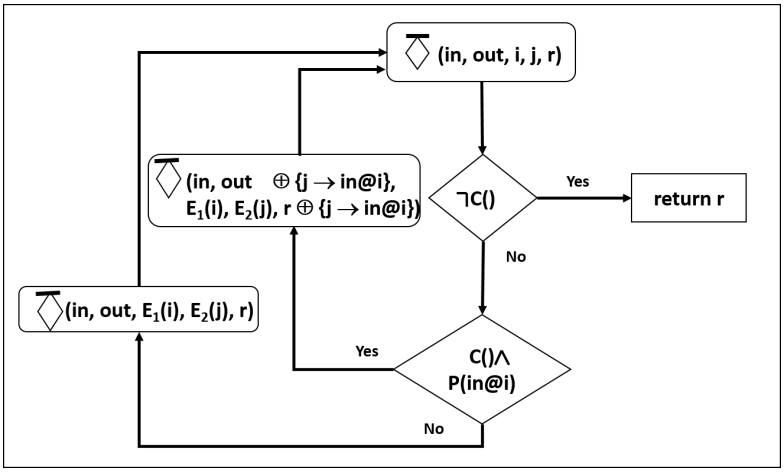


Figure 2: Definition of $\overline{\diamondsuit}$ function

The intended function states that the final value of *out* at positions *j*, $E_2(j)$, $E_2(E_2(j))$, and so on are those values of *in* at positions at *i*, $E_1(i)$, $E_1(E_1(i))$, and so on that satisfy the selection condition *P*. Figure 2 shows a visual representation of function $\overline{\diamondsuit}$.

4. Example Code

The while loop below copies all positive elements of an array *a* starting at index *i* to *b* starting at index *j*. Later we will show how one can derive the intended function of the loop by using the Selecting pattern.

```
/* [b[j..j+n-1], i, j := ◇̄a[i..a.length-1], anything, anything]
 *  where n is the number of positive values in array a starting at index i and ◇̄ is defined as follows.
 *      ◇̄(⟨⟩) ≡ ⟨⟩
 *      ◇̄(h ⊢ t) ≡ h > 0 ? (h ⊢ ◇̄(t)) : ◇̄(t)  */
while (i < a.length) {
    // [b[j], i, j := a[i] > 0 ?  a[i] : b[j], i + 1,  a[i] > 0 ? j + 1  :  j]
     if (a[i] > 0) {
         b[j] = a[i];
         j++;
     }
     i++;
}
```

5. Application

5.1. Suggested Process

We suggest the following general process consisting of four steps for applying this pattern.

Step 1: Formulate the intended function of the loop body. The first step is to formulate and specify the behavior of the loop body because the pattern is specified in terms of the intended function of the loop body, not its code structure. If the code of the loop body doesn't contain any nested loops, its code function can be systematically calculated using a technique like *trace tables* [6]. If the loop body contains another loop, however, the intended function of the nested loop may be found first by applying one of the patterns in this catalog. The intended function or code function of the loop body should document the side effect of the code, i.e., state changes caused by a single iteration of the loop.

Step 2: Find a matching pattern. The next step is to match the loop to one of the patterns documented in this catalog. For this we suggest one to analyze the loop along the four dimensions described in the introduction of this document: (a) how it acquires the values to be manipulated in the loop body, (b) what operation or manipulation it performs on the acquired values, (c) where the result of the manipulation is stored, and (d) when it terminates the iteration [1]. It is likely that most of the analysis, especially acquisition, manipulation, and storing are already performed and documented in the intended function of the loop body. The intended function of the loop body will have the following general form:

$$[s_1, s_2, \ldots, s_n := M_1(e, s_1), M_2(e, s_2), \ldots, M_n(e, s_n)]$$

where s_i is a state variable whose value may be changed in the loop body, e is the current element of the collection being iterated, M_i is a manipulation function defining the new value of s_i possibly in terms of its current value and the current element of the collection. The state variable s_i can be either a result variable or an iterator, and the current element e is typically given in terms of an iterator. If the intended function has the following specific form, the loop can be matched to the Selecting pattern.

$$[out@j, i, j \ := P(in@i) \ ? \ in@i : out@j, E_1(i), E_2(j)]$$

where *in* and *out* are input and output collections, P is a predicate defined on the elements of *in*, and i and j are iterators for *in* and *out*, i.e., the current elements are specified in terms of i and j.

Step 3: Unify intended functions of the code and the matching pattern. Once a matching pattern is found, the next step is to define a mapping or correspondence between variables, symbols, and expressions appearing in the intended functions of the loop body of the code and the matched pattern. This mapping will allow one to derive an intended function of the code from the skeletal intended function given by the pattern (see Step 4 below).

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to derive an intended function of the code by instantiating the skeletal intended function given in the pattern. For this, one needs to replace variables, symbols, and expressions appearing in the skeletal intended function with the actual ones of the code using the binding defined in Step 3 above.

5.2. Example

In this section we illustrate in detail the application of the Selecting pattern using the example code described in the previous section.

```
while (i < a.length) {
    if (a[i] > 0) {
        b[j] = a[i];
        j++;
    }
    i++;
}
```

Step 1: Formulate the intended function of the loop body. The code function of the loop body can be easily determined and is shown below; a[i] is conditionally copied to b[j].

[b[j], i, j := a[i] > 0 ? a[i] : b[j], i + 1, a[i] > 0 ? j + 1 : j]

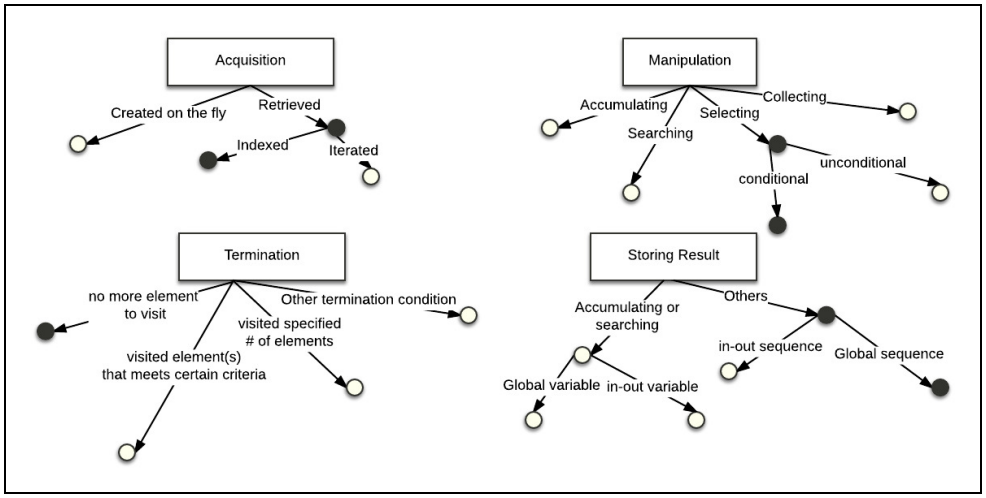
Step 2: Find a matching pattern. It is easy to see that the loop has the following characteristics along the four analysis dimensions.

Acquisition: index-based (*i, a*[*i*])
Manipulation: selecting (*a*[*j*] > 0 and *a*[*j*])
Storage: collection (*b*[*j*])
Termination: all elements accessed (*i* < *a.length*)

And the intended function of the loop body matches that of the Selecting pattern with the binding {*out* → *b*, *i* → *i*, *j* → *j*, *in* → *a*, *in@i* → *a*[*i*], *out@j* → *out*[*j*], *P*(*e*) → *e* > 0, $E_1(i) \rightarrow i + 1$, $E_2(j) \rightarrow a[i] > 0 \ ? \ j + 1 : j$}.

pattern: [out@j, i, j := *P*(in@i) ? in@i : out@j, E_1(i), E_2(j)]
code: [b[j], i, j := a[i] > 0 ? a[i] : b[j], i + 1, a[i] > 0 ? j + 1 : j]

Thus, the code matches the Selecting pattern. We can also use decision trees to find a matching pattern as shown below. For each analysis dimension, we determine its value to find a matching pattern, in this case an index-based, conditionally selecting.



Step 3: Unify intended functions of the code and the matching pattern. We map variables, symbols, and expressions from the matching pattern to those of code, and the result is summarized below.

pattern: [out@j, i, j := *P*(in@i) ? in@i : out@j, E_1(i), E_2(j)]
code: [b[j], i, j := a[i] > 0 ? a[i] : b[j], i + 1, a[i] > 0 ? j + 1 : j]

Pattern	Code
out	b
i	i
j	j
in	a
x@i	x[i]
P(x)	x > 0
E_1(x)	x + 1
E_2(x)	a[i] > 0 ? x + 1 : x

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to instantiate the skeletal intended function given by the pattern using the binding defined in the previous step.

[out@D, i, j := R, **anything**, **anything**] where D and R are domain and range of $\overline{\diamondsuit}$(in, out, i, j, ⟨⟩) with

$\overline{\diamondsuit}$(in, out, i, j, r) ≡

r	if ¬*C*(in, out, i, j)
$\overline{\diamondsuit}$(in, out ⊕ {j → in@i}, E_1(i), E_2(j), r ⊕ {j → in@i})	if *C*(in, out, i, j) ∧ *P*(in@i)
$\overline{\diamondsuit}$(in, out, E_1(i), E_2(j), r)	otherwise

≡ [b@D, i, j := R, **anything, anything**] where D and R are domain and range of $\overline{\diamondsuit}$(a, b, i, j, ⟨⟩) with

$\overline{\diamondsuit}$(in, out, i, j, r) ≡

r	if ¬(i < a.length)
$\overline{\diamondsuit}$(in, out ⊕ {j → in[i]}, i + 1, a[i] > 0 ? j + 1 : j, r ⊕ {j → in[i]})	if (i < a.length) ∧ a[i] > 0
$\overline{\diamondsuit}$(in, out, i + 1, a[i] > 0 ? j + 1 : j, r)	otherwise

From the definition, one can easily see that the instantiated function selects all positive values of *a* starting at index *i* and associates them to *b*'s indices starting at *j* and consecutively, i.e., *j*, *j* + *1*, *j* + *2*, and so on. Thus we can rewrite the intended function in a more familiar array-like notation.

[*b*[*j*..*j*+*n*-1], *i*, *j* := $\overline{\diamondsuit}$*a*[*i*..*a*.length-1], **anything, anything**] where *n* is the size of $\overline{\diamondsuit}$*a*[*i*..*a*.length-1] and $\overline{\diamondsuit}$ is

$\overline{\diamondsuit}$(⟨⟩) ≡ ⟨⟩

$\overline{\diamondsuit}$(*h* ⊦ *t*) ≡ *h* > *0* ? (*h* ⊦ $\overline{\diamondsuit}$(*t*)) : $\overline{\diamondsuit}$(*t*)

It can be easily seen that *n* is the number of positive elements in *a* starting at index *i*.

6. Variations and Related Patterns

There are many variations possible for this pattern. For example, each possible value of the four analysis dimensions can be a variation, e.g., index-based vs. iterator-based acquisition. Below we describe some of the variations that are not mentioned in the description of the four analysis dimensions in the introduction of this document.

Selection: The loop body of this pattern has a general form of [*out@j*, *i*, *j* := *P*(*in@i*) ? *in@i* : *out@j*, E_1(*i*), E_2(*j*)], and one possible variation is the case where the selection condition *P* is always true; that is, all elements are selected. In fact, this is so common we name it a separate pattern, *Unconditionally Selecting* (see the following chapter). Another possible variation is the one in which the condition *P* is written in terms of the iterator *i*, not in terms of the current element *in@i*. An example is to select every even-indexed element of a collection; the condition can be written as *P*(*i*) = *i* % 2 == 0.

Transformation: It is frequently the case that selected elements are transformed before they are stored. In fact it is so common that it is named and catalogued as a separate pattern, *Collecting* pattern (refer to later chapters).

Acquisition: Beside various ways of acquiring elements described in the introduction of this document, a loop can select elements of more than one collection using either a single or mulitple iterators. For example, a loop can select values of two arrays using a single iterator, e.g., [*c*[*i*], *i* := *a*[*i*] > *b*[*i*] ? *a*[*i*] : *b*[*i*], *i* + 1] or using two iterators, e.g., [*a*[*i*], *i*, *j* := *a*[*i*] > *b*[*j*] ? *a*[*i*] : *b*[*j*], *i* + 1, *j* + 1].

Storage: Instead of storing the selected elements to another collection, it is possible to store them to the input collection, e.g., [*a*[*i*-1], *i* := *a*[*i*], *i* + 1]. It is also possible to have more than one result collection. An example is to select both positive values and negative values of a collection. The loop body will have an intended function of the form:

```
[pos[i], neg[j], i, j, k :=
        a[k] > 0 ? a[k] : pos[i],
        a[k] < 0 ? a[k] : neg[j],
```

```
a[k] > 0 ? i + 1 : i,
a[k] < 0 ? j + 1 : j,
k + 1]
```

UNCONDITIONALLY SELECTING

This is a special case of the Selecting pattern (refer to the previous chapter) in which all elements are selected unconditionally. Because of its frequent occurrence, it is catalogued as a separate pattern. The pattern has the following structure.

```
[out@D, i, j :=  R, anything, anything], where D and R are domain and range of ◇̄(in, out, i, j, ⟨⟩)
while (C) {
  [out@j, i, j  := in@i, E1(i), E2(j)]
  ...
}
```

As in the Selecting pattern, the final value of *out* is defined using the following notation.

out@D: for a sequence *out* and an ordered index set *D*, a subsequence of *out* consisting of elements at positions *D*. It is a sequence consisting of elements projected by the index set *D* and can be defined recursively as follows.

$$s@\langle\rangle \equiv \langle\rangle$$
$$s@h \vdash t \equiv s@h \vdash s@t$$

where $\langle\rangle$ denotes an empty sequence and $\vdash$ denotes concatenation of an element to a sequence.

$\overline{\diamond}$: a function that calculates all the elements to be selected by the loop. The calculation is done by promoting the intended function of the loop body (specified at the element level) to collections. That is, $\overline{\diamond}(in, out, i, j, \langle\rangle)$ denotes the selected elements as a partial function from indices to values.

$$\overline{\diamond}(in, out, i, j, r) \equiv \begin{cases} r & \text{if } \neg C(in, out, i, j) \\ \overline{\diamond}(in, out \oplus \{j \to in@i\}, E_1(i), E_2(j), r \oplus \{j \to in@i\}) & \text{otherwise} \end{cases}$$

where $\oplus$ denotes function overriding (refer to the previous chapter).

The intended function states that the final value of *out* at positions j, $E_2(j)$, $E_2(E_2(j))$, and so on are the values of *in* at positions at i, $E_1(i)$, $E_1(E_1(i))$, and so on.

Below is an example loop along with its intended function that can be matched to the pattern. It selects or copies all elements of an array *a* starting at index *i* by storing them to *b* starting at index *j*.

```
// [b[i..a.length-1], i := a[i..a.length-1], anything]
while (i < a.length) {
    // [b[i], i, := a[i], i + 1]
     b[i] = a[i]
     i++;
}
```

This pattern can have several variations similar to those of the Selecting pattern (refer to the previous chapter).

COLLECTING

1. Purpose

This pattern provides a skeleton intended function for those while loops that select some elements of a collection, mutate or change them, and store the results to the same or a different collection. This pattern is a generalization of the Selecting pattern in that it may change the selected elements prior to their storage.

2. Description

A while loop is often used to collect some element of a collection in a sense that it picks the elements that satisfy a certain selection criterion, changes their values, and stores them in the same or different collection. The Collecting pattern captures this use of while loops. It is a generalization of the Selecting pattern, and the element type of the result collection may be different from that of the input collection. The intended function of the loop is defined by referring to the input and the result collections along with their iterators used to access and store the elements, the criterion for selecting elements, and the function to mutate the selected elements.

3. Structure

This pattern has the following code structure, where the intended function f_1 capture the behavior of the whole loop and f_2 specifies the behavior of the loop body only.

```
[f1]
while (C) {
   [f2]
   …
}
```

As shown, the loop body is not given in skeletal code but is abstracted to an intended function so that any code segment that correctly implements the intended function can be matched to the pattern. Below we first explain the intended function of the loop body (f_2) and then describe the intended function of the whole loop (f_1) which is written in terms of the components of f_2.

3.1. The loop

The while loop for this pattern has the structure shown below. We use a symbol *in* to denote the collection whose elements are to be collected. Since its elements are accessed in a certain order, it can be logically viewed as a sequence, and thus its elements can be denoted by their positions in the sequence. For this we introduce an abstract variable *i*—an abstraction of the iterator to access the elements of the collection—and we use a notation *in*@*i* to denote the *i*-th element of the collection *in*. Similarly we use symbols *out* and *j* to denote the result collection and its iterator, respectively.

```
while (C) {
  [out@j, i, j  := P(in@i) ? M(in@i) : out@j, E1(i), E2(j)]
  ...
}
```

As specified by the intended function, the loop body may change the values of three state variables, *out*, *i* and *j*. The variable *out* contains the result, i.e., the collected elements, and as explained previously *i* and *j* are an abstraction of the iterators to access the elements of the collection *in* and *out,* respectively. The new value of *out* is defined by using a conditional expression of the form E_1 ? E_2 : E_3, denoting either E_2 or E_3 depending on the value of a Boolean expression E_1. The following symbols and notation are used to define the values of *out*, *i* and *j*.

P(*x*): a predicate defined on the elements of the sequence *in*. It's a function of signature T → Boolean, where T is the element type of the collection *in* and specifies the criterion for collecting an element. If *P*(*x*) is true for an element *x* of the sequence *in* then *x* should be collected.

M(*x*): a function defined on the elements of the sequence *in*. It's a function of signature T → S, where T and S are the element type of *in* and *out*, respectively and models mutation of the selected elements by mapping them to possibly different values.

$E_1(i)$**:** an expression written in terms of the abstract variable *i*. It represents an advancement of the iterator *i* to the next or another element. Examples include *i* + 1 for an index-based collection and *i*.*next*() for an iterator-based collection.

$E_2(j)$**:** an expression written in terms of the abstract variable *j*. It represents an advancement of the iterator *j* to the next or another element. Examples include *j* + 1 for an index-based collection and *j*.*next*() for an iterator-based collection.

The new value of *out*@*j* is a mutated value of current element of *in* (i.e., *M*(*in*@*i*)) if the current element satisfies the selection criterion (*P*(*in*@*i*)); otherwise, it's its old value. The new values of *i* and *j* are *E*(*i*) and E(*j*), respectively, representing an advancement of the iterators *i* and *j* to the next elements. Operationally it means that if the element in the sequence *in* at position *i* satisfies the condition *P*, it will be stored in the sequence *out* at position *j* after mutation; otherwise, the element of *out* in position *j* will not be changed. And the values of iterators *i* and *j* will be changed to advance their positions.

3.2. The intended function

The intended function needs to specify which elements of the result collection *out* are changed. To express this in a representation-neutral way we view a collection abstractly as a partial function from its indices to elements. For example, a string sequence *s* consisting of two elements "Hello" and "World" can be viewed as a partial function from integers to strings defined as {0 → "Hello", 1 → "World"}. In this view our notation *s*@*i* can be interpreted as *c*(*i*), i.e., an application of *c* to *i*. We also introduce the following notation to manipulate collections abstractly.

dom: domain of a collection, e.g., dom *s* = {0, 1}. The result may be assumed to be an ordered set.

ran: range of a collection, e.g., ran s = {"Hello", "World"}. The result may be assumed to be an ordered bag.

⊕: function overriding. $c_1 \oplus c_2$ maps everything in the domain of c_2 the same value as c_2 does, and everything else in the domain of c_1 to the same value as c_1 does, e.g., *s* ⊕ {1 → "My", 2 → "Friend"} is {0 → "Hello", 1 → "My", 2 → "Friend"}.

Using this extended notation, the intended function of the whole loop is defined below. In the intended function, the keyword **anything** indicates that one doesn't care about the final value of the iterators *i* and *j*; in general, this is the case for an iterator for it will be a local or incidental variable used only for accessing the elements of a collection.

[out@D, i, j := R, **anything**, **anything**], where D and R are domain and range of $\overline{\diamond}$(in, out, i, j, ⟨⟩)
while (*C*) {
  [out@j, i, j := *P*(in@i) ? *M*(in@i) : out@j, E_1(i), E_2(j)]
  ...
}

The final value of *out* is defined using the following notation.

out@*D*: for a sequence *out* and an ordered index set *D*, a subsequence of *out* consisting of elements at positions *D*. It is a sequence consisting of elements projected by the index set *D* and can be defined recursively as follows.

s@⟨⟩ ≡ ⟨⟩

$s@h \vdash t \equiv s@h \vdash s@t$

where $\langle\rangle$ denotes an empty sequence and $\vdash$ denotes concatenation of an element to a sequence.

$\overline{\diamond}$: a function to determine all the elements to be collected along with their storing indices. The calculation is done by promoting the intended function of the loop body (specified at the element level) to collections (see Figure 3). That is, $\overline{\diamond}(in, out, i, j, \langle\rangle)$ denotes the collected elements as a partial function from indices to values.

$$\overline{\diamond}(\text{in, out, i, j, r}) \equiv$$

$$\begin{cases} r & \text{if } \neg C(\text{in, out, i, j}) \\ \overline{\diamond}(\text{in, out } \oplus \{j \rightarrow M(in@i)\}, E_1(i), E_2(j), r \oplus \{j \rightarrow M(in@i)\}) & \text{if } C(\text{in, out, i, j}) \wedge P(in@i) \\ \overline{\diamond}(\text{in, out, } E_1(i), E_2(j), r) & \text{otherwise} \end{cases}$$

where $\oplus$ denotes function overloading as described previously. The three cases represents (1) when the iteration is completed, (2) when the current element is collected, i.e., satisfies the collecting criterion, and (3) when the current element is not collected, i.e., fails the collecting criterion.

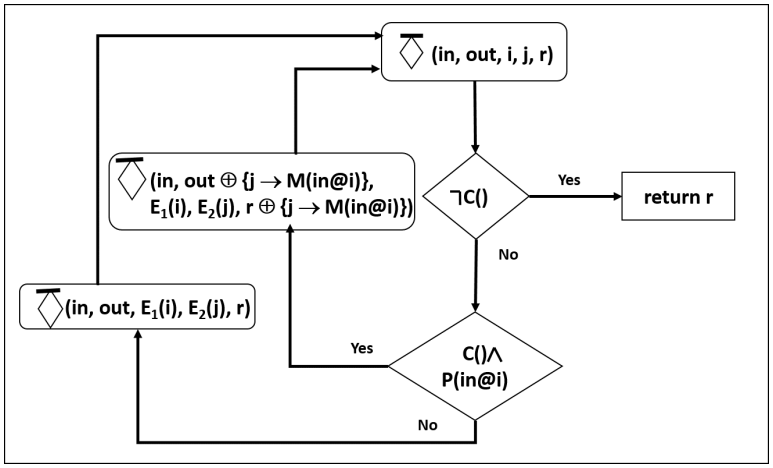


Figure 3: Definition of $\overline{\diamond}$ function.

The intended function states that the final value of *out* at positions *j*, $E_2(j)$, $E_2(E_2(j))$, and so on are those values of *in* at positions at *i*, $E_1(i)$, $E_1(E_1(i))$, and so on that satisfy the condition *P* and transformed with the function *M*.

4. Example Code

The while loop below collects all positive elements of an array *a* starting at index *i* by multiplying 2 to them and storing the results to an array *b* starting at index *j*. Later we will show how one can derive the intended function of the loop using the Collecting pattern.

```
/* [b[j..j+n-1], i, j := ◇̄a[i..a.length-1], anything, anything]
 *   where n is the number of positive values in array a starting at index i and ◇̄ is defined as follows.
 *     ◇̄(⟨⟩) ≡ ⟨⟩
 *     ◇̄(h ⊢ t) ≡ h > 0 ? (h *2 ⊢ ◇̄(t)) : ◇̄(t)  */
while (i < a.length) {

    // [b[j], i, j := a[i] > 0 ?  a[i] *2 : b[j], i + 1,  a[i] > 0 ? j + 1  :  j]
     if (a[i] > 0) {
         b[j] = a[i] * 2;
         j++;
     }
     i++;
```

```
}
```

5. Application

5.1. Suggested Process

We suggest the following general process consisting of four steps for applying this pattern.

Step 1: Formulate the intended function of the loop body. The first step is to formulate and specify the behavior of the loop body because the pattern is specified in terms of the intended function of the loop body, not its code structure. If the code of the loop body doesn't contain any nested loops, its code function can be systematically calculated using a technique like *trace tables* [6]. If the loop body contains another loop, however, the intended function of the nested loop may be found first by applying one of the patterns in this catalog. The intended function or code function of the loop body should document the side effect of the code, i.e., state changes caused by a single iteration of the loop.

Step 2: Find a matching pattern. The next step is to match the loop to one of the patterns documented in this catalog. For this we suggest one to analyze the loop along the four dimensions described in the introduction of this document: (a) how it acquires the values to be manipulated in the loop body, (b) what operation or manipulation it performs on the acquired values, (c) where the result of the manipulation is stored, and (d) when it terminates the iteration [1]. It is likely that most of the analysis, especially acquisition, manipulation, and storing are already performed and documented in the intended function of the loop body. The intended function of the loop body will have the following general form:

$$[s_1, s_2, \ldots, s_n := M_1(e, s_1), M_2(e, s_2), \ldots, M_n(e, s_n)]$$

where s_i is a state variable whose value may be changed in the loop body, e is the current element of the collection being iterated, M_i is a manipulation function defining the new value of s_i possibly in terms of its current value and the current element of the collection. The state variable s_i can be either a result variable or an iterator, and the current element e is typically given in terms of an iterator. If the intended function has the following specific form, the loop can be matched to the Collecting pattern.

$$[out@j, i, j \ := P(in@i) \ ? \ M(in@i) : out@j, E_1(i), E_2(j)]$$

where *in* and *out* are input and output collections of containing elements of type T, *P* is a predicate defined on the elements of *in*, *M* is a transformation function defined on the elements of *in*, and *i* and *j* are iterators for *in* and *out*, i.e., the current elements are specified in terms of *i* and *j*.

Step 3: Unify intended functions of the code and the matching pattern. Once a matching pattern is found, the next step is to define a mapping or correspondence between variables, symbols, and expressions appearing in the intended functions of the loop body of the code and the matched pattern. This mapping will allow one to derive an intended function of the code from the skeletal intended function given by the pattern (see Step 4 below).

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to derive an intended function of the code by instantiating the skeletal intended function given in the pattern. For this, one needs to replace variables, symbols, and expressions appearing in the skeletal intended function with the actual ones of the code using the binding defined in Step 3 above.

5.2. Example

In this section we illustrate in detail the application of the Collecting pattern using the example code described in the previous section.

```
while (i < a.length) {
    if (a[i] > 0) {
        b[j] = a[i] * 2;
        j++;
    }
```

```
        i++;
    }
```

Step 1: Formulate the intended function of the loop body. The code function of the loop body can be easily formulated; $a[i]$ is conditionally copied to $b[j]$ after multiplied by 2.

[b[j], i, j := a[i] > 0 ? a[i] * 2 : b[j], i + 1, a[i] > 0 ? j + 1 : j]

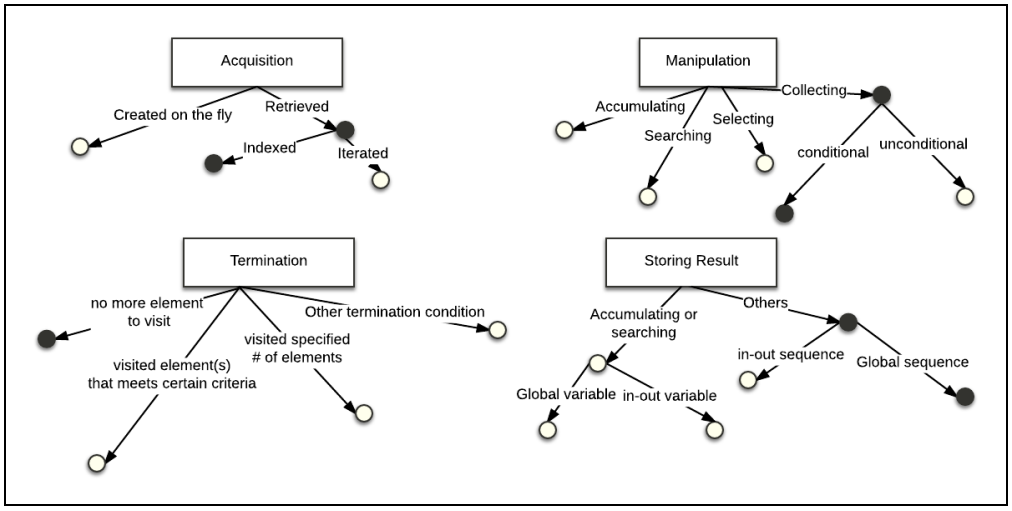
Step 2: Find a matching pattern. It can be easily determined that the loop has the following characteristics along the four analysis dimensions.

Acquisition: index-based (*i, a*[*i*])
Manipulation: collecting (*a*[*i*] * 2)
Storage: collection (*b*[*j*])
Termination: all elements accessed (*i* < *a.length*)

And the intended function of the loop body matches that of the Collecting pattern with the binding {$out \rightarrow b$, $i \rightarrow i$, $j \rightarrow j$, $in \rightarrow a$, $in@i \rightarrow a[i]$, $out@j \rightarrow out[j]$, $P(e) \rightarrow e > 0$, $M(e) \rightarrow e * 2$, $E_1(i) \rightarrow i + 1$, $E_2(j) \rightarrow a[i] > 0 ? j + 1 : j$}.

pattern: [out@j, i, j := *P*(in@i) ? *M*(in@i) : out@j, E_1(i), E_2(j)]
code: [b[j], i, j := a[i] > 0 ? a[i] * 2 : b[j], i + 1, a[i] > 0 ? j + 1 : j]

Thus, the code matches the Collecting pattern. We can also use decision trees to find a matching pattern as shown below. For each analysis dimension, we determine its value to find a matching pattern, in this case an index-based, conditional collecting.



Step 3: Unify intended functions of the code and the matching pattern. We map variables, symbols, and expressions from the matching pattern to those of code, and the result is summarized below.

pattern: [out@j, i, j := *P*(in@i) ? *M*(in@i) : out@j, E_1(i), E_2(j)]
code: [b[j], i, j := a[i] > 0 ? a[i] * 2 : b[j], i + 1, a[i] > 0 ? j + 1 : j]

Pattern	*Code*
in	a
out	b
i	i
j	j
x@i	x[i]
P(x)	x > 0

M(x)	x * 2
E_1(x)	x + 1
E_2(x)	a[i] > 0 ? x + 1 : x

Step 4: Instantiate the skeletal intended function of the pattern. The last step is to instantiate the skeletal intended function given by the pattern using the binding defined in the previous step.

[out@D, i, j := R, **anything**, **anything**], where D and R are domain and range of $\overline{\diamond}$(in, out, i, j, ⟨⟩) with

$$\overline{\diamond}(in, out, i, j, r) \equiv \begin{cases} r & \text{if } \neg C(in, out, i, j) \\ \overline{\diamond}(in, out \oplus \{j \rightarrow M(in@i)\}, E_1(i), E_2(j), r \oplus \{j \rightarrow M(in@i)\}) & \text{if } C(in, out, i, j) \wedge P(in@i) \\ \overline{\diamond}(in, out, E_1(i), E_2(j), r) & \text{otherwise} \end{cases}$$

≡ [b@D, i, j := R, **anything, anything**] where D and R are domain and range of $\overline{\diamond}$(a, b, i, j, ⟨⟩) with

$$\overline{\diamond}(in, out, i, j, r) \equiv \begin{cases} r & \text{if } \neg(i < in.length) \\ \overline{\diamond}(in, out \oplus \{j \rightarrow in[i]*2\}, i + 1, in[i] > 0 \ ? \ j + 1 \ : \ j, r \oplus \{j \rightarrow in[i]*2\}) & \text{if } (i < in.length) \wedge in[i] > 0 \\ \overline{\diamond}(in, out, i + 1, in[i] > 0 \ ? \ j + 1 \ : \ j, r) & \text{otherwise} \end{cases}$$

From the instantiated definition, one can see that the $\overline{\diamond}$ function collects all positive values of array *a* starting at index *i* by multiplying 2 and associating them to *b*'s indices starting at *j* and consecutively, i.e., *j*, *j* + *1*, *j* + *2*, and so on. Thus we can rewrite the intended function in a more familiar array-like notation.

[*b*[*j*..*j*+*n*-1], *i*, *j* := $\overline{\diamond}$*a*[*i*..*a*.length-1], **anything, anything**] where *n* is the size of $\overline{\diamond}$*a*[*i*..*a*.length-1] and $\overline{\diamond}$ is

$\overline{\diamond}(\langle\rangle) \equiv \langle\rangle$

$\overline{\diamond}(h \vdash t) \equiv h > 0 \ ? \ (h * 2 \vdash \overline{\diamond}(t)) : \overline{\diamond}(t)$

It can be easily seen that *n* is the number of positive elements contained in *a* starting at index *i*.

6. Variations and Related Patterns

There are many variations possible for this pattern. For example, each possible value of the four analysis dimensions can be a variation, e.g., index-based vs. iterator-based acquisition. Below we describe some of the variations that are not mentioned in the description of the four analysis dimensions in the introduction of this document.

Selection: The loop body of this pattern has a general form of [*out@j*, *i*, *j* := *P*(*in@i*) ? *M*(*in@i*) : *out@j*, E_1(*i*), E_2(*j*)], and one possible variation is the case where the collecting condition *P* is always true; that is, all elements are collected. In fact, this is so common we name it a separate pattern, *Unconditionally Collecting* (see the following chapter). Another possible variation is the one in which the condition *P* is written in terms of the iterator *i*, not in terms of the current element *in@i*. An example is to collect every even-indexed element of a collection; the condition can be written as *P*(*i*) = *i* % 2 == 0.

Transformation: It is frequently the case that the selected elements are stored without being changed or mutated. In fact it is so common that it is named and catalogued as a separate pattern, *Selecting* pattern (refer to the previous chapter). Collecting indices of the elements is also another form of transformation. In that case mapping will be from value to its position *(in@i →i)*

Acquisition: Beside various ways of acquiring elements described in the introduction of this document, a loop can collect elements of more than one collection using either a single or mulitple iterators. For example, a loop can collect values of two arrays using a single iterator, e.g., [*c*[*i*], *i* := *a*[*i*] + *b*[*i*], *i* + 1] or using two iterators, e.g., [*a*[*i*], *i*, *j* := *a*[*i*] + *b*[*j*], *i* + 1, *j* + 1].

Storage: Instead of storing the collected elements to another collection, it is possible to store them to the input collection, e.g., [*a*[*i*], *i* := *a*[*i*] < 0 ? -*a*[*i*] : *a*[*i*], *i* + 1]. It is also possible to have more than one result collection. An example is to calculate both an element-wise sum and product of two collections. The loop body will have an intended function of the form [*sum*[*i*], *prod*[*i*], *i* := *a*[*i*] + *b*[*i*], *a*[*i*] * *b*[*i*], *i* + 1].

UNCONDITIONALLY COLLECTING

This is a special case of the Collecting pattern (refer to the previous chapter) in which all elements are collected unconditionally. Because of its frequent occurrence, it is catalogued as a separate pattern. The pattern has the following structure.

[out@D, i, j := R, **anything**, **anything**], where D and R are domain and range of $\overline{\diamond}$(in, out, i, j, ⟨⟩)
while (*C*) {
  [out@j, i, j := *M*(in@i), E_1(i), E_2(j)]
  ...
}

As in the Collecting pattern, the final value of *out* is defined using the following notation.

out@*D*: for a sequence *out* and an ordered index set *D*, a subsequence of *out* consisting of elements at positions *D*. It is a sequence consisting of elements projected by the index set *D* and can be defined recursively as follows.

$s@\langle\rangle \equiv \langle\rangle$
$s@h \vdash t \equiv s@h \vdash s@t$

where ⟨⟩ denotes an empty sequence and ⊢ denotes concatenation of an element to a sequence.

$\overline{\diamond}$: a function that calculates all the elements to be collected by the loop. The calculation is done by promoting the intended function of the loop body (specified at the element level) to collections. That is, $\overline{\diamond}$(*in*, *out*, *i*, *j*, ⟨⟩) denotes the collected elements as a partial function from indices to values.

$$\overline{\diamond}(in, out, i, j, r) \equiv \begin{cases} r & \text{if } \neg C(in, out, i, j) \\ \overline{\diamond}(in, out \oplus \{j \rightarrow M(in@i)\}, E_1(i), E_2(j), r \oplus \{j \rightarrow M(in@i)\}) & \text{otherwise} \end{cases}$$

where ⊕ denotes function overloading (refer to the previous chapter).

The intended function states that the final value of *out* at positions *j*, E_2(*j*), E_2(E_2(*j*)), and so on are the values of *in* at positions at *i*, E_1(*i*) , E_1(E_1(*i*)), and so on transformed with the function *M*.

Below is an example loop along with its intended function that can be matched to the pattern. It collects all elements of an array *a* starting at index *i* by multiplying 2 and storing them to *b* starting at index *j*.

```
/* [b[i..a.length-1], i := ◇a[i..a.length-1], anything], where ◇ is defined as follows.
 *     ◇(⟨⟩) ≡ ⟨⟩
 *     ◇(h ⊢ t) ≡ h *2 ⊢ ◇(t)  */
while (i < a.length) {
    // [b[i], i, := a[i] *2, i + 1]
    b[i] = a[i] * 2;
    i++;
}
```

This pattern can have several variations similar to those of the Collecting pattern (refer to the previous chapter).

REFERENCES

[1] A. Barua, Y. Cheon, Finding Specifications of While Statements Using Patterns, *International Joint Conferences on Computer, Information, and Systems Sciences, and Engineering*, December 12-14, 2013, Springer.

[2] Y. Cheon, *Functional Specification and Verification of Object-oriented Programs*, Department of Computer Science, The University of Texas at El Paso, El Paso, TX, Technical Report 10-23, Aug. 2010.

[3] Y. Cheon, C. Yeep, and M. Vela, The CleanJava Language For Functional Program Verification, *International Journal of Software Engineering*, 5(1):47-68, Jan. 2012.

[4] E. Gmma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[5] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering.* Addison Wesley, Feb. 1999.

[6] A. Stavely, *Toward Zero Defect Programming*. Addison-Wesley, 1999.