4.2 From Code Smells to Refactoring Rules

Based on the criteria listed in section 3.2, we are selecting code smells that fit under *Method level* and *Redundant code*. As some of those reported smells are semantically very close, we decided to group them into more generic pattern descriptions. This resulted into the nine language-agnostic patterns which are listed in table 3.

Description of code patterns. The first two patterns correspond to the use of boolean variables. A common example of pattern P1 is to explicitly check if a boolean variable is true [24] in a conditional statement.

Pattern P2 is described in [47] as "checking a Boolean condition using an `if`-statement and then explicitly returning `true` or `false`." (For specific variations to this pattern refer to semantic style indicators IFRC, IFRT and IFASNB presented in [12]). Wellesley college's lesson on boolean expressions (see [11]) provides a detailed description of patterns P1 and P2 and the range of transformations that could simplify redundant expressions.

Table 3. Catalogue of Novice Code Smells inside a method

	Code smell	Description	Example	Detection	
				Clion	Visual Studio

P1	Simplify Boolean Expression	No need to check if Boolean variable or expression is equal to true or false	if (divis == 0):
			  divisor = true
			else
			  divisor = false
				✓	

P2	Simplify Boolean Return	Explicitly returning true and false after checking boolean with condition	if (num1 < 15 && num2 < 30): return *True* return *False*	✓	✓

P3	Collapsible Nested ifs	avoiding deep nested ifs by using and (&&) operator	if a > b:
			  if c > a:
			  print("middle value is %d", a)
				✓	✓

P4	Consolidate Conditional Expressions	avoid a series of consecutive if statements that when true they result in the same action by using or (||) operator	if work.seniority < 2:
			  return 0
			if work.seniority > 15:
			  return 0		

P5	Redundant conditional Check	Checking an expression that has a known True/False value	if 'foo' in myList:
			  print("Yay")
			elif not 'foo' in myList:
			  print("Nay")
				✓	✓

P6	No Use of else	Series of if statement with opposite conditions instead of using else	if (x % 2 == 0):
			  print("Is even")
			if (x % 2 == 1):
			print("Is odd")		

P7	Duplicate if-else Body	Repeated statement(s) in branches of conditional statement (could be taken out)	if (value%2 == 0):
			  numEvens++;
			  count++;
			else:
			  numOdds++;
			  count++;
				✓	

P8	Empty if Statement	The block associated with the if (or else) has no statements (or self-assignment)	if x > 20:
			  x = x/2
			else:
			  x = x // or continue
				✓ *	

P9	Dead Code	Code that is never executed	----------		

Pattern P3 covers nested conditionals, considered a code smell as they make the code harder to understand. They may grow more complicated over time if you keep adding more conditions. For simplicity, the pattern refers only to the nesting of simple `if` statements; nested `if`s with additional `else`/`elif` clauses could also be collapsed but they may require more complex logical transformations (see examples given in Figure 9 at [29]).

Pattern P4 has multiple tests that lead to the same effect (such as returning zero in the example of Table 3). Similarly to P3, it will be more readable if we group all guard clauses that lead to that return into a flat `if` command. In contrast to the first two novice "anti-patterns", P3 and P4 are common code smells for both novices and intermediate programmers (see Refactoring Guru, [17]).

Another common habit for novices is to check the implicit condition on the `else` branch, adding an unnecessary `if` (called UIF in [12]). Pattern P5 covers the use of a conditional expression that is always *True* (or *False*). If the condition is always false, the code associated to it is dead code, thus both the condition and the block should be removed. If the condition is always true, we should remove the condition so that the associated block is executed.

Pattern P6 is cited as a typical novice pattern in [46], and involves writing a series of consecutive `if` statements using conditions that are mutually exclusive. For example, comparing two numbers with 3 statements (>, ==, <). This means, the three conditions are evaluated every time, hence it is better to use a single `if` statement with two `else` clauses.

Pattern P7 indicates the same fragment of code appears in all branches of a conditional statement, such as the increment of variable *count* in the example shown in table 3 (which matches figure 12 from [12]). This code smell is common for both novices and intermediate programmers, called "consolidate duplicate conditional fragment" in [16].

Pattern P8 indicates conditionals with empty blocks, which have been reported within student code [12], and are detected by code style tools. A related novice pattern that reflects low code comprehension is the inclusion of a (useless) `else` block that rewrites a variable with its current value, (named self-assignment in [24]). Note *self-assignment* is equivalent to an empty block as both can be removed without changing the program's logic.

Finally, it is not uncommon to find dead code in students' submissions. It is impossible without additional information to know if they were unaware the code is not reachable, or if they knew but did not care to clean it up. Thus, pattern P9, a generic "dead code" pattern, is included to challenge students to think about their program flow; it may also help them introspect whether or not parts of their code will ever be used.

Pattern detection. We typed examples of pattern P1 to P8 into the IDEs used by our students — Eclipse, CLion and VisualStudio (with the Resharper C++ extension) — in order to explore refactoring's feedback from each system. Eclipse did not identify any, the outcomes of the other systems are shown in the last two columns of Table 3. CLion did best but missed P4, P6 and examples of P8 without self-assignment.

From patterns to rules. After the list of code smells to target at novice level was established, we reshuffled and grouped the patterns by the similarity of their transformations (for simplicity Table 3 already shows them in that order).

The final set of rules is summarised in table 4. The first two rules focus on making conditional expressions and conditional statements more concise, as they play a significant role in readability and program flow. Latter rules focus on removing redundancies; the third rule addresses redundant conditions and the last rule asks students to look for redundant statements, such as duplicated ones (P7), useless ones (P8) or more general dead code (P9).

Initially novices are expected to detect patterns P7 and P8, but as they progress on their code comprehension and coding practice, they should be able to identify and deal with other occurrences of duplicated/dead code.

When introducing each rule to students, we will rely on examples from each code pattern to model the transformations needed to clean those smells. Figure 3 illustrate the examples given when introducing the first rule (note we have

colour-coded them by setting in **red** the parts removed upon refactoring, and in **blue** when modified). The full set of examples will be provided as supplementary material.

Table 4. Refactoring rules and associated patterns

Rule	Short description	Patterns
1	Do not check a Boolean variable – you can use it directly.	P1, P2
2	Collapse IF statements into one	P3, P4
3	Look for and remove redundant conditions known to be true/false	P5, P6
4	Look for and remove redundant statements	P7, P8, P9

	Original code	Refactored code
Check if true	if a == true: print("it is correct")	if a: print("it is correct")
Check if false	if a == false: print("it is false")	if not a: print("it is false")

Check and assign	if a == true:
	b = false
	else
	b = true
		b = not a

Check and return	def **function**():
	if isinstance(a, b) or issubclass(b, a):
	return True
	return False
		def **function**():
		return isinstance(a, b) or issubclass(b, a)

	Original code	Refactored code

Collapsible Template 1	if condition1:
	if condition2:
	# ...
		if condition1 and condition2:
		# ...

Nested example	if a > b:
	if c > a:
	print("both conditions are true")
		if a > b and c > a:
		print("both conditions are true")

Collapsible Template 2	if condition1:
	do-this
	if condition2:
	do-this
		if condition1 or condition2:
		do-this

consecutive example	if work.seniority < 2:
	return 0
	if work.seniority > 15:
	return 0
		if work.seniority < 2 or
		work.seniority > 15:
		return 0

Fig. 2. Rules 1 and 2 pattern descriptions using concrete examples and their refactored counterparts

	Original code	Refactored code

Redundant else question	if 'foo' in myList:
	  print("Yay")
	elif not 'foo' in myList:
	  print("Nay")
		if 'foo' in myList:
		  print("Yay")
		else:
		  print("Nay")

Consecutive if statements that are related	if (x % 2 == 0):
	  print("Is even")
	if (x % 2 == 1):
	  print("Is odd")
		if (x % 2 == 0):
		  print("Is even")
		else:
		  print("Is odd")

An example of True condition due to flow (assume no update of x)	if x <= 0
	  return -1
	if x > 20
	  do-something
	elif x > 0
	  do-something-else
		if x <= 0
		  return -1
		if x > 20
		  do-something
		else
		  do-something-else

	Original code	Refactored code

Duplicated Statement in if/else	if sold > DISCOUNT_AMOUNT:
	  total = sold * DISCOUNT_PRICE
	  label = f'Total: {total}'
	else:
	  total = sold * PRICE
	  label = f'Total: {total}'
		if sold > DISCOUNT_AMOUNT:
		  total = sold * DISCOUNT_PRICE
		else:
		  total = sold * PRICE
		label = f'Total: {total}'

Self-assignment	count = count	// *remove it – it has no effect*

first value never used	var = 0
	// sequential code that ignores var
	var = 1
		// sequential code ignores var
		var = 1

Value reset (not redundant)	var = 0
	// code that read/updates var
	var = 0
		// *No refactoring is possible*

Fig. 3. Rules 3 and 4 pattern descriptions using concrete examples and their refactored counterparts

4.3 Rule validation

Finally, as explained in section 3.3, we examine the applicability of the four refactoring rules to code written by novice programmers. We considered students' answers to the question in figure 4(a). From the ninety-five practical examination papers, 18 were empty or poor attempt, 31 were partially correct and 46 students (48%) wrote correct code that given the values (m,n) would produce the correct list. However, only six of them were quality solutions with no need to refactor.

As in other novice programmer's studies [23, 29], the solution space is large. By ignoring minor style changes and focusing on the conditional statement structure, we narrowed that solution space into six different approaches, five of which are shown in Figure 4. We are primarily interested in seeing if those correct solutions could be refactored, however we have also classified partial attempts to illustrate novice's utilisation of conditional statements.

Single-if (20 entries, 10 correct) The optimal solution, shown in figure 4, contains a `for`-loop with a single if statement. 10 students correctly wrote a loop with a single nested if, Note that only six students matched the optimal solution. The other 4 correct submissions tested two requested but *redundant* conditions (see figure 3.(b)) since the index "i" already fits into the `for`-loop range.