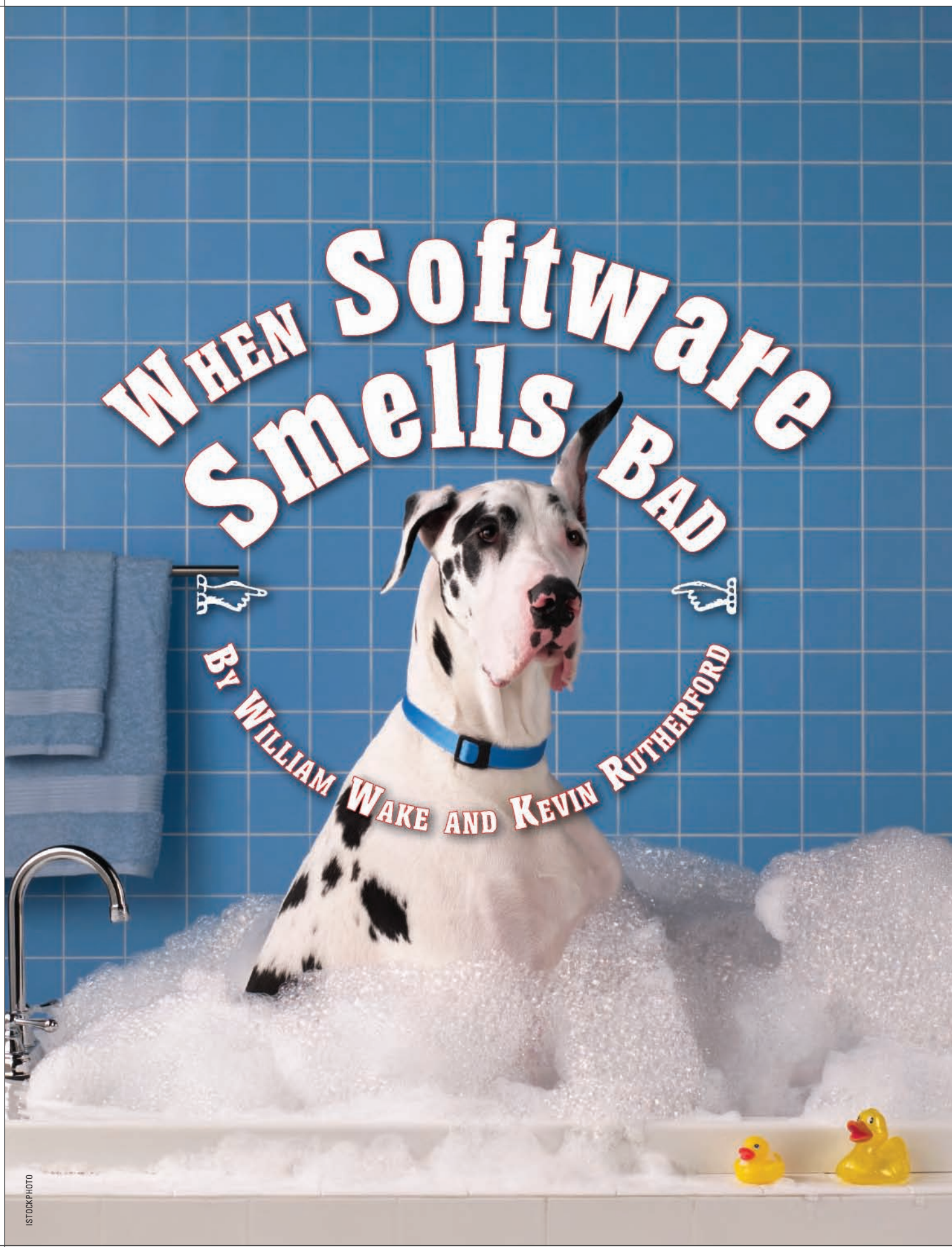
When Software Smells Bad
By William Wake and Kevin Rutherford
ISTOCKPHOTO

Imagine a piece of software that works correctly and yet doesn't "feel" well written. How could you describe the code's quality in concrete terms? How do you convert the vague notion—"this is poor code"—into a plan for fixing it? Let us introduce a catalog of software design anti-patterns called code smells and show how to use them both to describe software quality in concrete terms and to plan remedial actions for improving code.

What Is Code Quality?

In recent years, agile methods in general and test-driven development (TDD) in particular have become significantly

```
t = []          # tasks
skp = []        # skip count for each task
c = 0           # index of current task

print "TODO> "
STDOUT.flush
line = gets.chomp

while (!(line == "quit"))
 rest = line.sub(/[^ ]+ /, '')    # remove command from front of line

  case line
  when /^todo/
   t << rest
   skp << 0

  when /^done/
   if (t.length > 0)
    t.delete_at(c)
    skp.delete_at(c)
    c = 0 if c >= t.length
   end

  when /^skip/
   if (t.length > 0)
    skp[c] = skp[c] + 1

    if (skp[c] == 3)
      puts "Too many skips; deleting - " + t[c]
      t.delete_at(c)
      skp.delete_at(c)
      c = 0 if c >= t.length
    else
      c = (c + 1).modulo t.length
    end

   end

  when /^list/
   puts t
   puts

  when /^$/
   # do nothing

  else
   puts "Unknown command. Commands: todo, done, skip, list, or quit"
  end

  if (t.length > 0)
    puts "Current: " + t[c]
  end

  print "TODO> "
  STDOUT.flush
  line = gets.chomp
end
```

Figure 1: todo.rb

```
TODO> todo Make the bed
Current: Make the bed
TODO> todo Check email
Current: Make the bed
TODO> skip
Current: Check email
TODO> done
Current: Make the bed
TODO> todo Wash the dishes
Current: Make the bed
TODO> skip
Current: Wash the dishes
TODO> skip
Current: Make the bed
TODO> skip
Too many skips; deleting - Make the bed
Current: Wash the dishes
TODO>
```

Figure 2: A sample session

more popular. The microprocess of TDD tells us to work to the rhythm of red-green-refactor. Specifically, we write a test and see it fail; then, we write code in the most straightforward way possible to make it pass; then, we refactor to clean up any mess we made. But, in that third step, just what should we refactor and why? And if we need to refactor a pile of code that wasn't developed in a test-driven manner, how do we get a handle on where to begin?

Refactoring is defined as "improving software without changing what it does." The intent is to replace bad code with something better. The next questions must then be: What is bad code and what is good code? How can we recognize which is which? And, when we find bad code, what should we do to eliminate it?

The first step is to understand why code quality matters. The answer to that lies in our need to change the code in the future. If this code will never be read again and if it will never be changed, then it only has to be functionally correct. Such code can be messy and disorganized as long as it functions correctly. But, if in the future someone needs to fix a bug, add a new feature, or make a small tweak to the code's behavior, then we need good code. The moment we discover that the code is tortuous to navigate, difficult to understand, or hard to change is the moment we wish it were better written. Over and above functional correctness (which we take as a given in this article), we can define software quality as the ability to change the code easily: Good code supports change; bad code hampers change.

Exercise, Part 1

Before we go any further, look at the program in figure 1. This is a small script written as a command line to-do list. The script is written in Ruby but should be fairly easy to understand even if you are not familiar with scripting languages. If you haven't seen much Ruby, these hints will get you through:

- The "case/when" block is similar to a "switch/case" statement in other languages (but with no fall-through from one case to the next).
- Arrays (the variables initialized to "[]") have a "<<" operator that appends an item to the end of the array.
- Regular expressions are bracketed by "//" and form a kind of pattern object.

The application is written as a big `while` loop, reading input lines and interpreting them as commands. To use this tool, you add new tasks to the

A Few Important Smells

Uncommunicative Name

What to Look For:

- One- or two-character names, vowels omitted, or misleading names

What to Do:

- Rename the variable, class, etc.

What to Look for Next:

- Duplication: Look for places where the same thing has different names.

Feature Envy

What to Look For:

- Code references another object more than it references itself, or several clients manipulate a particular type of object in similar ways.

What to Do:

- Extract Method to isolate the similar code.
- Move Method to put it with the referenced object.

What to Look for Next:

- Duplication: Look for further duplication around the clients.
- Communication: Review names for the receiving class for consistency.

Long Method

What to Look For:

- A method with a large number of lines (five lines is large in Ruby)

What to Do:

- Extract Method to break up the long code.

What to Look for Next:

- Duplication: Are the extracted pieces similar? Can they be consolidated?
- Communication: Review names to make sure they communicate well.
- Abstraction: Is there a missing class?
- Flexibility: Is there Feature Envy, where code seems more concerned with another class than with its own class?

Duplicated Code

What to Look For:

- Code that is nearly identical in its text or in its effects

What to Do:

- For code with similar effects, Substitute Algorithm to make the code have similar text.

- For duplication within a class, Extract Method to isolate common parts.
- For duplication among sibling classes, Pull Up Method and Pull Up Instance Variable to bring common parts together. Consider whether there should be a Template Method.
- For duplication among unrelated classes, extract the common part into a separate class or module, or consolidate code onto one class.

What to Look for Next:
- Abstraction: Look for related responsibilities and missing abstractions.

Greedy Module
What to Look For:
- A method with more than one responsibility (especially decisions that will change at different frequencies)
- Clumsy test fixture setup

What to Do:
- Extract Class or Extract Module to split up the module.

What to Look for Next:
- Communication: Review names to make sure they communicate well.
- Simplicity: Check for Feature Envy.
- Testability: Simplify the unit tests.

Open Secret
What to Look For:
- Several classes or modules know how to interpret a simple value.
- Several classes or modules know what data is held in each slot of an array or hash.

What to Do:
- Extract Class or Introduce Parameter Object to consolidate the interpreting code.
- For an array or hash, Replace Array (or Hash) with Object.

What to Look for Next:
- Duplication: Look for Feature Envy around the new class.
- Communication: Review related names to make sure they communicate well.
- Flexibility: Consider whether the new object can be used earlier in time.

(Summarized from *Refactoring in Ruby* [2])

list via the "todo" command. See figure 2 for a sample session. Working the list in order, you can either mark the current task "done" or skip it. If you skip a task three times, it's deleted; you'll have to manually re-add it if you intend to do it. If you move past the end of the list, you go back to the first not-done task and start over. There are two other commands: "list" prints a list of all incomplete tasks, and "quit" exits the system (losing all the data—this code hasn't addressed persistence; it's just a simple example).

You can download this code, along with some tests that demonstrate its behavior, from the StickyNotes.

Take five minutes to look over the code and become familiar with it. Now, imagine you have been given responsibility for the future maintenance of this script. Review the code's changeability in response to the following potential change requests:

- Add persistence.
- Add a more sophisticated user interface (web or GUI).
- Add the notion of a recurring task, one that is automatically added to the end of the list when it is marked "done." (Think of a task that never goes away, such as "check email.")
- Change the rule for choosing the next task. For example, try an "elevator" rule: At the end of the list, reverse direction rather than starting back at the beginning.
- Keep completed tasks showing on the list until they're specifically cleaned out (but never make them current tasks).
- Manage a separate list that holds deleted tasks.
- Gather statistics about tasks (e.g., number of tasks completed vs. skipped, average time from task creation to completion).

Describe what qualities of the existing script might make those changes difficult or easy. We're not asking you to design the changes—just assess which might be easy and which could be difficult and why. Better yet, do this exercise with a partner or workgroup and discuss the changes you would make. Go ahead, mark up the code. We'll wait.

All done? How easy was that? What vocabulary did you use? If you're anything like us, you'll now have notes about the code being "tangled" or even "monolithic"—still a little vague and not a great contribution toward an improvement plan.

The Language of Smells

To help with this difficulty, the agile community has developed an informal catalog of the most common ways in which software can render change difficult. These problems are often called "code smells," after Kent Beck's analogy between code and babies—"If it stinks, change it" [1]. Most smells are either problems of poor communication (because it's hard to change code you don't understand) or duplication (which more than doubles the risk of making the changes).

Each code smell has an indicative name, a set of tell-tale symptoms so you can spot it easily, an indication of what refactorings you can do to remove the smell, and an indication of what other smells you might want to look for next. We've outlined a few smells in the sidebar.

Exercise, Part 2

Using the smell descriptions in the sidebar as a guide, revisit the to-do list script in figure 1 and identify any smells you can. Again, work in a discussion group if you are able. Come back when you're done.

How was the exercise this time? At this point, the smells catalog generally provides a number of benefits. First, our search for problems was more focused because we knew what symptoms to look for. Second, we were

1. ***Rename Variable***—changed `t` to `task_list`
2. ***Rename Variable***—changed `skp` to `skip_counts`
3. ***Rename Variable***—changed `c` to `current_task_index`
4. ***Extract Method***—created a `next_line` method that prompts the user and returns a String of input text; used in two places in the script
5. ***Extract Class***—created a `TodoList` class; assigned a singleton instance to a constant
6. ***Move Field***—moved `task_list` to be a field `@task_list` on `TodoList`
7. ***Move Field***—moved `skip_counts` to be a field `@skip_counts` on `TodoList`
8. ***Move Code***—moved the code that parses the arguments into the branch for the "todo" command
9. ***Extract Method***—created `append_task`, called from the "todo" branch of the case
10. ***Move Method***—made `append_task` a method on `TodoList`
11. ***Extract Method***—created `delete_task`, called from the "done" and "skip" branches of the case
12. ***Move Method***—made `delete_task` a method on `TodoList`
13. ***Extract Method***—created `length`, called from the "done" and "skip" branches of the case
14. ***Move Method***—made `length` a method on `TodoList`
15. ***Move Code***—moved the wrap-around check on `current_task_index` outside of the case statement
16. ***Move Field***—moved `current_task_index` to be a field `@current_task_index` on `TodoList`
17. ***Remove Parameter***—`TodoList.delete_task` can use `@current_task_index` instead of taking a parameter
18. ***Extract Method***—created `current_task_description`, called from the "skip" and "list" branches of the case
19. ***Move Method***—made `current_task_description` a method on `TodoList`
20. ***Extract Method***—created `skip_current` to hold the body of the "skip" branch of the case
21. ***Move Method***—made `skip_current` a method on `TodoList`
22. ***Replace Conditional with Guard Clause***—inverted the check at the top of `skip_current` in order to reduce the nesting levels in that method
23. ***Extract Method***—`created check_for_current_overflow`, called from `TodoList.delete_task` and `TodoList.skip_current`
24. ***Extract Surrounding Method***—created private method `TodoList.edit_list`, which yields to a supplied block only if `@task_list` is non-empty, and then wraps `@current_task_index` around to the top afterwards if necessary; called from `TodoList.skip_current` and `TodoList.delete_current_task`
25. ***Inline Method***—inlined `TodoList.check_for_current_overflow` into its only caller, `TodoList.edit_list`
26. ***Remove Method***—deleted the public getters and setters for `@skip_counts` and `@current_task_index` as these fields are no longer used outside of the `TodoList` class
27. ***Extract Method***—created a `to_s` method to return a String listing the tasks
28. ***Move Method***—moved `to_s` into `TodoList`
29. ***Remove Method***—deleted the public getter and setter for `@task_list` as this field is no longer used outside of the `TodoList` class
30. ***Extract Method***—created a `empty?` method to indicate whether the list has any tasks; called from two places in the main loop
31. ***Move Method***—made `empty?` a method on `TodoList`
32. ***Remove Method***—`TodoList.length` is no longer needed
33. ***Extract Method***—created an `execute(command, arg)` method containing the whole of the case statement; returns true to continue, or false if the user wants to quit

Figure 3: Refactoring change log

able to be more precise about the code's problems and where they occurred. And finally, we had the beginnings of an action plan—a list of things to do that might help this code become more adaptable to future change.

Here are some problems we found with the to-do list script:

Uncommunicative Names—`t`, `skp`, and `c` are too short to communicate their roles.

Comments—The code isn't self-explanatory enough.

Long Method—The whole thing is written as one big script; some code is nested four levels deep (`while / case / if / if`).

Magic Number—"3" is the number of times a task is skipped before it's deleted. English text appears in strings and patterns (limiting internationalization).

Duplicated Code—Sometimes we have literal duplication (like the prompt and line fetching code or the way tasks are deleted). Other times, it's more subtle, such as the two ways used to wrap around when we hit the end of the list (explicitly checking and setting to zero, or using the modulus operation).

Greedy Module—The user interface (such as it is) is

mixed up with the business logic, even though these two aspects of the code will change in different directions at different times in response to different forces.

Open Secret—A task is maintained as a string, and the list is maintained as a pair of arrays and a counter. Note how the two arrays are kept parallel.

Feature Envy—Chunks of the code want to "live with" the tasks list.

Note that by naming the smells, we have made them more tangible and more precisely located the code's problems. Now, by collecting the various "What to Do" notes from the smells we found, we can begin to select a few first refactoring actions. In order to address the smells we found, we'd likely make the names longer and more descriptive of the program's design, pull out a `TodoList` class, and perhaps pull out a `Task` class.

This plan derives directly from knowledge of the code smells. The smells catalog has helped us review the code, communicate clearly about its problems, and form an action plan for refactoring it. See figure 3 for a log of the detailed changes made during our most recent run at tidying up the basic smells in this code.

After these steps, the code is simple and clean. The mechanics of dealing with the list are hidden inside the `TodoList` class, which has no public fields, and those mechanics are nicely separated from the tool's user interface. We never found a need to move to the third step of the plan and extract a Task class; you could do that yourself as an exercise.

As a footnote to the refactoring log: Ten of the thirty-three steps used Extract Method, and twenty-two of the thirty-three were operations on whole methods. Methods are fundamental units of code reuse, which means that the remedies for most code smells involve creating or manipulating methods. We performed thirty refactorings on methods in order to extract and encapsulate one class. While this seems like a lot, many were quite small steps, and all followed the basic recipe set out in *Refactoring in Ruby* for fixing the Greedy Module smell.

Conclusion

Code smells form a vocabulary for discussing code quality, and, hence, for describing how well suited code might be to change. The smells also provide good indications as to what to refactor and how. Learn the language of code smells to get started on refactoring—the road to faster development. **{end}**

william.wake@acm.org
kevin@rutherford-software.com

Sticky Notes

For more on the following topics, go to www.StickyMinds.com/bettersoftware.

- `Todo.rb` and demonstration tests
- References