Practicing Horizontal Refactoring

Overview
- Iterative application of Flocking Rules to remove special cases in the "99 Bottles" song verses.
- Result: A single abstract template producing every possible verse.
- Focus on replacing differences with common abstractions.

Replacing Difference With Sameness
- **Initial Case Analysis**
  - Verse 0, 1, and else cases differ in several ways.
  - 1 and else cases are more alike than 0 and either.
- **Refactoring Steps**
  - Replace hard-coded numbers with variable references (e.g., replace "1" with "#{number}").
  - Abstract differences like "bottle" vs "bottles" into a container method.
  - Make phrases identical by substituting differing strings with method calls.

Equivocating About Names
- **Naming Challenges**
  - Naming abstractions like container is straightforward.
  - Naming more generic concepts (e.g., "it" vs "one") is harder.
  - Strategies for naming when perfect names are elusive:
    - Spend limited time choosing a good enough name.
    - Use temporary meaningless names (e.g., foo) expecting later improvement.
    - Ask a knowledgeable person (a "name guru").
- **Example: pronoun Method**
  - Replace "it" and "one" with calls to pronoun(number).
  - pronoun returns "it" if number == 1, else "one".

Deriving Names From Responsibilities
- **Analyzing Phrase Four Differences**
  - Difference between "no more" and "#{number -1}".
  - Concept identified as quantity: string to be sung in place of any number.
  - Naming concepts based on their responsibilities aids clarity.
- **Choosing Meaningful Defaults**
  - Temporary defaults like :FIXME help in refactoring but require careful use.
  - For quantity, default argument must be meaningful (e.g., 0) to avoid errors.
  - Return values should be consistent in type to avoid sender-side conversions.

Consistency and the Liskov Substitution Principle
- **Importance of Consistent Method Shape**
  - Extracted methods quantity, container, and pronoun share similar structure:
    - Take a single argument.
    - Use a conditional testing equality.
  - Consistency improves understandability and facilitates further refactoring.
- **Liskov Substitution Principle Applied**
  - Methods should return values consistent in type.
  - Avoid forcing callers to handle different return types.
  - Refactor quantity to always return a string, ensuring it supports methods like capitalize.

Summary
- Application of Flocking Rules iteratively reveals hidden abstractions.
- Abstracting differences into meaningful methods improves code clarity despite increased lines.
- Consistency in method shape and return types supports maintainability and future refactorings.
- Abstractions reduce dependencies and isolate responsibilities.
- Refactoring produces code that multiple programmers would independently arrive at, differing mainly in naming.

Final Refactored Code Structure
- **verse(number) returns a template string using:**
  - quantity(number), container(number), action(number), and quantity(successor(number)), container(successor(number)).
- All conditionals from the original case statement removed.
- **Extracted methods:**
  - quantity(number) returns "no more" if 0, else number as string.
  - container(number) returns "bottle" if 1, else "bottles".
  - pronoun(number) returns "it" if 1, else "one".
  - action(number) returns "Go to the store and buy some more" if 0, else "Take #{pronoun(number)} down and pass it around".
  - successor(number) returns 99 if 0, else number – 1.

Discovering Deeper Abstractions
- **Final Differences in Phrase Four**
  - Difference between "99" and "#{quantity(number -1)}" reveals a new concept: successor.
  - successor(number) returns:
    - 99 if number == 0.
    - number - 1 otherwise.
  - Separates verse progression logic from quantity representation.
- **Proper Use of Abstractions**
  - Replace all uses of number –1 with successor(number) to depend on abstractions.
  - Results in identical 0 and else cases in verse method.

Taking Bigger Steps
- **Combining Refactoring Steps**
  - After practicing small steps, combine several steps into bigger changes.
  - Example: Create action(number) method to abstract third phrase differences:
    - Returns "Go to the store and buy some more" if number == 0.
    - Else returns "Take #{pronoun(number)} down and pass it around".
  - Such combined steps still follow the Flocking Rules but require tests to remain green.