Rediscovering Simplicity in Programming

The Nature of Complexity and Abstraction

- Growth in Programming Skills
  - Beginners write simple code; experience leads to complex solutions.
  - Complexity is seen as natural and inevitable.
- Focus Shift in Code Design
  - From understandability to changeability.
  - Code becomes more abstract and initially harder to understand.
- Object-Oriented Design (OOD)
  - Accepts complexity increase in some areas for benefits in others.
  - Trade-offs involve costs balanced by benefits.
  - Examples:
    - DRY principle reduces duplication but hides behavior details.
    - Splitting classes improves reuse but obscures relationships.
    - Dependency injection abstracts dependencies but hides their concrete classes.
- The Challenge of Abstractions
  - Picking the right abstractions is critical.
  - Wrong abstractions lead to convoluted, confusing code.
  - Early abstractions are often premature and incorrect.
  - Suggests resisting abstraction until absolutely necessary.

Simplifying Code

- Balancing Concrete and Abstract
  - Code must be concrete enough to understand and abstract enough to allow change.
  - Exists on a continuum from most concrete (long procedures) to most abstract (many small classes).
  - The best solutions lie in the middle, balancing comprehension and flexibility.
- Four Solutions to the "99 Bottles" Problem
  - Illustrate different points along the abstraction continuum.
  - Encourage readers to solve the problem before examining solutions.

Solution 1: Incomprehensibly Concise

- Characteristics
  - Very concise but hard to understand.
  - Embeds complex logic within strings.
- Problems
  - Inconsistent conditional styles.
  - Duplication of data and logic (e.g., pluralization).
  - Lack of meaningful names; hidden concepts not isolated.
- Impact
  - Difficult to read, understand, and maintain.
  - Low cost to write, but high cost to understand and change.
  - Code does not clearly expose domain concepts.
- Terminology Notes
  - Methods define behavior.
  - Sending messages invokes behavior.
  - Prefers "send a message" over "call a method" to emphasize abstraction.

Solution 2: Speculatively General

- Characteristics
  - Uses lambdas and Verse objects for each verse variant.
  - Attempts extensibility and abstraction.
- Domain Exposure
  - Four verse variants identified.
  - Clear which verses are alike or different.
  - Next verse rule buried in code.
- Problems
  - More code than necessary.
  - Complex indirection with unclear necessity.
  - Difficult to understand and change.
- Critique
  - Complexity not justified by benefits.
  - Overly clever code distracts and complicates.
  - Advises writing simpler code despite ability to create complex solutions.

Solution 3: Concretely Abstract

- Characteristics
  - Many small methods, DRY code.
  - Attempts to name domain concepts.
- Issues with Domain Clarity
  - Difficult to determine verse variants and differences.
  - Parts understandable individually but hard to see whole picture.
- Problems with Naming and Abstraction Levels
  - Method names tied to current implementation (e.g., "beer") rather than domain meaning.
  - Changing drink from "beer" to "Kool-Aid" requires many renames and changes.
  - Naming should reflect domain concepts (e.g., "beverage") to isolate implementation changes.
- Summary
  - DRY achieved but at cost of inappropriate abstraction and naming.
  - High cost to change despite DRY.
  - Some abstractions are wrong despite good intention.

Solution 4: Shameless Green

- Characteristics
  - Simple, straightforward code.
  - Uses case statements with duplicated strings.
  - Easy to understand and write.
- Domain Exposure
  - Clearly shows four verse variants.
  - Identifies which verses are alike and different.
  - Next verse rule still implicit.
- Value/Cost Evaluation
  - Easy to write and understand.
  - Cheap to change despite duplication.
  - Lacks abstraction and DRYness but is cost-effective.
- Conclusion
  - Best balance of understandability and changeability.
  - Embarrassingly duplicative but practical.
  - Good starting point before introducing abstractions.

Judging Code

- Subjectivity of Code Quality
  - Everyone has opinions but lacks universal, actionable definitions.
  - Definitions often qualitative, e.g., elegant, crisp, caring author.
- Need for Concrete Measures
  - Good code defined as highest value for lowest cost.
  - Metrics provide measurable facts beyond opinion.

Metrics for Evaluating Code

- Source Lines of Code (SLOC)
  - Measures size but imperfect predictor of quality or productivity.
  - Larger code not necessarily worse or better.
- Cyclomatic Complexity
  - Counts unique execution paths (conditionals).
  - Helps identify hard-to-test/maintain code.
  - Lower complexity preferred.
  - Limited focus on conditionals only.
- Assignments, Branches, and Conditions (ABC) Metric
  - Counts assignments, message sends (branches), and conditionals.
  - Reflects cognitive load and complexity.
  - Popular Ruby tool: Flog.
  - Helps identify complex code difficult to understand and maintain.

Comparing the Four Solutions with Metrics

- Metrics Summary Table
  - Incomprehensibly Concise: 19 SLOC, 15 conditionals, ABC 15.
  - Speculatively General: 63 SLOC, 6 assignments, 10 branches, 4 conditionals, ABC 12.
  - Concretely Abstract: 87 SLOC, 4 assignments, 26 branches, 9 conditionals, ABC 28.
  - Shameless Green: 34 SLOC, 4 conditionals, ABC 4.
- Insights
  - Shameless Green has lowest ABC score, indicating simplest cognitive load.
  - Incomprehensibly Concise has very high conditionals packed in few lines.
  - Concretely Abstract spreads complexity across many calls and lines.
  - Larger SLOC loosely correlates with higher ABC but not always.

Summary and Recommendations

- Programmers' Growth and Complexity
  - Experience leads to comfort with complexity, but simplicity is possible.
  - Aim for blindingly simple code rather than infinitely complex.
- Shameless Green as Best Starting Point
  - Prioritizes understandability over premature abstraction.
  - Uses tests to build understanding before abstracting.
  - Accepts temporary duplication to avoid wrong abstractions.
- Practical Advice
  - Write simple, understandable code first.
  - Avoid premature abstraction.
  - Use metrics to guide complexity and maintainability.
  - Refactor when necessary based on real change demands.