Refactoring code and scaling up

1 Code Complexity

Metrics

- Lines of code
  - Easy to count and automate
  - Focuses on implementation
  - Counts vary between language
  - Can encourage counter-productive coding practices
  - More a measure of effort rather than complexity
  - Remember that readability counts
- Cyclomatic complexity
  - Treats the program as a graph
  - Measures how many paths there are through your code
  - Node(N), actions
  - Edge(E), program flow between actions
  - P is the number of connected components (for a single piece of code P is equal to 1)

$$C = E - N + 2P$$

$1 - 10$	Easy program, low risk
$11 - 20$	Complex program, tolerable risk
$21 - 50$	Complex program, high risk
50+	Impossible to test

- Halstead metrics
  - Relates to the size of the codebase
    * n1 = Number of distinct operators
    * n2 = Number of distinct operands
    * N1 = Total number of occurrences of operators
    * N2 = Total number of occurrences of operands
  - Halstead volume

$$V = N \times \log_2(n)$$

    Where:
    * N=N1+N2
    * n=n1+n2
- Maintainability index
  - A combination of the previous metrics

$$MI = 171 - 5.2\ln(V) - 0.23(C) - 16.2\ln(LoC) + 50\sin\sqrt{2.4CM}$$

    Where
    * V = Halstead Volume
    * C = Cyclomatic Complexity
    * LoC = Lines of code
    * CM = average percentage of comment lines
  - High MI (> 75) - Good maintainability
  - Low MI (< 25) - Code restructuring necessary

2 Complexity and Refactoring

Why refactor?

- Greater predictability - plan for amount of maintenance required
- Software Risk Mitigation - Can assess and mitigate the risks of introducing bugs into existing code
- Reduced costs - Keeping the software relatively simple can reduce financial overhead in terms of maintenance
- Extended value - reducing complexity means that the software will retain value for longer
- Decision support - having the data to hand to help make a retirement decision (cost of re-write vs new code)

3 Refactoring

Refactoring is changing a software system by improving its internal structure without changing its external behaviour

- When refactoring you should not add functionality
- Refactoring improves the understandability of the software and so reduces the need for documentation
- Changes are easier to make because the code is well structured and clear

Refactoring improves the code to slow down the degradation of the code through change

3.1 Where refactoring helps

- Code smell - a surface indication that there might be a deeper problem in the system, a subjective assessment
- Unhealthy dependencies between classes and packages
- Duplicate code - similar code may be included at different places in a program
- Long methods
- Data clumping - same group of data items reoccur in different places
- Untidy code and poorly named attributes and methods

3.2 Why refactor?

- Improves the design of software
- Easier to understand
- To find the bugs
- To program faster

3.3 Resistance to refactoring

Maintenance is hard:

- Programmers don't want to touch an existing codebase
- Often hard to follow another programmers code
- Code maintenance is boring so avoided if possible

Stakeholders

- Project manager - It doesn't add any new desired functionality, nor does it fix any bugs
- Senior developers - Too much investment in the system and its easier if nothing changes
- Junior developers - Don't have enough experience and not fun or leads to promotion

3.4 When to refactor

- Best to do it continuously, but if not possible, then set an MI boundary that you and your team wish to maintain