Software Design Goals

CMPT 145

Learning Objectives

After studying this chapter, a student should be able to:

- Define design goals of correctness, and efficiency.
- Define implementation goals of robustness, adaptability, and reusability.
- Assess, at a preliminary level, the quality of example code with respect to the design and implementation goals.

Design Goals

Quantitative goals that can be scientifically measured:

- Correctness
  - Software does everything it should do
  - Does nothing it shouldn't do
- Efficiency
  - Relative measure of resource consumption
  - We aim for effective use of resources (time, space=memory)

Implementation Goals

Qualitative goals that can't be scientifically measured:

- Robustness
  - Software behaves well when something unexpected happens
- Adaptability
  - Small changes in behaviour require only small changes in code
- Reusability
  - Software can be used more than once

Example 1

Let's review our code solution to the Sieve of Eratosthenes problem.

(a) What can we say about our design & implementation goals?
- Correctness
- Efficiency
- Robustness
- Adaptability
- Reusability

(b) How can we improve the code?

Counting Primes: Script 2

```
n = 20  # end of range of numbers to check for primes

still_is_prime = (n+1)*[True] # assume prime until disproven

for i in range(2, n):
    if still_is_prime[i]:
        # mark multiples of i as not prime
        j = 2*i
        while j <= n:
            still_is_prime[j] = False
            j += i

# now, every possible prime is a definite prime
count = sum([1 for v in still_is_prime[2:] if v])

print("# Prime numbers between 2 and " + str(n) + ":", count)
```

Counting Primes: Code Review

It satisfies some design and implementation goals.

- Correctness:
  - Displays the number of primes between two and n without doing anything incorrectly.
- Efficiency:
  - For n in the range needed for cryptography, the list is quite large, and processing it takes some time.
  - Note: Script 2 is better than Script 1, because the lists saves time! But in this problem, even using a list to save time only goes so far for large n.

Counting Primes: Code Review

- Robustness:
  - Very little room for unexpected behaviour.
  - However, there is no warning in the code about negative `n`.
- Adaptability:
  - Easy to edit `n`.
  - However, it might be better for the script to ask the user for n, to avoid editing.
- Reusability:
  - The only way to reuse this code is copy/paste, which is terrible.

Counting Primes: Code Improvement

- Efficiency: There are strategies to use less memory for large n. Google it!
- Robustness: Add a warning in the comments about negative n.
- Reusability: Encapsulate the counting code within a function that returns the count. Then add a check for negative n.

Example 2

Let's review our code solution to the Gambler's Ruin problem.

(a) What can we say about our design & implementation goals?
- Correctness
- Efficiency
- Robustness
- Adaptability
- Reusability

(b) How can we improve the code?

Example 3

Let's review our code solution to the Coupon Collector's problem.

(a) What can we say about our design & implementation goals?
- Correctness
- Efficiency
- Robustness
- Adaptability
- Reusability

(b) How can we improve the code?

Example 4

Let's review our code solution to the Self-Avoiding Random Walks problem.

(a) What can we say about our design & implementation goals?
- Correctness
- Efficiency
- Robustness
- Adaptability
- Reusability

(b) How can we improve the code?