COMP1531

6.1 - Software Engineering Design Principles

Miscellaneous

- Exam replacement
- Project/test structure
- Testing for HTTP errors
- Timers

Design Smells

- **Rigidity**: Tendency to be too difficult to change
- **Fragility**: Tendency for software to break when single change is made
- **Immobility**: Previous work is hard to reuse or move
- **Viscosity**: Changes feel very slow to implement
- **Opacity**: Difficult to understand
- **Needless complexity**: Things done more complex than they should be
- **Needless repetition:** Lack of unified structures
- **Coupling:** Interdependence between components

Design Principles

Purpose is to make items:

- Extensible
- Reusable
- Maintainable
- Understandable
- Testable

Often, this is achieved through **abstraction**. Abstraction is the process of removing characteristics of something to reduce it some a more high level concept

DRY

"Don't repeat yourself" (DRY) is about reducing repetition in code. The same code/configuration should ideally not be written in multiple places.

Defined as:

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system"

DRY

How can we clean this up?

```
 1 import sys
 2
 3 if len(sys.argv) != 2:
 4         sys.exit(1)
 5
 6 num = int(sys.argv[1])
 7
 8 if num == 2:
 9         for i in range(10, 20):
10                 result = i ** 2
11                 print("i ** 2 = {result}")
12
13 elif num == 3:
14         for i in range(10, 20):
15                 result = i ** 3
16                 print("i ** 3 = {result}")
17
18 else:
19         sys.exit(1)
```

DRY

How can we improve this?

```
1 import jwt
2
3 encoded_jwt = jwt.encode({'some': 'payload'}, 'applepineappleorange', algorithm='HS256')
4 print(jwt.decode(encoded_jwt, 'applepineappleorange', algorithms=['HS256']))
```

DRY

What about the name server example?

What if we wanted to count how many times the name has been changed?

KISS

"Keep it Simple, Stupid" (KISS) principles state that a software system works best when things are kept simple. It is the believe that complexity and errors are correlated.

Your aim should often be to use the simplest tools to solve a problem in the simplest way.

KISS

Example 1: Write a python function to generate a random number with up to 50 characters that consist of lowercase and uppercase characters

KISS

Example 2: Write a function that prints what day of the week it is today

https://stackoverflow.com/questions/9847213/how-do-i-get-the-day-of-week-given-a-date-in-python

KISS

Example 3: Handling command line arguments

```
1 python3 commit.py -m "Message"
2 python3 commit.py -am "All messages"
```

Encapsulation

Encapsulation: Maintaining type abstraction by restricting direct access to internal representation of types (types include classes)

Encapsulation

Example:

```
1 class Point:
2     def __init__(self, x,y):
3         self.x = x
4         self.y = y
5
6 def distance(start, end):
7     return sqrt((end.x - start.x)**2 + (end.y -
```

What if we wanted to store points in polar coordinates?

Encapsulation

Example:

```
1 class Queue:
2     def __init__(self):
3         self.entries = []
4
5     def enqueue(self, entry):
6         self.entries.append(entry)
7
8     def dequeue(self):
9         return self.entries.pop(0)
```

Can we prevent stealing spots in the queue?

Top-down thinking

Also commonly known as "You aren't gonna need it" (YAGNI) that says a programmer should not add functionality until it is needed.

Top-down thinking says that when building capabilities, we should work from high levels of abstraction down to lower levels of abstraction.

Top-down thinking

Question 1: Given two Latitude/Longitude coordinates, find out what time I would arrive at my destination if I left now. Assume I travel at the local country's highway speed

Why is well designed software important?

- When you only do this loop once, writing bad code has minimal impacts
- When we complete this "cycle" many times, modifying bad code comes at a high cost

Why is well designed software important?

"Poor software quality costs more than $500 billion per year worldwide" – Casper Jones

Systems Sciences Institute at IBM found that it costs four- to five-times as much to fix a software bug after release, rather than during the design process

Why do we write bad code?

Often, our default tendency is to write bad code. Why?

- It's quicker not to think too much about things
  - Good code requires thinking not just about now, but also the future
- Pressure from business we're looking for
- Refactoring takes time

Bad code: Easy short term, hard long term

Good code: Hard short term, easy long term

Why do we want to write **good** code?

- More consistent with Agile Manifesto
  - "Welcome changing requirements"
- Adapt easier to the natural SD life cycle

Refactoring

Restructuring existing code *without* changing its external behaviour.

Typically this is to fix code or design smells and thus make code more *maintainable*

Finding a balance

- Don't over-optimise to remove design smells
- Don't apply principles when there are no design smells - unconditional conforming to a principle is a bad idea, and can sometimes add complexity back in