COMP1531

7.1 - Software Engineering Design Principles

Updates

- Lab06 due date extended to Tuesday 5th November
- Help sessions will be added for Friday + multiple in week 8 & 9

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.

Why?

- Takes up space with source code
- Makes your code break when change is made

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)
```

https://dbader.org/blog/python-first-class-functions

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']))
```

What about a config file?

DRY

What do we think about this?

```
1  from flask import Flask
2  app = Flask(__name__)
3
4  authStr = "auth"
5  @app.route(f"/{authStr}/register")
6  def register():
7      return "Hello World!"
8
9  @app.route(f"/{authStr}/login")
10 def login():
11     return "Welcome back!"
12
13 if __name__ == "__main__":
14     app.run()
```

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

```
1 def randomGenerate(elems):
2     pass
```

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: Create your own git commit command

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

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

Encapsulation

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

Encapsulation

Example:

1. Create a file that stores an x and y coordinate
2. Convert that file to a class type with two fields
3. Encapsulate those fields for abstraction reasons
4. Modify the internal representation to polar form

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

2) Example of starting project
3) Find my best timetable
- Pro: Prevents creating pointless capabilities
- Con: Sometimes initially creates pointless separation

Top-down thinking

Question 2: Determine my 20T1 UNSW timetable with the minimal amount of days spend at UNSW.

Top-down thinking

The main thing to be careful with is sometimes this approach can add overly complex abstractions. Often, refactoring is useful to undertake afterward.

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

Iteration 3

Iteration 3 has been released

More front-end additions are coming in week 8

Iteration 3 \ Exceptions

Error format for the frontend

If you return the following from a route, the frontend will successfully display the error as intended.

```
dumps({
    "code": 400,
    "name": "ValueError",
    "message": "This is the text displayed",
})
```

Iteration 3 \ Refactoring

Code refactoring: Process of restructuring existing code without changing it's external behaviour - often with the purpose of making it easier to understand and simpler to modify

Often, this appears as trying to remove **design smells** in order to employ good **design principles**.

A huge part of iteration 3 will be refactoring.

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

Next Monday

- Project structures and importing
- Help with timers
- Help with file uploads