COMP1531

🐶 Software Engineering
5.1 - Design - Writing good software

In this lecture

Why?

- Writing good software makes your team members happy, and makes your code less likely to break

What?

- What makes good software
- Elements of good design
- Elements of clean code

Writing good software

Generally speaking, we can consider software well written if:

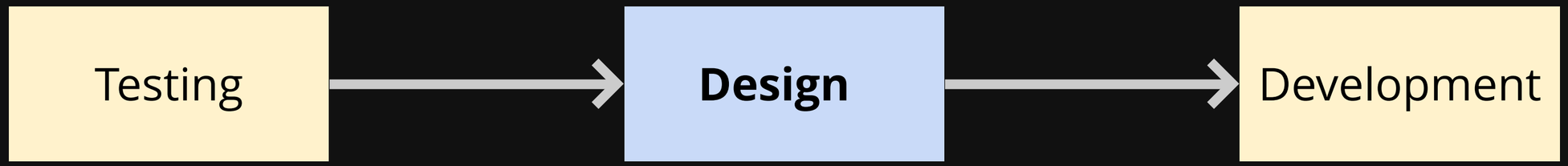
- (Testing) It's correctness is verifiable in an automated way
- (Design) It is planned, modular, and resistant to breaking changes
- (Development) It is clean and easy to work with

This lecture focuses on the design and development aspects of good software.

Well designed software

Something happens between writing tests and finishing code: **Design**

The design of software happens when you know what problem you're trying to solve with code, but want to think about the best way to solve the problem before you finish the code.



Design Principles

Purpose is to make things:

- Reusable / Modular
- Maintainable
- Robust to changes

Design approaches that do not make things better are called **design smells**

Generally speaking, well designed code is simple, clear, and **resists the tendency to break as the software changes or grows.**

This is more critical than ever with the rapidly iterative nature of modern web development.

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

Where is "design"?

The term **design** is thrown around very broadly. But we can think of it in two different lens:

1. **Before coding**: Design means thoughtfulness in planning

2. **During coding**: Design means robustness in coding

Design: Thoughtful planning

Thinking hard before you code is a great tactic to ensure that your time coding is efficient and that you carry useful documentation.

Examples include:

- Writing pseudocode
- Flow diagrams
- Component diagrams
- State diagrams
- etc

We cover *some* of this later in the course

Design: Robust Coding

Just because you plan software well, it doesn't mean when you come to write it that it will turn out well designed in it's execution.

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 working for

Bad code: Easy short term, hard long term

Good code: Hard short term, easy long term

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"

Why do we care?

When you repeat yourself less, a change in one module is unlikely to break entire systems

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(f"{i} ** 2 = {result}")
12
13 elif num == 3:
14         for i in range(10, 20):
15                 result = i ** 3
16                 print(f"{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']))
```

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.

"Every line of code you don't write is bug free"

Why do we care?

- Complicating things with boutique solutions means more code to maintain (more likely to break)
- More code written means more code to test

KISS

Example 1: Write a python function to generate a random string 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"
```

Minimal Coupling

Coupling is the degree of interdependence between software components.

The more software components are connected, the more changes and alterations to one component may break another (either at compile time or runtime)

Excessive coupling can lead to **spaghetti code**.

Top-down thinking

Similar to "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.

Why do we care?

Removes unnecessary code, less to maintain, less likely to cause problems

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

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

Well developed software

Software can be correct (tested), well designed, but still written like absolute garbage. We want to write nice code, because:

1. It's easier for future you to read and understand
2. It's easier for others to read and understand
3. It's easier to find and spot errors now and in future

"Clean code" is often a **language specific** topic.

- Some things we talk about are universal
- Some things only work in a handful of languages
- Some things apply to only python specifically

What is clean code?

Clean code is generally defined by:

- Being as simple as possible
- Following standard & understood conventions

Being Pythonic

On the topic of following standard conventions: Within python, being "**Pythonic**" means that your code generally follows a set of idioms agreed upon by the broader python community.

"When a veteran Python developer calls portions of code not “Pythonic”, they usually mean that these lines of code do not follow the common guidelines and fail to express its intent in what is considered the most readable way. On some border cases, no best way has been agreed upon on how to express an intent in Python code, but these cases are rare."

Hitchhiker's guide to python (read more on this)

Some of these include destructuring, enumerate

Docstrings

Docstrings are an important way to document code and make clear to other programmers the intent and meaning behind what you're writing. We are somewhat different on the formatting, but we want it to include 1) Description, 2) Parameters, 3) Returns

docstring.py

```
1  def string_find(str1, str2):
2      """ Returns whether str2 can be found within str1
3
4      Parameters:
5          str1 (str): The haystack
6          str2 (str): The needle
7
8      Returns:
9          (bool): Whether or not str2 could be found in str1
10
11     """
```

Map, Reduce, Filter

Map, reduce, filter are functions that help as accomplish basic iterative tasks without the overhead of a loop setup

- **Map**: creates a new list with the results of calling a provided function on every element in the given list
- **Reduce**: executes a **reducer** function (that you provide) on each member of the array resulting in a single output value
- **Filter**: creates a new array with all elements that pass the test implemented by the provided function

Map

Map: creates a new array with the results of calling a provided function on every element in the calling array

map.py

```
1 def shout(string):
2     return string.upper() + "!!!!"
3
4 if __name__ == '__main__':
5     tutors = ['Simon', 'Teresa', 'Kaiqi', 'Michelle']
6     angry_tutors = list(map(shout, tutors))
7     print(angry_tutors)
```

Filter

Filter: creates a new array with all elements that pass the test implemented by the provided function

filter.py

```
from functools import reduce

if __name__ == '__main__':
    marks = [ 65, 72, 81, 40, 56 ]
    passing_marks = list(filter(lambda m: m >= 50, marks))
    total = reduce(lambda a, b: a + b, passing_marks)
    average = total/len(passing_marks)
    print(average)
```

Reduce

Reduce: executes a **reducer** function (that you provide) on each member of the array resulting in a single output value

reduce.py

```python
from functools import reduce

def custom_sum(first, second):
    return first + second

if __name__ == '__main__':
    studentMarks = [ 55, 43, 34, 23, 22, 10, 44 ]
    total = reduce(lambda a, b: a + b, studentMarks)
    print(total)
```

Combined

allthree.py

```
1 from functools import reduce
2
3 if __name__ == '__main__':
4     marks = [ 39, 43.2, 48.6, 24, 33.6 ] # Marks out of 60
5     normalised_marks = map(lambda m: 100*m/60, marks)
6     passing_marks = list(filter(lambda m: m >= 50, normalised_marks))
7     total = reduce(lambda a, b: a + b, passing_marks)
8     average = total/len(passing_marks)
9     print(average)
```

Exceptions > Error Codes

C-style programming follows a principle of methodical process of using return values to denote particular errors. Whilst this makes programs more easy to reason with, it convolutes codea nd makes it hard to understand. For this reason, we prefer exceptions.

early.py

```
1 def sqrt(num):
2     if num < 0:
3         return None
4     return num ** 0.5
5
6 myNum = int(input())
7 if sqrt(myNum) is not None:
8     print(sqrt(myNum))
```

The problems though are:

- Often we can only use "None" or some arbitrary return (-1) to signify that it didn't work
- It's harder to check for a client using it

Exceptions > Early Returns

Using exceptions. And we can make our own.

early.py

```
1  class SqrtException(Exception):
2      pass
3
4  def sqrt(num):
5      if num < 0:
6          raise SqrtException("Number cannot be < 0")
7      return num ** 0.5
8
9  try:
10     print(sqrt(int(input())))
11 except SqrtException as e:
12     print(e)
```

Multi-line strings

Someones strings need to exist over multiple lines,
there are two good approaches for this

multiline.py

```
1  if __name__ == '__main__':
2      text1 = """hi
3
4      this has lots of space
5
6      between chunks"""
7
8      text2 = (
9          "This is how you can break strings "
10         "into multiple lines "
11         "without needing to combine them manually"
12     )
13
14     print(text1)
15     print(text2)
```

Meaningful Abstractions

Programmers will often go through 3 stages:

- (Bad) Not appreciating abstraction
- **(Bad) Over-appreciating abstraction**
  - **Creating benign-abstractions**
- (Good) Appreciating abstraction appropriately

benign.py

```
1 from datetime import datetime
2
3 def dateNow():
4   return datetime.now()
5
6 if __name__ == '__main__':
7   print(dateNow())
```

Feedback