Part 2 - Introduction

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Main topics of the second part of the course

- Coding Guidelines and Best Practices
- Modularity
- Introduction to OOP
- Design Patterns
- Introduction to Web applications

No lecture on:

- Oct. 31
- Nov. 01
- Nov. 18
- Nov. 28

Coding Guidelines and Best Practices

Lesson outline

- Software quality and requirements
- Coding best (an bad) practices
- Code refactoring
- Error handling

Coding Guidelines and Best Practices

But first, what is software *quality*?

Code requirements

- **Functional requirements**, *i.e.*, the code works correctly and conforms to other specifications (reliability, efficiency, security, etc.)
- **Non-functional requirements**, *i.e.*, the code is:
  - Understandable
  - Mantainable
  - Testable
  - Extendable
  - Reusable
  - etc.

We will now review some guidelines and best practices to help satisfy non-functional requirements.

Some (coding) Best Practice

KISS: "Keep It Simple, Stupid!" (but also "Keep It Short and Simple")

> *"Simple is better than complex. Complex is better than complicated."*
> *The Zen of Python*

Some (coding) Best Practice

- Follow coding conventions (*e.g.*, Python PEP 8)
  - they might include rules/guidelines for code organization, formatting, naming conventions, etc.
- Add the documentation and keep it updated
  - but remember: the documentation is for users, not for developers
- Use code formatters and linters
  - they automatically check (and possibly fix) formatting, small errors and bugs, etc.

Some (coding) Best Practice

- avoid '**magic numbers**' - use identifiers with meaningful names

Using identifiers (variables or constants) in our code, as opposed to using raw numbers (such as PI instead of 3.14 or `working_hours_for_day` instead of 8) allows you to change the value easily, and provides immediate clarity on the meaning behind each value

Some (coding) Best Practice

- **DRY**: don't repeat yourself! If there are duplicate parts of the code, we can put them together.

Over time, two identical blocks of code may naturally diverge.
When we fix a bug, there is a tangible risk of unintentionally fixing it in one block while ignoring the other.
This risk extends to making code improvements.

By consolidating the code into a single location, we ensure that corrections or enhancements need only be applied once.

Some (coding) Best Practice

- **Anticipation of change**. Change is inevitable in software systems.
  - user requirements may not be fully understood in the initial phase of the project
  - customer needs new functionality
  - environment changes
  - we must improve the software because we have to beat the competitors.

- We need to identify
  - changes that will probably happen in the near future
  - plan for change

Some (coding) Best Practice

SoC: Separation of Concerns (modularity)

- Avoid monolithic code
- Divide problems into smaller separate sub-problems and assign them to distinct sections, each one with a single responsibility

We will return to this point later

A simple example

FIZZ BUZZ

Write a function that takes an integer as input and prints:

- FIZZ if the integer is multiple of 3
- BUZZ if the integer is multiple of 5
- FIZZBUZZ if the integer is multiple of both 3 and 5

otherwise, print the integer itself

```python
# fizzbuzz_01.py


def fb(i):
  if (i % 3 == 0) and (i % 5 == 0):
    print("fizzbuzz")
  elif i % 3 == 0:
    print("fizz")
  elif i % 5 == 0:
    print("buzz")
  else:
    print(i)


for i in range(10):
  fb(i)
```

```python
def fb(i):
  if (i % 3 == 0) and (i % 5 == 0):
    print("fizzbuzz")
  elif i % 3 == 0:
    print("fizz")
  elif i % 5 == 0:
    print("buzz")
  else:
    print(i)


for i in range(10):
  fb(i)
```

Let's see what changes can be made.

1. Eliminate magic numbers.

```python
def fb(i):
  if (i % 3 == 0) and (i % 5 == 0):
    print("fizzbuzz")
  elif i % 3 == 0:
    print("fizz")
  elif i % 5 == 0:
    print("buzz")
  else:
    print(i)


for i in range(10):
  fb(i)
```

2. The same conditions are evaluated more than once (**DRY**).
   In this case, the conditions are trivial, but if the condition is complex to calculate or requires access to the internet or a database, it could take a considerable amount of time. By calculating it twice, we waste resources. Moreover, if it's complex, it might contain bugs or be open to improvement. Another reason to apply DRY.

```python
def fb(i):
  if (i % 3 == 0) and (i % 5 == 0):
    print("fizzbuzz")
  elif i % 3 == 0:
    print("fizz")
  elif i % 5 == 0:
    print("buzz")
  else:
    print(i)


for i in range(10):
  fb(i)
```

3. The function has **too many responsibilities**.
   It has to CALCULATE the condition and USE the condition. If the output is wrong, we don't know if the responsibility lies with the CALCULATION of the condition or the USE of the condition. Let's separate these two responsibilities.

Of course, this is an exaggeration; the function is so simple that it can be left as is. However, I wanted to use a simple example to emphasize the concept.

```python
def fb(i):
  if (i % 3 == 0) and (i % 5 == 0):
    print("fizzbuzz")
  elif i % 3 == 0:
    print("fizz")
  elif i % 5 == 0:
    print("buzz")
  else:
    print(i)


for i in range(10):
  fb(i)
```

4. Finally, **Anticipation of change**.
   The function might evaluate other divisors beyond 3 and 5. We can also generalize it further and consider a list of conditions, not just two

Final code

```python
# fizzbuzz_01.py


def is_multiple_of(n, d):
  return n % d == 0


def fb(i, div1=3, div2=5):
  cond1 = is_multiple_of(i, div1)
  cond2 = is_multiple_of(i, div2)
  if cond1 and cond2:
    print("fizzbuzz")
  elif cond1:
    print("fizz")
  elif cond2:
    print("buzz")
  else:
    print(i)
```

```python
div1 = 3
div2 = 5
for i in range(10):
  fb(i, div1, div2)
```

Code refactoring

Refactoring is the process of modifying the structure of the source code without altering its functionality

- The main objective of refactoring is to comply with non-functional requirements
- For instance, the code might be:
  - cleaned and reformatted
  - simplified
  - decomposed
- Refactoring should be performed in *micro-steps*
  - ideally, tests should be set up **before** to start with refactoring
- There are several tools that automate refactoring (IDEs, LLMs)

Error handling

- You should carefully handle errors and exceptions.

Consider the solution of the previous exercise, where parameters and inputs are given by the user. What happens if invalid inputs are given to the function?

```python
# fizzbuzz_03.py


def is_multiple_of(n, d):
  return n % d == 0


def fb(i, div1=3, div2=5):
  cond1 = is_multiple_of(i, div1)
  cond2 = is_multiple_of(i, div2)
  if cond1 and cond2:
    print("fizzbuzz")
  elif cond1:
    print("fizz")
  elif cond2:
    print("buzz")
  else:
    print(i)
```

```python
while True:
  div1 = int(input("Value for div1: "))
  div2 = int(input("Value for div2: "))
  i = int(input("Input value: "))
  fb(i, div1, div2)
```

Let's try to avoid crashes by preventing two common cases: division by zero and wrong input type.

```python
# fizzbuzz_04.py


def is_multiple_of(n, d):
  return n % d == 0


def fb(i, div1=3, div2=5):
  cond1 = is_multiple_of(i, div1)
  cond2 = is_multiple_of(i, div2)
  if cond1 and cond2:
    print("fizzbuzz")
  elif cond1:
    print("fizz")
  elif cond2:
    print("buzz")
  else:
    print(i)
```

```python
while True:
  div1 = input("Value for div1: ")
  div2 = input("Value for div2: ")
  i = input("Input value: ")

  if not div1.isdigit() or not div2.isdigit() \
    or not i.isdigit():
    print("Error: some input is not valid.
           Please pass only integers.")
    continue

  div1, div2, i = int(div1), int(div2), int(i)

  if div1 == 0 or div2 == 0:
    print("Error: you passed zero for a divisor.")
    continue

  fb(i, div1, div2)
```

This is the **Look before you leap (LBYL)** approach. This coding style explicitly tests for pre-conditions before making calls or lookups.

- it works well if you have to handle frequent *known* conditions
- useless for unexpected exceptions
- it might make the code bulky and hard to read (and mantain)
- in multi-threaded environments, concurrency issues might affect the evaluations of if statements

Try - Except

Python (and many other languages) implements a syntax construct to easily handle exceptions.

```
try:

  # code to be executed and may lead to exceptions

except SomeException:

  # code to be executed if `SomeException` is raised
```

Exceptions can be catched in several ways

```python
try:
  # code to be executed and may lead to exceptions

except ExceptionOne:
  # executed if `ExceptionOne` is raised

except ExceptionTwo:
  # executed if `ExceptionTwo` is raised

except (ExceptionThree, ExceptionFour, ExceptionFive):
  # executed if one of the specified exceptions is raised

except Exception:
  # catches any exception
  #  (every exception is a subclass of `Exception`)
```

Other useful features

```python
try:
    # code to be executed and may lead to exceptions

except ExceptionOne:
    # executed if `ExceptionOne` is raised

else:
    # executed after try block, only if `ExceptionOne` is not raised
```

```python
try:
    # code to be executed and may lead to exceptions

except ExceptionOne:
    # executed if `ExceptionOne` is raised

else:
    # executed after try block, only if `ExceptionOne` is not raised

finally:
    # always executed as last task after the try block,
    #  even is exception are raised
    # useful to perform clean-up actions
```

Let's apply try-except to our example

```python
# fizzbuzz_05.py


def is_multiple_of(n, d):
  return n % d == 0


def fb(i, div1=3, div2=5):
  cond1 = is_multiple_of(i, div1)
  cond2 = is_multiple_of(i, div2)
  if cond1 and cond2:
    print("fizzbuzz")
  elif cond1:
    print("fizz")
  elif cond2:
    print("buzz")
  else:
    print(i)
```

```python
while True:
  try:
    div1 = int(input("Value for div1: "))
    div2 = int(input("Value for div2: "))
    i = int(input("Input value: "))
    fb(i, div1, div2)

  except ValueError:
    print("Error: some input is not valid.
           Please pass only integers.")

  except ZeroDivisionError:
    print("Error: you passed zero for a divisor.")

  # Too broad exception clause, should be avoided
  except Exception:
    print("Something went wrong.")
```

This is the **Easier to ask for forgiveness than permission (EAFP)** approach. This coding style assumes the existence of valid keys or attributes and catches exceptions if the assumption proves false.

- the code is more readable
- it is effective for both *known* and *unknown* exceptions
- no issues in multi-threaded environments
- unefficient if exceptions are frequent, it should be used when exceptions are actually exceptions
- sometimes it is better to prevent exceptions rather than handle them
    - be careful when receiving inputs, this was just a toy example!

Anti-patterns

- They are recurrent solutions adopted in software design/development that are usually ineffective and counterproductive
- There are dozens of anti-patterns, related to software development, software architecture design and project management

Brown, William J.; Malveau, Raphael C.; McCormick, Hays W. "Skip"; Mowbray, Thomas J. (1998). Hudson, Theresa (ed.). AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. John Wiley & Sons, ltd.

Anti-pattern: Spaghetti Code

Code written with too little or no structure. Very difficult to mantain, test, extend, reuse.

Anti-pattern: Spaghetti Code

Code written with too little or no structure. Very difficult to mantain, test, extend, reuse.

Solutions

- develop a plan before writing code
- code refactoring and cleanup

Anti-pattern: Dead code

Unused (or commented-out) portions of code, without justification/documentation and clear relationships with the system.

- Typically occurs when research/prototype code quickly moves into production
- Makes software difficult to inspect and test
- Additional memory load might impact performance

Anti-pattern: Dead code

Unused (or commented-out) portions of code, without justification/documentation and clear relationships with the system.

- Typically occurs when research/prototype code quickly moves into production
- Makes software difficult to inspect and test
- Additional memory load might impact performance

Solutions

- proper architecture design *before* developing production code
- remove unneccessary code!

Anti-pattern: Reinvent the Wheel

If a solution for a problem already exists, it should be better to reuse it instead of re-implementing it from scratch.

Often, free or commercial tools and libraries are available

- they are typically well tested and mantained
- it is very unlikely that your re-implemented solution will be better

Don't Reinvent the Wheel - related problems

Continuous Obsolescence: tools/libraries/frameworks rapidly evolve and become incompatible with the rest of the system
Solutions: when possible, use *open systems standards*; try to depend upon interfaces that are stable.

Don't Reinvent the Wheel - related problems

Continuous Obsolescence: tools/libraries/frameworks rapidly evolve and become incompatible with the rest of the system
Solutions: when possible, use *open systems standards*; try to depend upon interfaces that are stable.

Vendor Lock–In: your system is highly dependent upon external components. When they are updated, the system experiments issues and constantly needs adaptations.
Solution: implement an **isolation layer**, which creates a stable custom interface for the external component, separating its functionality and interface from your system. When it is updated, you only have to change the isolation layer.

Other Anti-patterns

Other Anti-patterns

Cut−and−Paste Programming: reuse by copying entire blocks of code. Violates DRY principle.
Solution: leverage alternative form of reuse, such as black-box reuse.

Other Anti-patterns

Cut−and−Paste Programming: reuse by copying entire blocks of code. Violates DRY principle.
Solution: leverage alternative form of reuse, such as black-box reuse.

Input Kludge: software accepting free user inputs does not properly handle them, or use ad-hoc algorithms that easily fail. Often unit test are not able to detect these issues, while end users quickly make the system crash.
Solutions: use available input validation algorithms; perform monkey and fuzz tests.

Other Anti-patterns

Cut–and–Paste Programming: reuse by copying entire blocks of code. Violates DRY principle.
Solution: leverage alternative form of reuse, such as black-box reuse.

Input Kludge: software accepting free user inputs does not properly handle them, or use ad-hoc algorithms that easily fail. Often unit test are not able to detect these issues, while end users quickly make the system crash.
Solutions: use available input validation algorithms; perform monkey and fuzz tests.

Golden hammer: a (familiar) solution is obsessively applied in many contexts, without clear advantages (other than knowing it) and often drawbacks.
Solutions: keep updating and expanding your knowledge on current technologies and alternative approaches; define boundaries between software components to facilitate replaceability.

Code smells

Code smells are characteristics in the source code that *might* reveal deeper issues.

- The presence of code smells does not imply bugs or errors, but wrong/poor code design/organization.
  - this in turn may affect code development, functionality, mantainability, testing, etc.
- They should be easy to spot, and their definition is subjective
  - they vary by programming language, developer and development methodology

Some example of code smells

Too long code block (or parameter list): the code might be difficult to understand, test and mantain. Probabily you are violating the separation of concerns principle.

Some example of code smells

Too long code block (or parameter list): the code might be difficult to understand, test and mantain. Probabily you are violating the separation of concerns principle.

Obscure/incoherent identifiers: always assign meaningful names to variables and other elements. They should make clear the purpose of the code.

Some example of code smells

Too long code block (or parameter list): the code might be difficult to understand, test and mantain. Probabily you are violating the separation of concerns principle.

Obscure/incoherent identifiers: always assign meaningful names to variables and other elements. They should make clear the purpose of the code.

Comments: code should be self-explanatory. If comments are necessary to understand a piece of code, it might be overly complex. Comments should be only used to explain the *why* and not the *what*.

Some example of (already seen) code smells

- Too complex conditional statements
- Duplicate code
- Magic numbers
- Dead code

What to do when you find a code smell?

- Identify what is wrong in your code

What to do when you find a code smell?

- Identify what is wrong in your code
- Understand the implications of the found smell

What to do when you find a code smell?

- Identify what is wrong in your code
- Understand the implications of the found smell
- Evaluate the advantages and the drawbacks of fixing it
    - You don't always have to do it!

What to do when you find a code smell?

- Identify what is wrong in your code
- Understand the implications of the found smell
- Evaluate the advantages and the drawbacks of fixing it
  - You don't always have to do it!
- Check the guidelines on how to fix it

What to do when you find a code smell?

- Identify what is wrong in your code
- Understand the implications of the found smell
- Evaluate the advantages and the drawbacks of fixing it
  - You don't always have to do it!
- Check the guidelines on how to fix it
- Perform the refactoring