S.O.L.I.D Principles and Agile Design

COMP 1531, 17s2

Aarthi Natarajan

Week 5, Tuesday

Why does **Software Rot**?

- We write **bad code**

Why do write bad code ?

- We **do not know** how to write better code
- We are in a **hurry**
- **Changes** and changes requires **refactoring** and refactoring requires **time** and we say **we do not have the time**

Bad code, in fact slows us down

Why do software developers **fear change ?**

- Changes to requirements after design and implementation
- Requires refactoring and lots of code needs to re-written
- Changes impact other parts of the system

But, one of the key principles of Agile Manifesto states…

- Change is a natural and inevitable part of software development life-cycle
- “**Welcome changing requirements”**

How do we recognize a software rot ?

Design Smells

- Rigidity

  - Tendency of software to be difficult to change, even in simple ways

- Fragility

  - Tendency of program to break in many places when a single change is made

- Immobility

  - Program contains parts that could be useful in other systems, but the effort and risk involved with separating those parts from the original system are too great

OO and Agile Design

Why is OO particularly suitable to Agile Design?

All the languages (Ruby, Rails, Python, C#,Java, C++…) support OO

Agile says

- deliver value to the customer,
- working software should model customer's requirements (acceptance criteria),
- if requirements change then quickly modify the associated code

OO stresses:

- Model the customer's problem domain
- Advocates a domain model which establishes an unambiguous understanding of the problem domain
- This domain-based organization is an essential element of agility, because it's the domain-level things that change with every story.

SOLID Principles

Five basic principles of Object Oriented Programming

- SRP – Single Responsibility Principle
- OCP – Open Closed Principle
- LSP – Liskov Substitution Principle
- ISP – Interface Segregation Principle
- DIP – Dependency Injection Principle

SOLID Goals

Make software entities easy to:

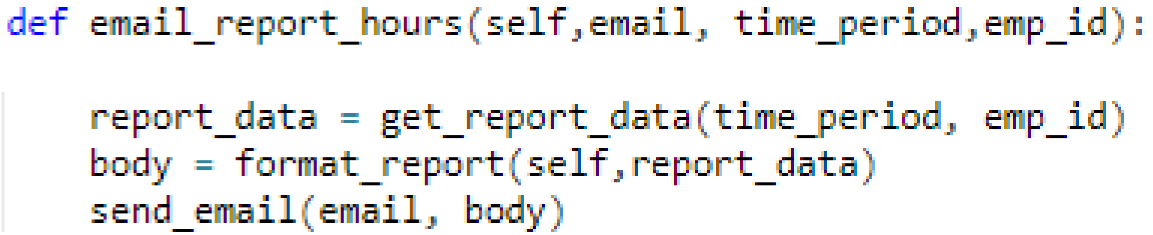
- Understand
- Maintain
- Extend
- Reusable
- Testability

Single Responsibility Principle

"A class should have ***one and only one*** reason to change"

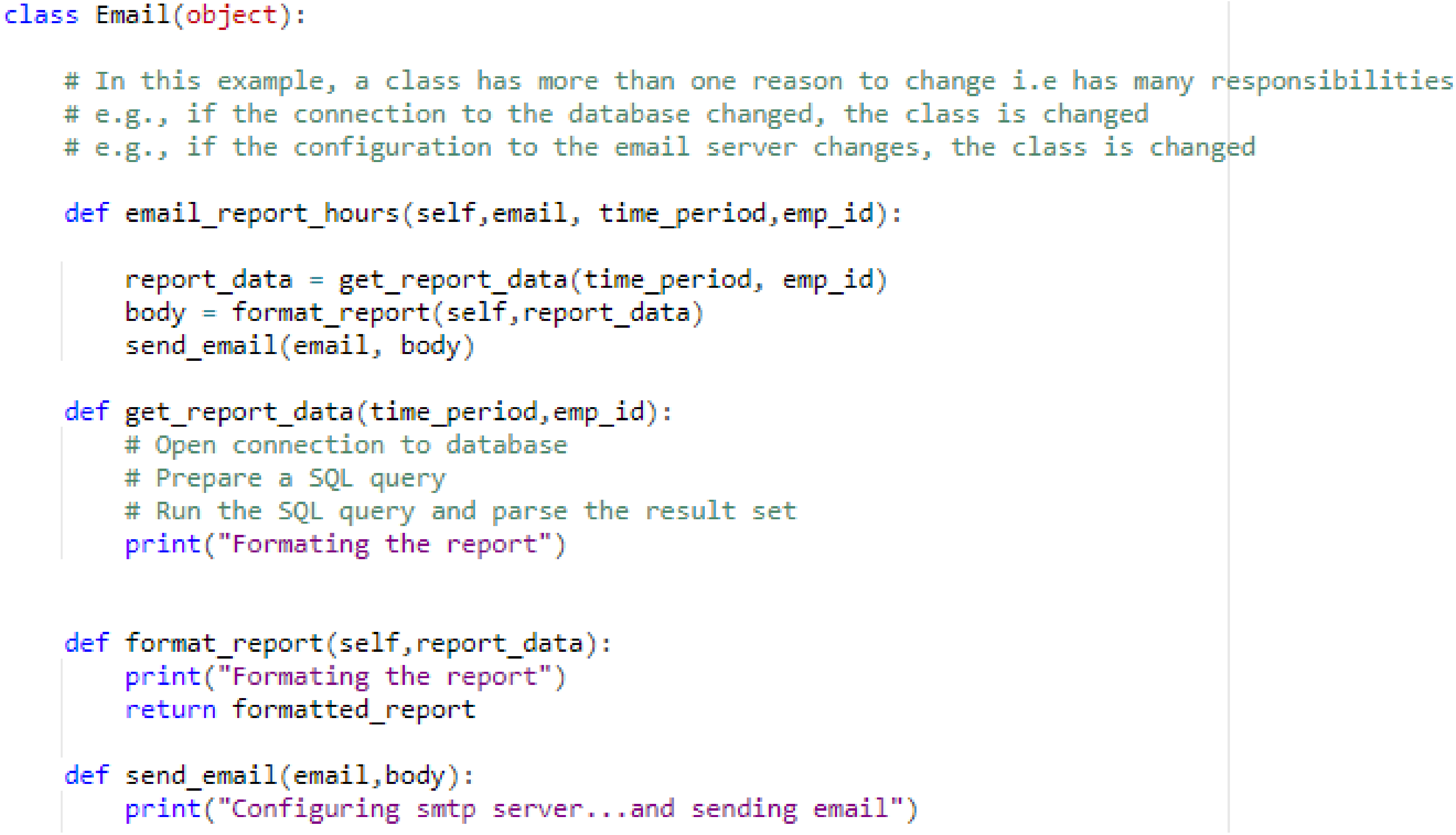
- Every class should have only one responsibility
- Responsibility is defined as a *reason for change*
- If a class has more than one responsibility, the responsibilities become coupled
  - Changes to one responsibility may impair the class's ability to meet the others.
  - Leads to fragile designs that break in unexpected ways when changed.
- Avoid grouping methods from different domains (business rules, persistence, data input…)
- Avoid orchestration and object creation in the same code

- SRP does not imply "do only one thing"
- One function can invoke several other functions, but it should not be responsible for how these functions are implemented
- Consider this example…



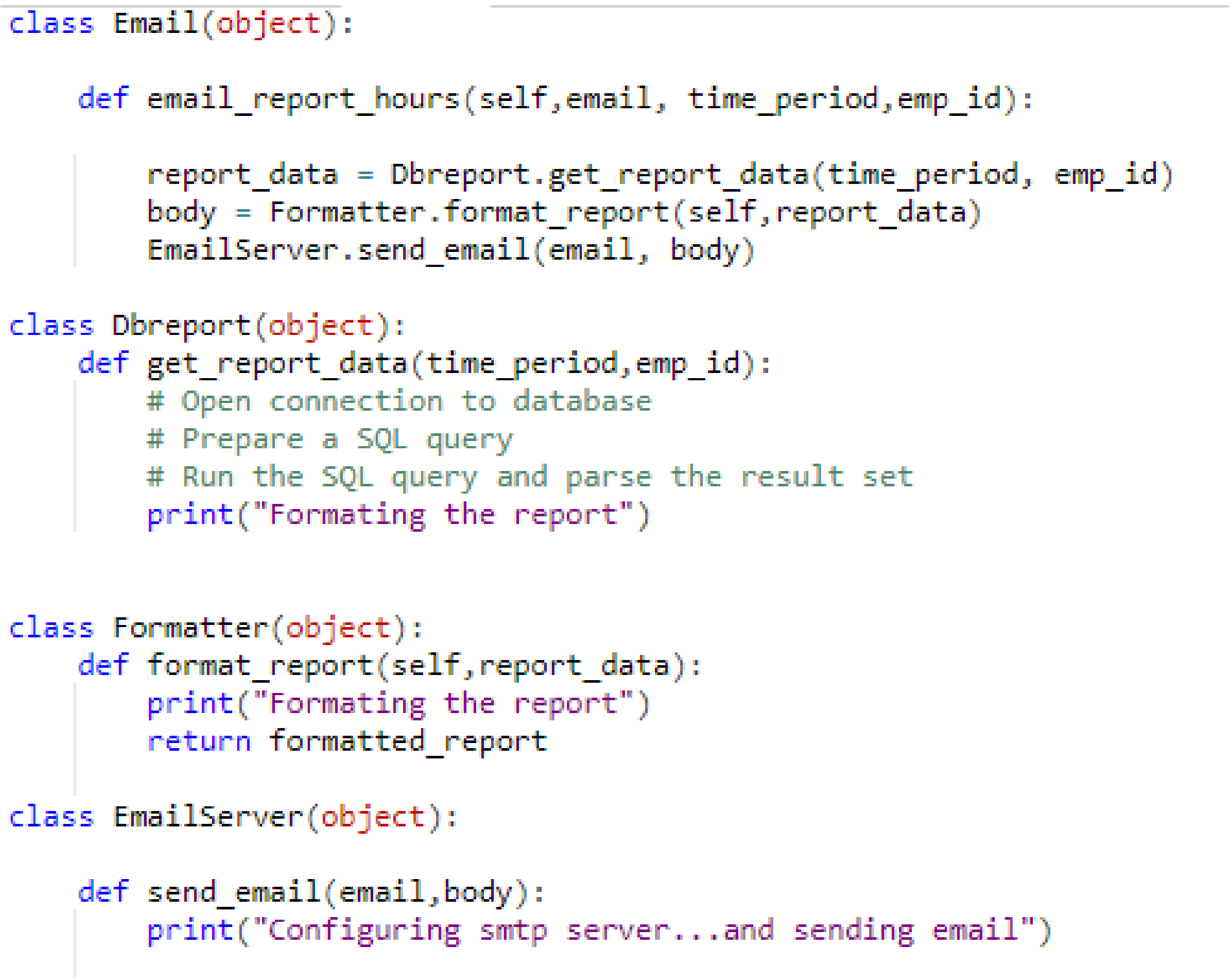
```
def email_report_hours(self,email, time_period,emp_id):

    report_data = get_report_data(time_period, emp_id)
    body = format_report(self,report_data)
    send_email(email, body)
```

Violation of SRP



```python
class Email(object):

    # In this example, a class has more than one reason to change i.e has many responsibilities
    # e.g., if the connection to the database changed, the class is changed
    # e.g., if the configuration to the email server changes, the class is changed

    def email_report_hours(self,email, time_period,emp_id):

        report_data = get_report_data(time_period, emp_id)
        body = format_report(self,report_data)
        send_email(email, body)

    def get_report_data(time_period,emp_id):
        # Open connection to database
        # Prepare a SQL query
        # Run the SQL query and parse the result set
        print("Formating the report")


    def format_report(self,report_data):
        print("Formating the report")
        return formatted_report

    def send_email(email,body):
        print("Configuring smtp server...and sending email")
```

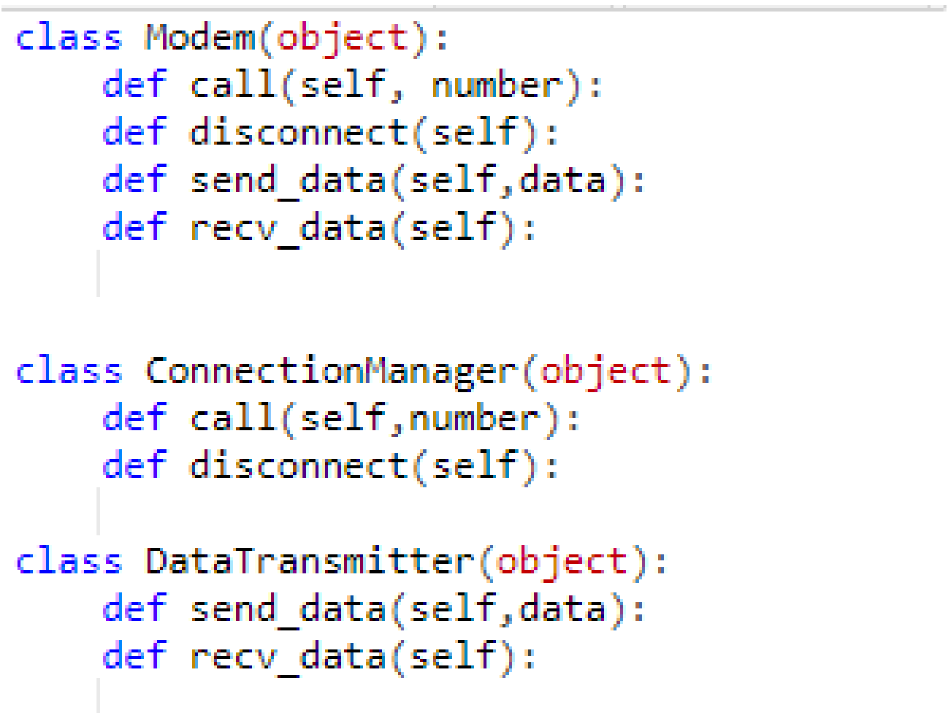
A better design that conforms to SRP



```
class Email(object):

    def email_report_hours(self,email, time_period,emp_id):

        report_data = Dbreport.get_report_data(time_period, emp_id)
        body = Formatter.format_report(self,report_data)
        EmailServer.send_email(email, body)

class Dbreport(object):
    def get_report_data(time_period,emp_id):
        # Open connection to database
        # Prepare a SQL query
        # Run the SQL query and parse the result set
        print("Formating the report")


class Formatter(object):
    def format_report(self,report_data):
        print("Formating the report")
        return formatted_report

class EmailServer(object):

    def send_email(email,body):
        print("Configuring smtp server...and sending email")
```

Is SRP violated?

- Do we need to de-couple the responsibilities ?



```
class Modem(object):
    def call(self, number):
    def disconnect(self):
    def send_data(self,data):
    def recv_data(self):


class ConnectionManager(object):
    def call(self,number):
    def disconnect(self):

class DataTransmitter(object):
    def send_data(self,data):
    def recv_data(self):
```

- A responsibility is an axis of change only if changes occur

Advantages of SRP

One of the fundamental design principles, yet difficult to get it right. When implemented correctly, it helps to achieve

Readability:

- Easier to focus on one responsibility and you can identify the responsibility

Reusability:

- The code can be re-used in different contexts

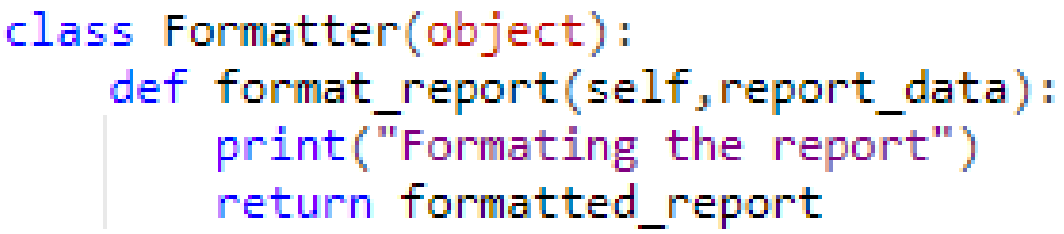
Testability:

- Each responsibility can be tested in isolation
- When a class has encapsulates several responsibilities, several test-cases are required

Open Closed Principle

“Software entities (classes, modules, functions etc) should ***be open for extension closed*** but ***closed for modification***”

- Consider this example…



```
class Formatter(object):
    def format_report(self,report_data):
        print("Formating the report")
        return formatted_report
```

- What if you had to format the report in several different ways, HTML, PDF…?

Open Closed Principle

"Software entities (classes, modules, functions etc) should ***be open for extension*** but ***closed for modification***"

- DRY – Don't Repeat Yourself
- Modules that conform to OCP have two primary attributes:
  - *Open for extension*: As the requirements change, the behaviour of the class can be extended with new behaviours
  - *Closed for modification:* Extending the behaviour of the module does not result in changes to the source, or binary code of the module.
- However, our previous example:
  - Closed for extension and open for modification
  - Cannot use a different formatter without modifying the function's code

“How do write software that is ***open for extension*** but ***closed for modification***”

- The answer is ***abstraction and inheritance***…
- These abstractions are abstract base classes, that represent an unbounded group of possible behaviours.
- The unbounded group of possible behaviours (or the extensions) are defined by possible derived classes (sub-classes)

Conformance to OCP is:

Not Easy

- A skill gained through experience by knowing the users, industry to be able to judge the various kinds of changes
- Educated guesses could be right or wrong. If wrong, you loose time

Expensive:

- Takes development time and effort to create the appropriate abstraction
- Abstractions increase the complexity of software design

Yet yields great benefits:

- Flexibility, Reusability and Maintainability