SWPP Practice Session #10

Design Patterns

Announcement

- Schedule for the rest of the course:
  - 11/2 - Design pattern
  - 11/9 - Project Mid Presentation (Offline)
  - 11/16 - Deployment
  - 11/23 - Optimization + Code refactoring
  - 11/30 - Final Exam (Offline)
  - 12/7 - Testing session
  - 12/14 - Project poster session (Offline)

For today’s lab session,

1. Please clone the repository below. Problems will be uploaded during the practice session.

git clone
https://github.com/swpp22fall-practice-sessions/swpp-p9-design-patterns

1. Now, all the problems are uploaded.

2. Solutions will be uploaded after class

For today’s lab session,

2. Integrate CoverAlls on your team repo

- follow the instructions on the slides
- we will check it by the end of the week

Today’s Contents

1. Design Patterns Practices
2. Integrating CoverAlls (Testing Coverage Tool)
3. Mid presentation

Design Pattern

Goal

- You will be writing short Python codes for some of the design patterns.
- You need to submit the code blocks by the end of each problem.
- So, be prepared for writing some Python codes before we start!

Design Patterns

- Typical solutions to commonly occurring problems in software design.
- Patterns are often confused with algorithms.
  - An analogy to an algorithm is a cooking recipe: both have clear steps to achieve a goal.
  - On the other hand, a pattern is more like a blueprint: you can see what the result and its features are, but the exact order of implementation is up to you.

Why should I learn Design Patterns?

- You might manage to work as a programmer without knowing about a single pattern. Even more, you might be implementing some patterns without even knowing it.
- **So why would you spend time learning design patterns?**
  - Design patterns are **a toolkit of tried and tested solutions** to common problems in software design.
  - Design patterns define **a common language** that you and your teammates can use to communicate more efficiently.

But Design Patterns are not the ultimate solutions

- **Adapting solutions based on design patterns can sometimes be inefficient.**
  - Since patterns try to systematize approaches that are already widely used, adapting patterns without the understandings of their project can be inefficient.
- **Unjustified use of patterns can happen.**
  - Having learned about patterns, programmers try to apply them everywhere, even in situations where simpler code would do just fine.
  - *“If all you have is a hammer, everything looks like a nail.”*

Classification of patterns

- **Creational patterns** provide object creation mechanisms that increase flexibility and reuse of existing code.
  - e.g. Factory, Builder, Singleton
- **Structural patterns** explain how to assemble objects and classes into larger structures, while keeping the structures flexible and efficient.
  - e.g. Adapter, Composite, Proxy, Façade, Decorator
- **Behavioral patterns** take care of effective communication and the assignment of responsibilities between objects.
  - e.g. Observer, Strategy, Visitor

Practice Time!

- Now, you will be given 5 ~ 7 mins for each problem.
- For each problem,
  1. choose the name of the pattern,
  2. fill in the TODO in Python language,
  3. submit your solution code screenshot to Google Form.

- You will be encouraged to present your solution in front of the class!

Maybe a practice as a future TA..?

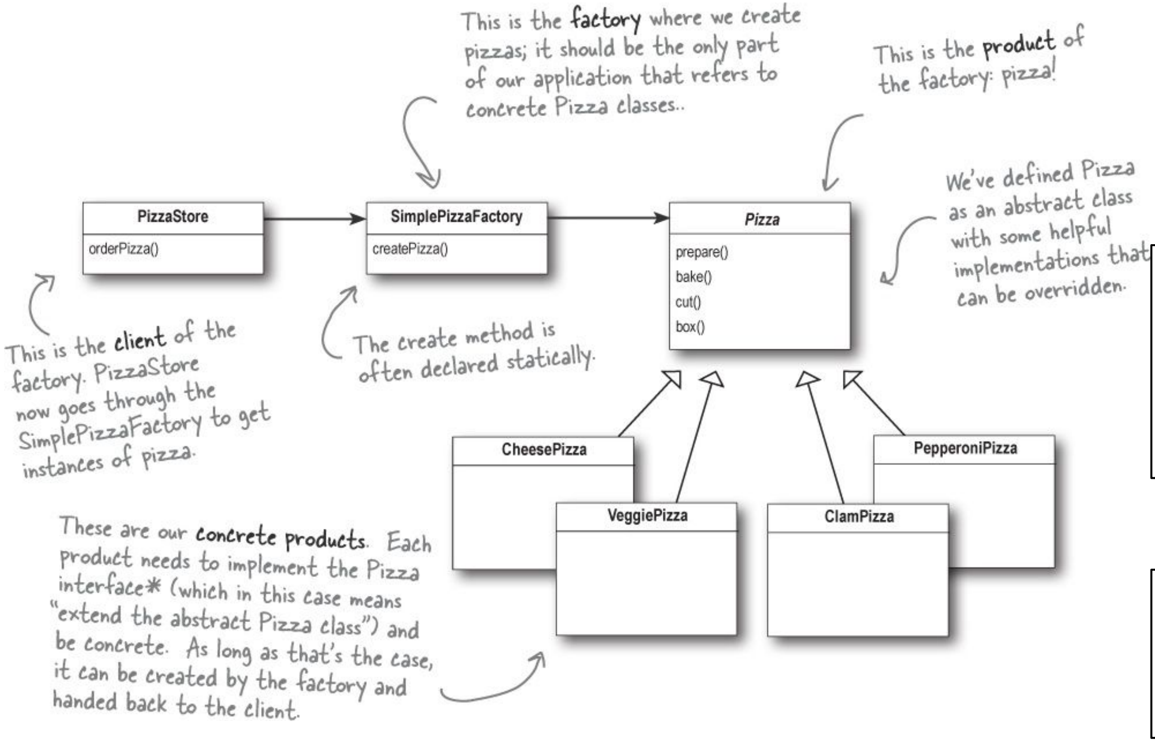
[Problem 1] Which Design Pattern Should We Use?

- We have multiple shapes, and we want a single place to create these shapes accordingly.

Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

Factory example

- SimplePizzaFactory
  - Create pizzas of each type → returns each pizza
- Each type of pizza
  - inherits Pizza type for general interface



This is the factory where we create pizzas; it should be the only part of our application that refers to concrete Pizza classes..

This is the product of the factory: pizza!

We've defined Pizza as an abstract class with some helpful implementations that can be overridden.

This is the client of the factory. PizzaStore now goes through the SimplePizzaFactory to get instances of pizza.

The create method is often declared statically.

These are our concrete products. Each product needs to implement the Pizza interface* (which in this case means "extend the abstract Pizza class") and be concrete. As long as that's the case, it can be created by the factory and handed back to the client.

```
orderPizza(type) {
  pizza = if type == "cheese" return ChessPizza()
             elif type == "veggie" return VeggiePizza()
             …
  pizza.prepare/bake/cut/box()
}
```

factory pattern

```
orderPizza(type) {
  pizza = simplePizzaFactory.createPizza(type)
  pizza.prepare/bake/cut/box()
}
```

[Problem 1] Fill in TODO

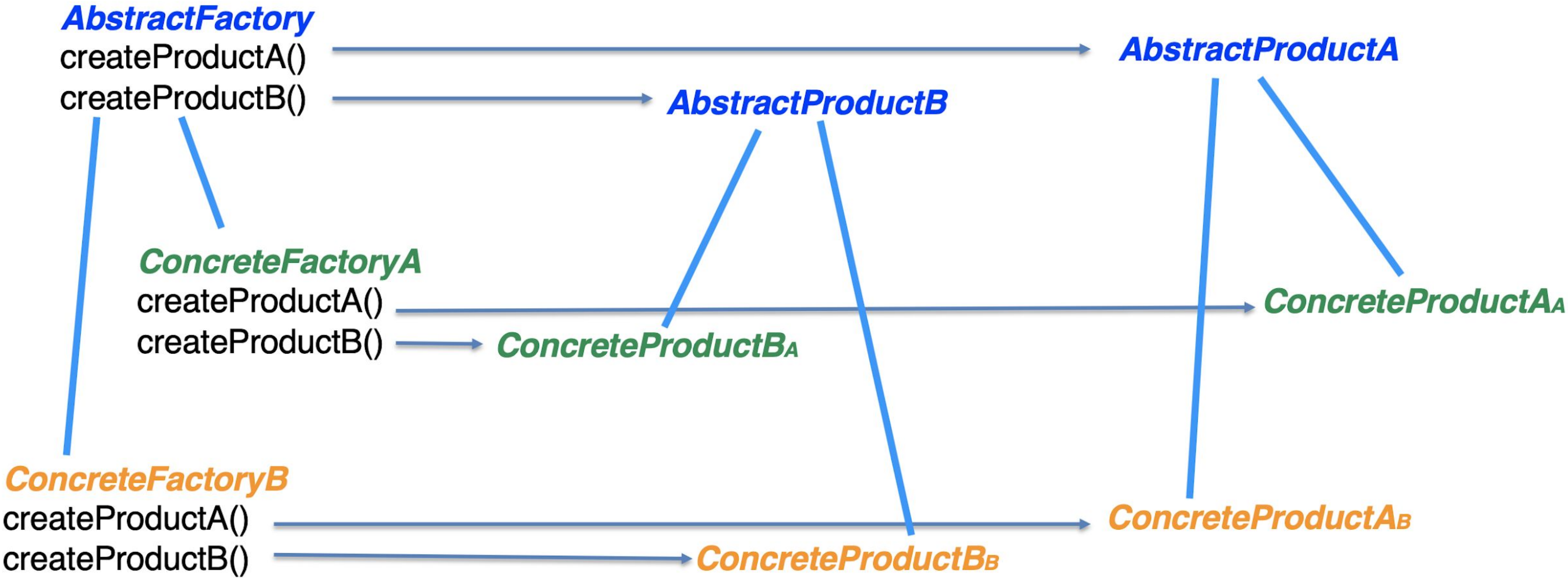
We have multiple shapes, and we want a single place to create these shapes accordingly.

```
class Shape:
    @staticmethod
    def create(name):
        # TODO: Fill the code here
        # NOTE: Raise `ValueError("Invalid name",
                name)` if there is no such shape
class Circle(Shape):
    def draw(self):
      print("○")

class Square(Shape):
    def draw(self):
      print("□")

class Line(Shape):
    def draw(self):
      print("—")
```

```
if __name__ == "__main__":
    x = Shape.create("circle")
    x.draw()
    x = Shape.create("square")
    x.draw()
    x = Shape.create("line")
    x.draw()
```

[Problem 1] Submission Link

Abstract Factory in Python



AbstractFactory
createProductA()
createProductB()

AbstractProductA

AbstractProductB

ConcreteFactoryA
createProductA()
createProductB()

ConcreteProductA$_A$

ConcreteProductB$_A$

ConcreteFactoryB
createProductA()
createProductB()

ConcreteProductA$_B$

ConcreteProductB$_B$

Abstract Factory in Python

Python's abc (abstract base class) module helps us make `AbstractClass`, `Interface`, …

```python
from abc import ABC as AbstractBaseClass
from abc import abstractmethod
class AbstractProductA(AbstractBaseClass):
    @abstractmethod
    def function_A(self):
        pass
class AbstractProductB(AbstractBaseClass):
    @abstractmethod
    def function_B(self):
        pass
class AbstractFactory(AbstractBaseClass):
    @abstractmethod
    def create_product_A(self) -> AbstractProductA:
        pass

    @abstractmethod
    def create_product_B(self) -> AbstractProductB:
        pass
```

```python
class ConcreteFactory1(AbstractFactory):
    def create_product_A(self):
        return ConcreteProductA1()

    def create_product_B(self):
        return ConcreteProductB1()

class ConcreteFactory2(AbstractFactory):
    def create_product_A(self):
        return ConcreteProductA2()

    def create_product_B(self):
        return ConcreteProductB2()
```

Abstract Factory in Python

Python's abc (abstract base class) module helps us make `AbstractClass`, `Interface`, …

```python
class ConcreteProductA1(AbstractProductA):
    def function_A(self):
        return "function of product A1"

class ConcreteProductA2(AbstractProductA):
    def function_A(self):
        return "function of product A2"

class ConcreteProductB1(AbstractProductB):
    def function_B(self):
        return "function of product B1"

class ConcreteProductB2(AbstractProductB):
    def function_B(self):
        return "function of product B2"
```

```python
if __name__ == "__main__":
    a = ConcreteFactory1().create_product_A()
    print(a.function_A())  # function of product A1

    b = ConcreteFactory2().create_product_B()
    print(b.function_B())  # function of product B2
```

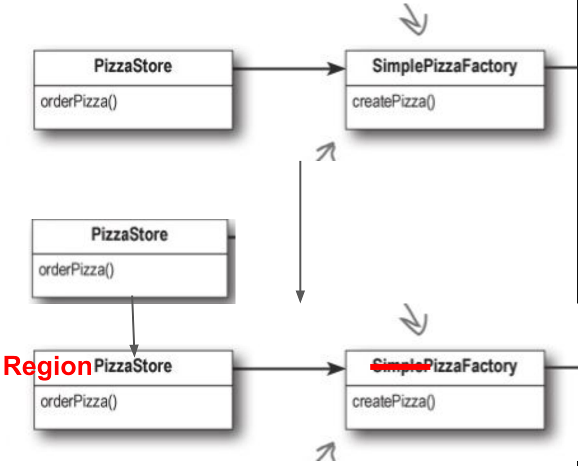
Dependency Injection in Python

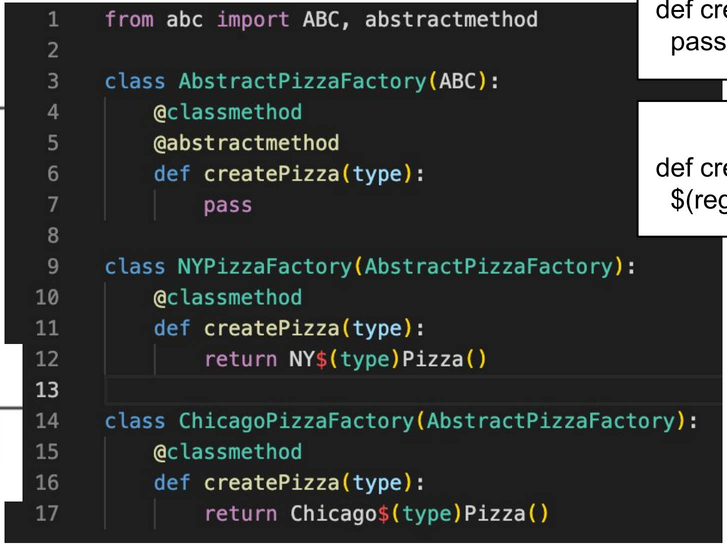
- Some tools in Python for dependency injection
  - https://github.com/ets-labs/python-dependency-injector
  - https://github.com/google/pinject

Abstract factory example

What if we want

- region-different(NY, Chicago, etc.,) style pizza stores
- each employ different pizza factories
- each serve different region-flavored pizza types





```python
from abc import ABC, abstractmethod

class AbstractPizzaFactory(ABC):
    @classmethod
    @abstractmethod
    def createPizza(type):
        pass

class NYPizzaFactory(AbstractPizzaFactory):
    @classmethod
    def createPizza(type):
        return NY$(type)Pizza()

class ChicagoPizzaFactory(AbstractPizzaFactory):
    @classmethod
    def createPizza(type):
        return Chicago$(type)Pizza()
```

Pizza Store

```
def orderPizza(type):
  pizza = createPizza(type)
  pizza.prepare/bake/cut/box()


def createPizza(type):
  pass
```

Regional Pizza Store

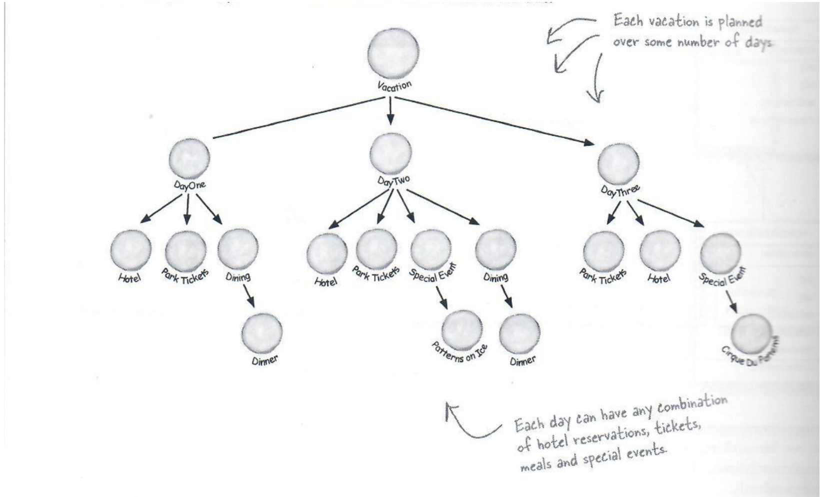
```
def createPizza(type):
  $(region)PizzaFactory.createPizza(type)
```

[Problem 2] Which Design Pattern Should We Use?

- Assume we are developing building renderer. We are going to render every construction steps. Moreover, we have multiple build targets.

Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

Builder scenario

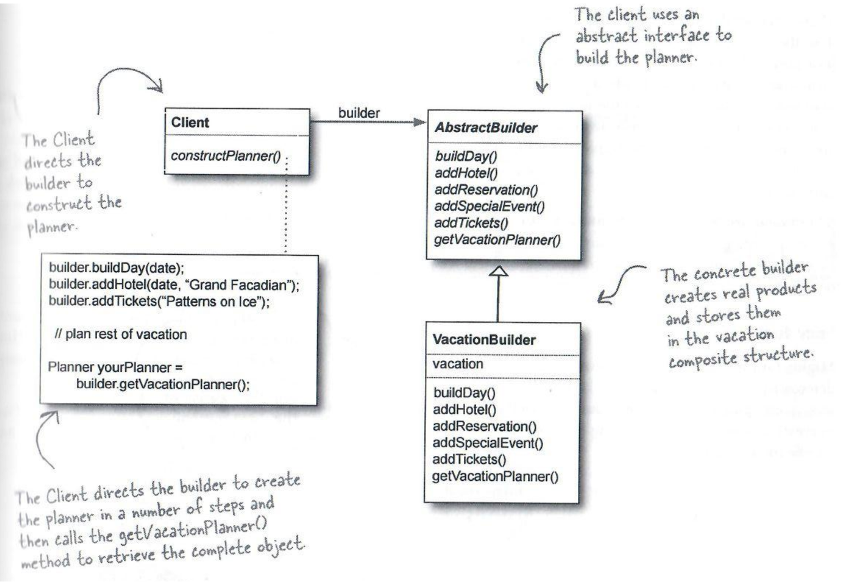


Each vacation is planned over some number of days.

Each day can have any combination of hotel reservations, tickets, meals and special events.

- Clients should be able to build **flexible** plans by following complex creation steps
  - Flexible vacation plan that can represent all client planners and their variations!

⇒ Encapsulate creation of the planner in an object (“builder”) → client ask the builder to construct the vacation planner structure for it



```
builder.buildDay(date);
builder.addHotel(date, “Grand Facadian”);
builder.addTickets(“Patterns on Ice”);

 // plan rest of vacation

Planner yourPlanner =
      builder.getVacationPlanner();
```

The client uses an abstract interface to build the planner.

The Client directs the builder to construct the planner.

The concrete builder creates real products and stores them in the vacation composite structure.

The Client directs the builder to create the planner in a number of steps and then calls the getVacationPlanner() method to retrieve the complete object.

Builder benefits	Builder uses and drawbacks
- Encapsulate complex construction of object - Allow construction in multi-step & varying process (<--> factory) - Hide internal representation of product from the client - Product implementation can be modified since client only sees abstract interface	- Often used for building composite structures - Constructing objects requires more domain knowledge of the client than when using a factory

[Problem 2]

Assume we are developing build-process renderer. We are going to render every construction steps. Moreover, we have multiple build targets (floor -> body -> ceiling).

```python
class A:
    def build_floor(self):
        raise NotImplementedError
    def build_body(self):
        raise NotImplementedError
    def build_ceiling(self):
        raise NotImplementedError

class House(A):
    def build_floor(self):
        print("Build House Floor")

    def build_body(self):
        print("Build House Body")

    def build_ceiling(self):
        print("Build House Ceiling")
```

```python
class Building(A):
    def build_floor(self):
        print("Build Building Floor")

    def build_body(self):
        print("Build Building Body")

    def build_ceiling(self):
        print("Build Building Ceiling")

def build(cls):
    # TODO: fill this method

if __name__ == "__main__":
    build(House)
    build(Building)
```

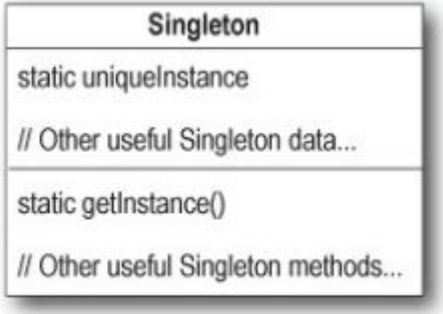
[Problem 2] Submission Link

[Problem 3] Which Design Pattern Should We Use?

- We want to continuously update a single object from different places (e.g. machine state, global configuration). We want the data to be consistent from previous changes.

Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.



Singleton

static uniqueInstance

// Other useful Singleton data...

static getInstance()

// Other useful Singleton methods...

The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.

[Problem 3]

We want to continuously update a single object from different places. We want the data to be consistent from previous changes.

The code below prints the name(val) that is set by the constructor.

```
class B:
    class _B:
        def __init__(self, value):
            self.value = value
    _instance = None
    @staticmethod
    def get_instance():
        if B._instance is None:
            B()
        return B._instance

    def __getattr__(self, name):
        return getattr(B._instance, name)
    def __init__(self, value=0):
        # TODO: fill constructor
```

```
if __name__ == ‘__main__’:
    b1 = B(10)
    print(b1.value)
    print(B.get_instance())
    b2 = B(20)
    print(b2.value)
    print(b1.value)
    print(B.get_instance())

#### OUTPUT (example)
10
<__main__.B._B object at 0x10cbcc210>
20
20
<__main__.B._B object at 0x10cbcc210>
```

[Problem 3] Submission Link

[Problem 4] Which Design Pattern Should We Use?

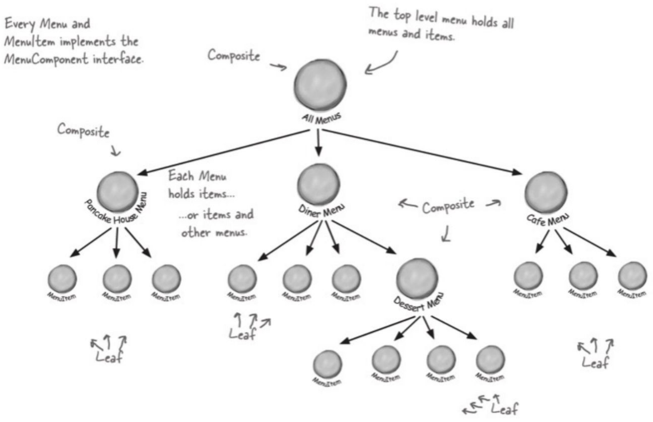
- We want to treat three different functions as a single composite function

```
f(x) = x + 1
g(x) = x * x
h(x) = x - 3

func(x) = h(g(f(x)))
```

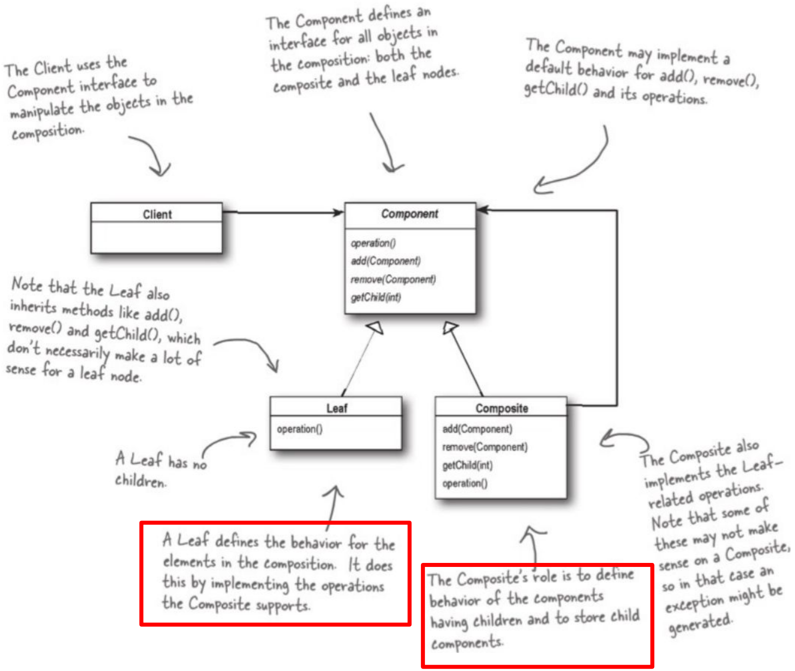
Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

Composite pattern

- Compose objects into tree structures ⇒ Represent “part-whole” hierarchy
  - Part-whole: Tree(whole) of objects, which are also small tree(part) of objects
  - ⇒ Can treat individual objects and compositions uniformly
    - Operations(e.g., print()) can be applied to the whole or the parts



Every Menu and MenuItem implements the MenuComponent interface.
Composite
The top level menu holds all menus and items.
All Menus
Composite
Pancake House Menu
Each Menu holds items...
...or items and other menus.
Diner Menu
Composite
Cafe Menu
Dessert Menu
MenuItem
Leaf
Leaf
Leaf
Leaf



The Client uses the Component interface to manipulate the objects in the composition.
The Component defines an interface for all objects in the composition: both the composite and the leaf nodes.
The Component may implement a default behavior for add(), remove(), getChild() and its operations.
Client
Component
operation()
add(Component)
remove(Component)
getChild(int)
Note that the Leaf also inherits methods like add(), remove() and getChild(), which don't necessarily make a lot of sense for a leaf node.
Leaf
operation()
Composite
add(Component)
remove(Component)
getChild(int)
operation()
A Leaf has no children.
The Composite also implements the Leaf-related operations. Note that some of these may not make sense on a Composite, so in that case an exception might be generated.
A Leaf defines the behavior for the elements in the composition. It does this by implementing the operations the Composite supports.
The Composite's role is to define behavior of the components having children and to store child components.

- Composite -> print(): Call print() of all children
- Leaf -> print(): Print its value

[Problem 4]

We want to treat 3 functions as a single composite function

```
class Function:
    def __call__(self, value):
        raise NotImplementedError
class CompositeFunction(Function):
    def __init__(self, *functions):
        # TODO: fill this code
        # NOTE: `functions` is a tuple of arguments
    def __call__(self, value):
        # TODO: fill this code
        # HINT: `reversed(TUPLE)` returns a tuple in
                reversed order
class F(Function):
    def __call__(self, value): return value + 1

class G(Function):
    def __call__(self, value): return value ** 2

class H(Function):
    def __call__(self, value): return value - 3
```

```
if __name__ == "__main__":
    f = F()
    g = G()
    h = H()

    func = CompositeFunction(h, g, f)

    print(func(2)) # 6
```

[Problem 4] Submission Link

[Problem 5] Which Design Pattern Should We Use?

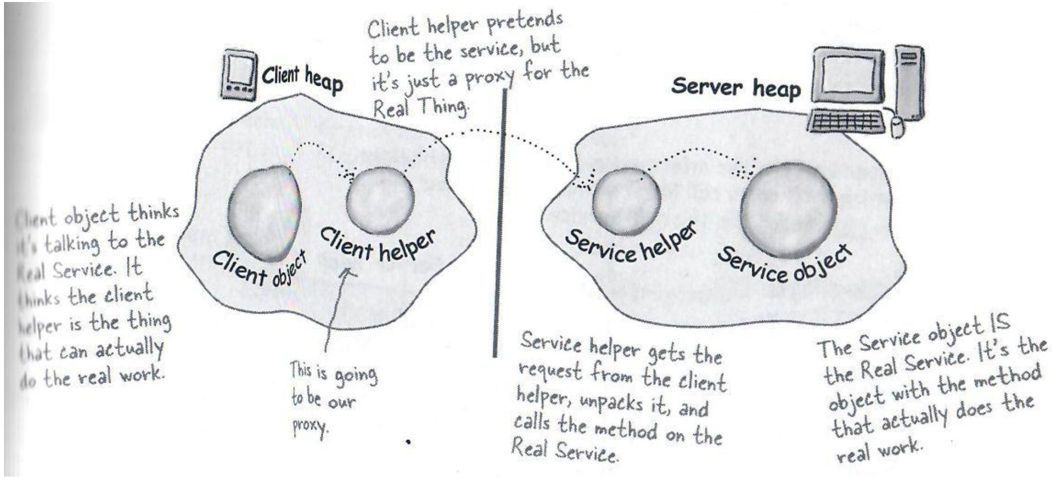
- There are two `Counselor`s, who will talk if a client asks via `PhoneCall` on weekdays, and wouldn’t talk if a client asks via `PhoneCall` on weekends.
  Also, even if a client asks via `PhoneCall` on weekdays, if both counselors are busy, it will return a message saying “no available counselor”.

Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

Proxy

: “Stand in”(i.e., “act”) for a *real* object, but behind the scenes it is communicating over the network to talk to the real object

e.g., Takes a method invocation → transfers it over the network → invoke the same method on remote object


Client heap
Client helper pretends to be the service, but it's just a proxy for the Real Thing.
Server heap
Client object
Client helper
Service helper
Service object
Client object thinks it's talking to the Real Service. It thinks the client helper is the thing that can actually do the real work.
This is going to be our proxy.
Service helper gets the request from the client helper, unpacks it, and calls the method on the Real Service.
The Service object IS the Real Service. It's the object with the method that actually does the real work.

[Problem 5]

```
import time

class Counselor:
    def __init__(self):
        self.busy = False

    def talk(self):
        print("I am ready to talk")

class PhoneCall:
    def __init__(self, current_time='weekdays'):
        self.counselors = [Counselor(), Counselor()]
        self.current_time = current_time

    def talk(self):
        if self.current_time == 'weekdays':
            # TODO: fill this function
        else:
            time.sleep(0.1)
            print('Counselor will not talk to you')
```

```
if __name__ == "__main__":
    p = PhoneCall('weekdays')
    p.talk()
    p.talk()
    p.talk()

    p = PhoneCall('weekend')
    p.talk()
```

[Problem 5] Submission Link

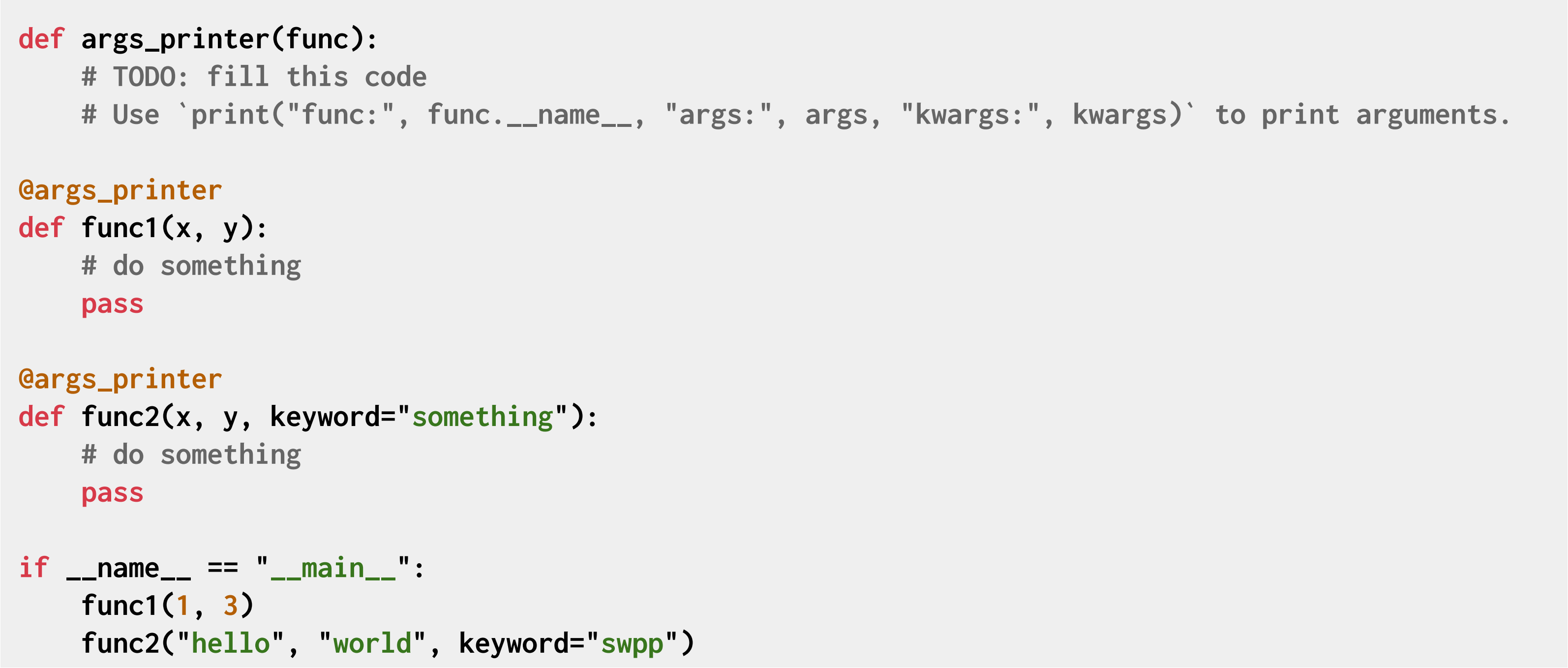
[Problem 6] Which Design Pattern Should We Use?

- You are developing a program and you are now debugging an error step by step. For debugging purpose, you want to print all passed arguments of every function you wrote.

Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

[Problem 6]

You are developing a program and you are now debugging an error step by step. For debugging purpose, you want to print all passed arguments of every function you wrote.



```python
def args_printer(func):
    # TODO: fill this code
    # Use `print("func:", func.__name__, "args:", args, "kwargs:", kwargs)` to print arguments.

@args_printer
def func1(x, y):
    # do something
    pass

@args_printer
def func2(x, y, keyword="something"):
    # do something
    pass

if __name__ == "__main__":
    func1(1, 3)
    func2("hello", "world", keyword="swpp")
```

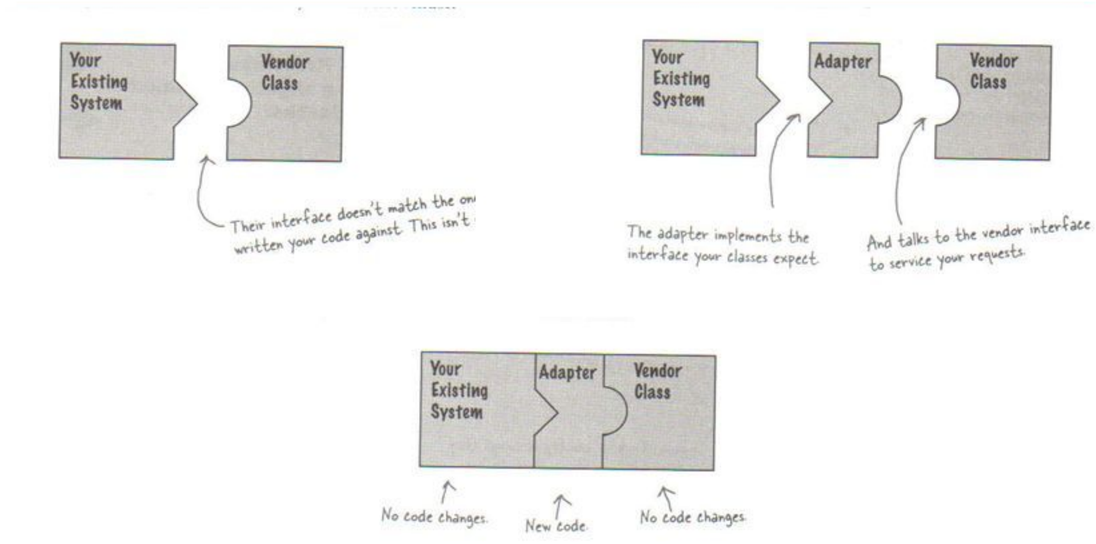
[Problem 6] Submission Link

[Problem 7] Which Design Pattern Should We Use?

- We have a Dog that we want it to *bark* when we tell it to *talk*. We initially have a Person and a Dog class, and we want to tell a Dog object to *talk* using the New Class.

Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

Adapter


Your Existing System
Vendor Class
Their interface doesn't match the on
written your code against. This isn't
Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect.
And talks to the vendor interface to service your requests.
Your Existing System
Adapter
Vendor Class
No code changes.
New code.
No code changes.

- Vendor suddenly changed their interfaces!
- I don't want to change my existing code.. (this vendor may not last long..?) and I change the vendor's code

⇒ Write a class that adapts the new vendor interface

⇒ it will act as the middleman

- And when the original vendor comes back.. bye bye adapter!

[Problem 7]

We have a Dog that we want it to *bark* when we tell it to *talk*. We initially have a Person class and a Dog class, and we want to tell a Dog object to *talk* using the New Class.

```python
class Dog:
    def __init__(self):
        self.name = "Dog"

    def bark(self):
        return "woof!"
class Person:
    def __init__(self):
        self.name = "Human"

    def talk(self):
        return "talk"
class C:
    def __init__(self, obj, **adapted_methods):
        # TODO: fill this
    def __getattr__(self, attr):
        # TODO: fill this
```

```python
if __name__ == "__main__":
    dog = Dog()
    talkable = C(dog, # TODO: fill this)
    print(talkable.name)
    print(talkable.talk())
```

[Problem 7] Submission Link

[Problem 8] Which Design Pattern Should We Use?

- We want to manage `MatrixInfo` class, which contains `shape` and `value` information of a matrix. However, we want to prevent changing `shape` attribute arbitrarily. Moreover, we do not want to name getter/setter as `get_shape`, `set_shape`

Decorator / Builder / Factory / Abstract Factory / Façade / Singleton / Proxy / Adapter / Composite / Iterator & Generator / Getter & Setter, Observer

[Problem 8]

We want to manage `MatrixInfo` class, which contains shape and value information of a matrix. However, we want to prevent changing `shape` attribute arbitrarily. Moreover, we do not want to name getter/setter as `get_shape`, `set_shape`

```python
class MatrixInfo:
    def __init__(self, matrix):
        self._shape = [len(matrix), len(matrix[0])]
        self._value = matrix

    def get_shape(self):
        return self._shape
if __name__ == "__main__":
    # 3 x 4 matrix
    sample_matrix = [[i for i in range(4)] for _ in range(3)]

    container = MatrixInfo(sample_matrix)
    print(container.get_shape())  # [3, 4]
    container.get_shape()[0] = 0
    print(container.get_shape())  # [0, 4], do not want this behavior
```

[Problem 8] Submission Link

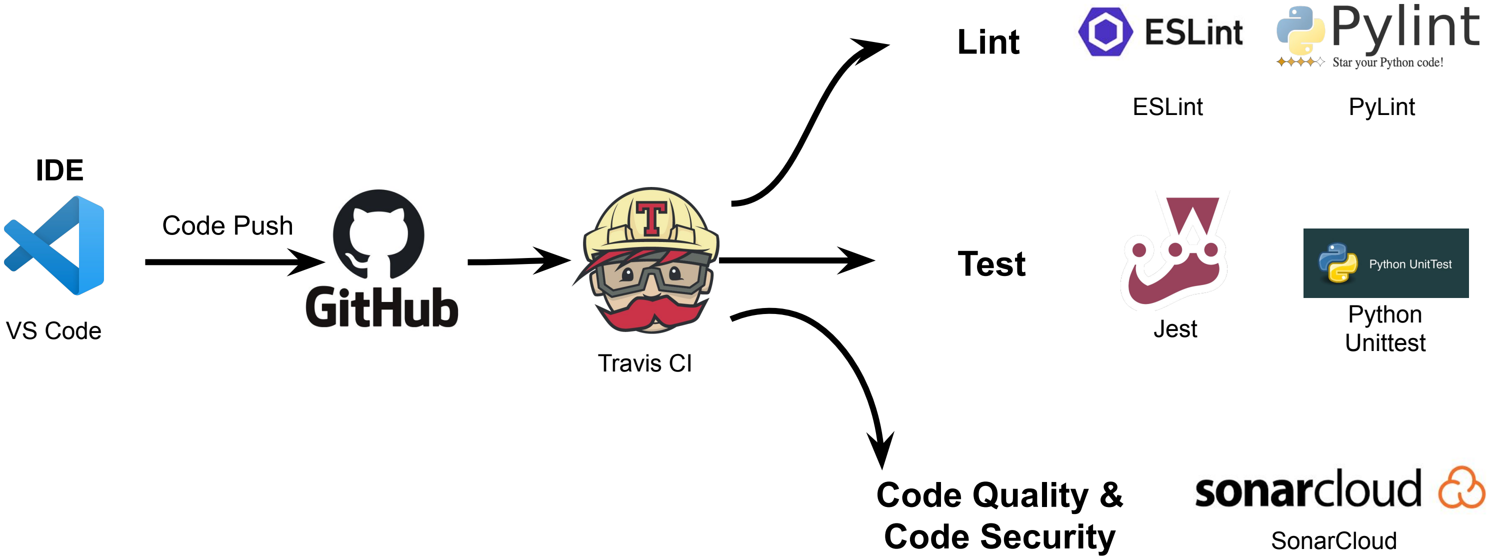
References

- https://github.com/faif/python-patterns
- https://refactoring.guru/design-patterns/python

Coveralls

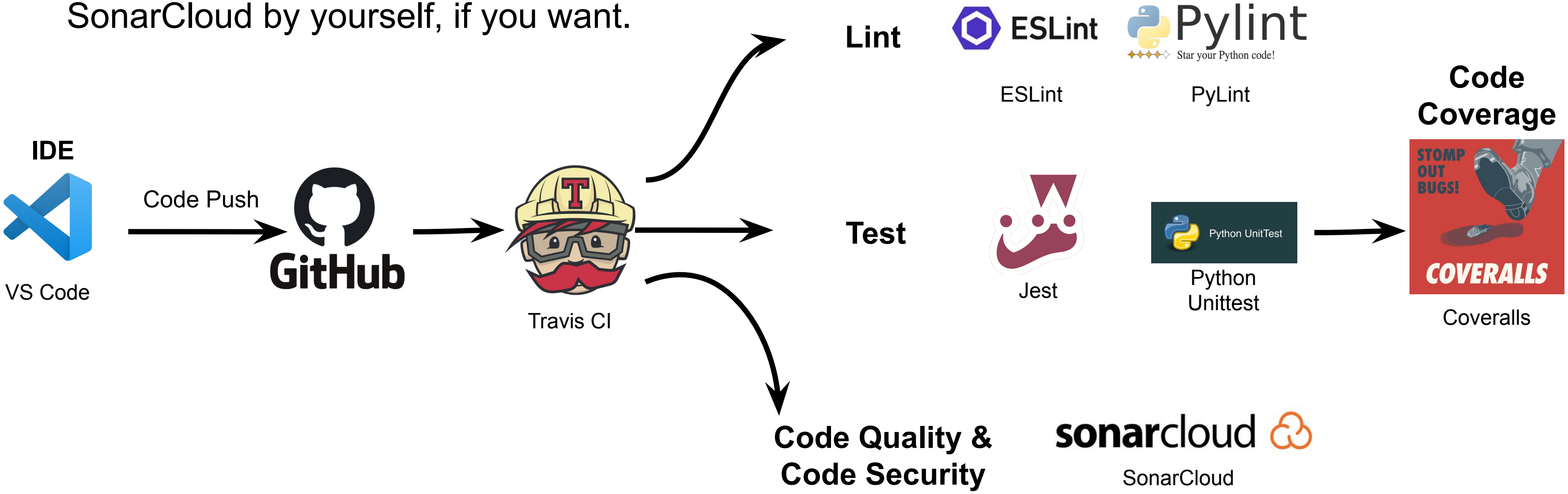
In the last lab session…

- Integration of Travis CI with the team repo
  - Automatic testing/code analysis for every push/PR
  - Service reproduction under the same environment



Code coverage report with CoverAlls

- We will integrate CoverAlls to help you track your code coverage.
- Note that different settings are required for different languages.
- Also, we are not going to use coverage metric in SonarCloud. You can set the coverage in SonarCloud by yourself, if you want.



IDE

VS Code

Code Push

Travis CI

Lint

ESLint

PyLint

Test

Jest

Python Unittest

Code Coverage

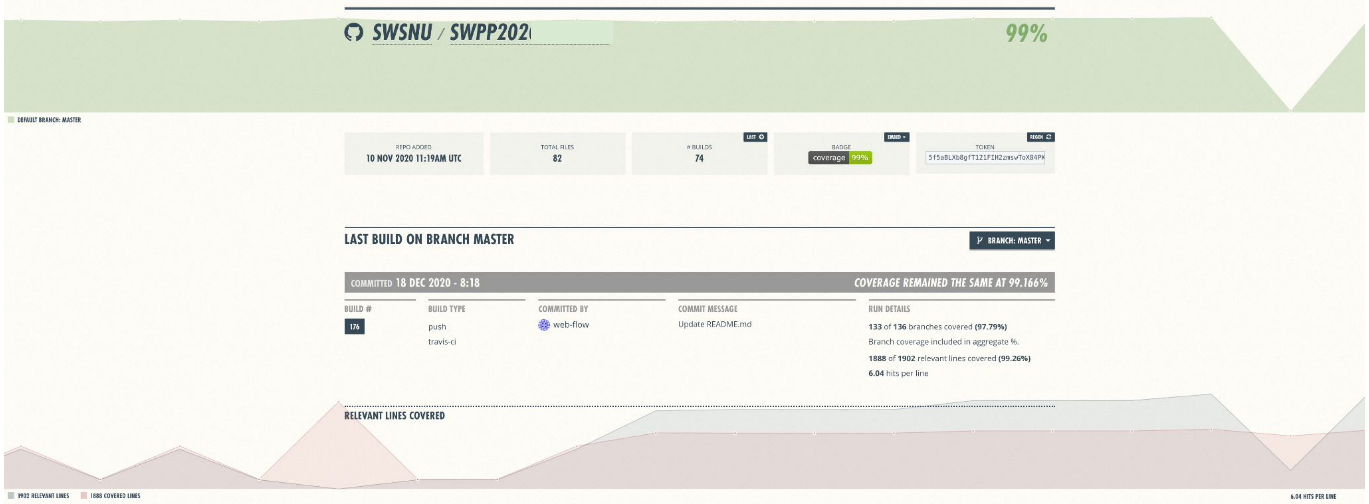
Coveralls

Code Quality & Code Security

SonarCloud

CoverAlls

- CoverAlls is a web service which helps you track the code coverage history and statistics.
- By integrating your GitHub repo and Travis CI, you get take a look at the code coverage for all repo actions(e.g., PR).
- Now, I will show you how to integrate CoverAlls in your repo.
- You can check the examples in [our organization](#).



SWSNU / SWPP202

99%

DEFAULT BRANCH: MASTER

REPO ADDED	TOTAL FILES	# BUILDS	BADGE	TOKEN
10 NOV 2020 11:19AM UTC	82	74	coverage 99%	5f5aBLXb8gfT121FIH2zmswToXB4PK

LAST BUILD ON BRANCH MASTER

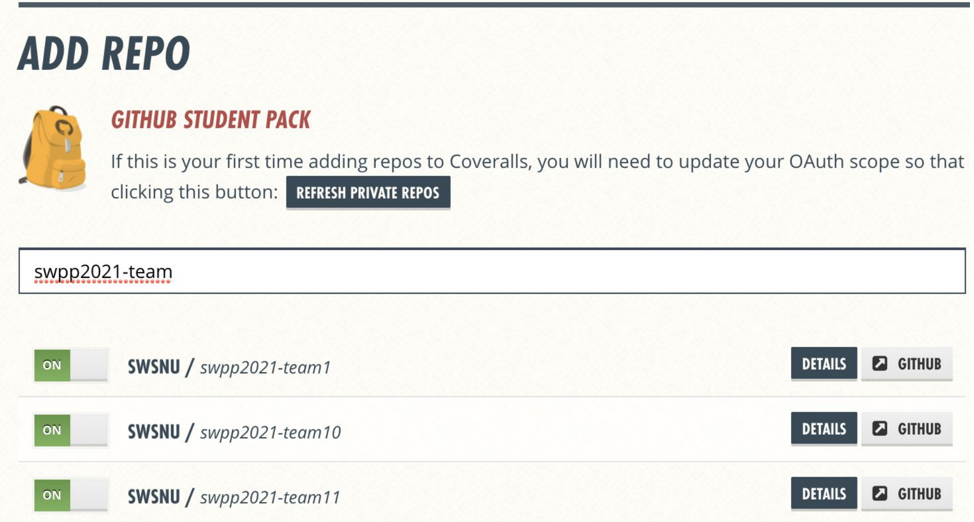
COMMITTED 18 DEC 2020 - 8:18 COVERAGE REMAINED THE SAME AT 99.166%

BUILD #	BUILD TYPE	COMMITTED BY	COMMIT MESSAGE	RUN DETAILS
176	push travis-ci	web-flow	Update README.md	133 of 136 branches covered (97.79%) Branch coverage included in aggregate %. 1888 of 1902 relevant lines covered (99.26%) 6.04 hits per line

RELEVANT LINES COVERED

How to integrate CoverAlls

STEP 1. Go to `Add Repo` and activate CoverAlls in the repository.



ADD REPO

GITHUB STUDENT PACK

If this is your first time adding repos to Coveralls, you will need to update your OAuth scope so that clicking this button: REFRESH PRIVATE REPOS

swpp2021-team

ON SWSNU / *swpp2021-team1* DETAILS GITHUB

ON SWSNU / *swpp2021-team10* DETAILS GITHUB

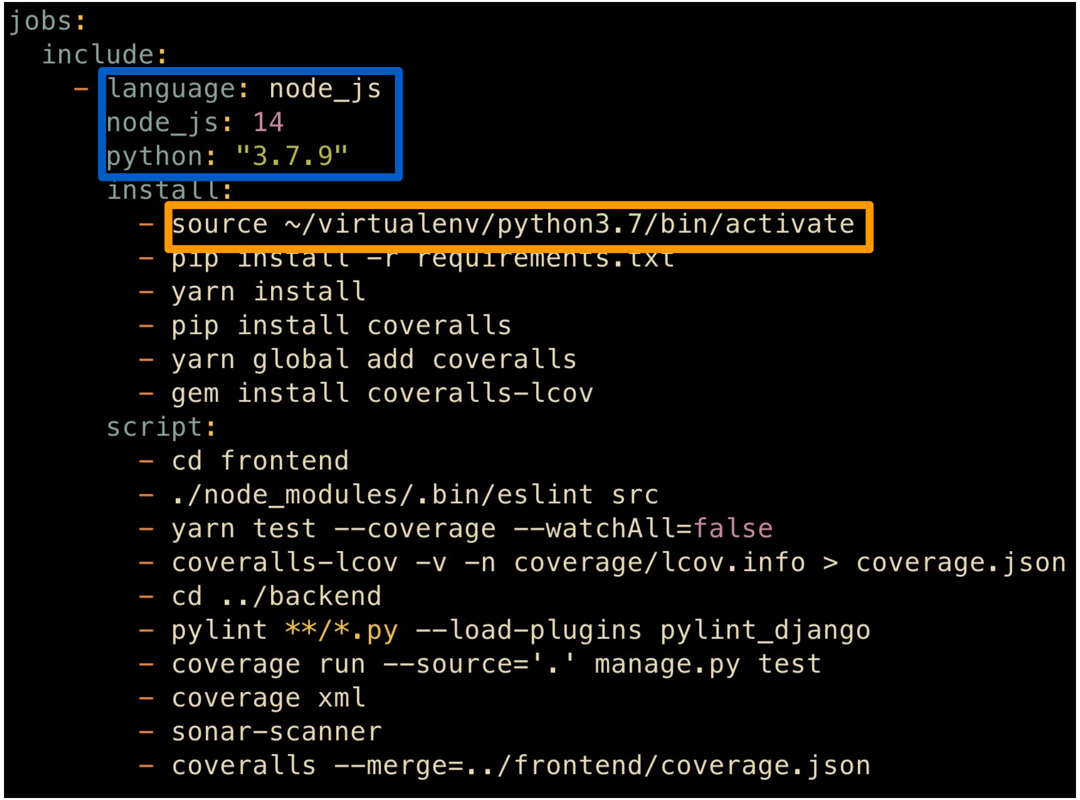
ON SWSNU / *swpp2021-team11* DETAILS GITHUB

How to integrate CoverAlls

STEP 2. Configure Travis build job.

Both Node JS and Python versions should be specified to run JS and Python in a single Travis job.

Note that you should activate Python 3.7 manually. If you don’t, the default Python with version 2.7 will be used.



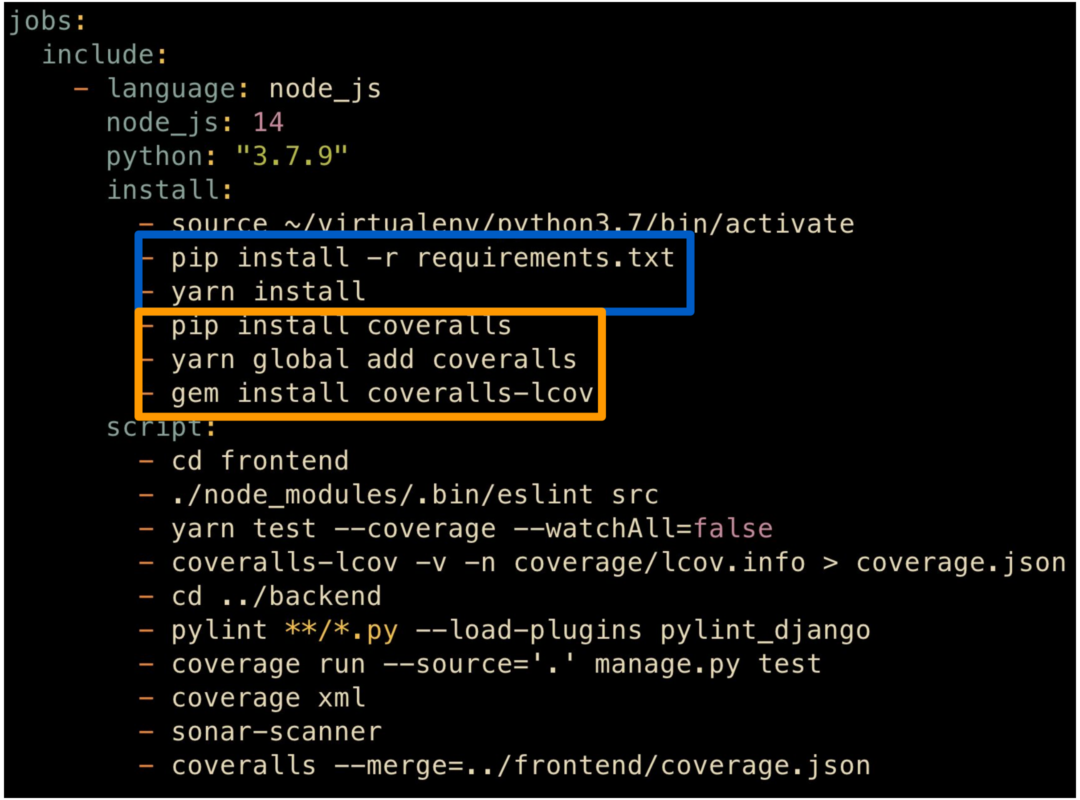
```
jobs:
  include:
    - language: node_js
      node_js: 14
      python: "3.7.9"
      install:
        - source ~/virtualenv/python3.7/bin/activate
        - pip install -r requirements.txt
        - yarn install
        - pip install coveralls
        - yarn global add coveralls
        - gem install coveralls-lcov
      script:
        - cd frontend
        - ./node_modules/.bin/eslint src
        - yarn test --coverage --watchAll=false
        - coveralls-lcov -v -n coverage/lcov.info > coverage.json
        - cd ../backend
        - pylint **/*.py --load-plugins pylint_django
        - coverage run --source='.' manage.py test
        - coverage xml
        - sonar-scanner
        - coveralls --merge=../frontend/coverage.json
```

How to integrate CoverAlls

STEP 2. Configure Travis build job.

Install dependencies for both Python and Javascript.

Install CoverAlls packages for each language.



```
jobs:
  include:
    - language: node_js
      node_js: 14
      python: "3.7.9"
      install:
        - source ~/virtualenv/python3.7/bin/activate
        - pip install -r requirements.txt
        - yarn install
        - pip install coveralls
        - yarn global add coveralls
        - gem install coveralls-lcov
      script:
        - cd frontend
        - ./node_modules/.bin/eslint src
        - yarn test --coverage --watchAll=false
        - coveralls-lcov -v -n coverage/lcov.info > coverage.json
        - cd ../backend
        - pylint **/*.py --load-plugins pylint_django
        - coverage run --source='.' manage.py test
        - coverage xml
        - sonar-scanner
        - coveralls --merge=../frontend/coverage.json
```

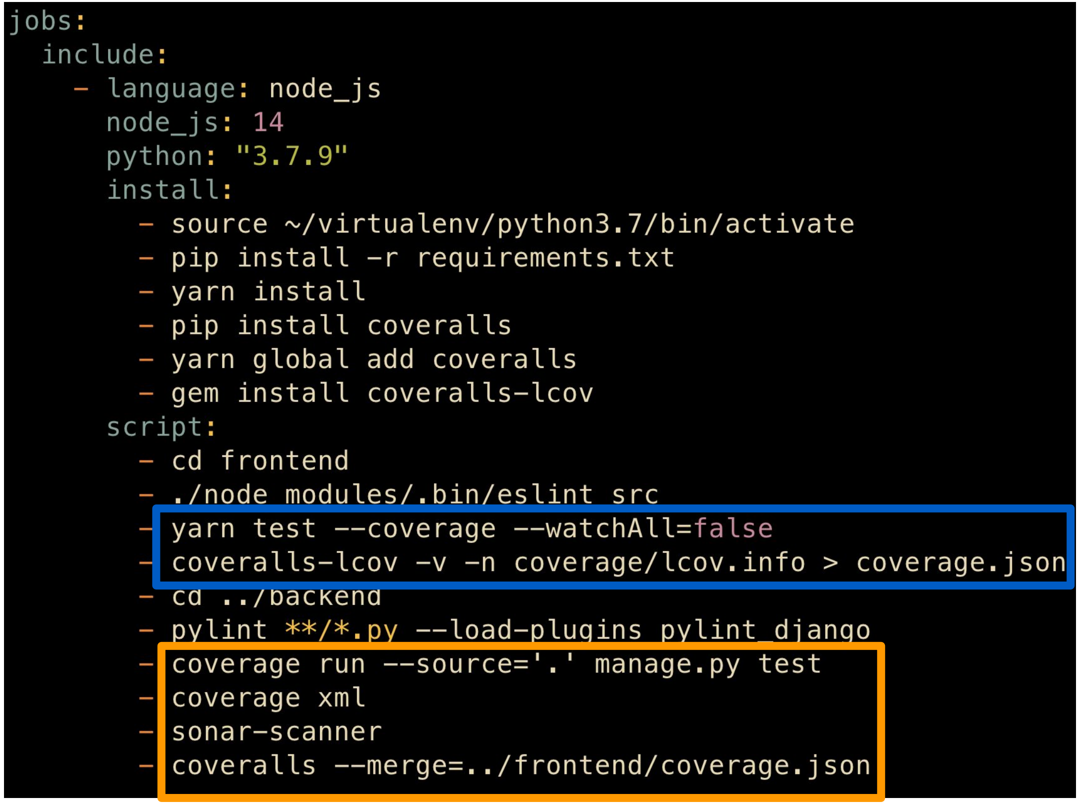
How to integrate CoverAlls

STEP 2. Configure Travis build job.

Run jest coverage command.
The coverage report will be generated under `coverage/lcov.info`.
Then `coveralls-lcov` leverages the report to create CoverAlls page.

Run coverage for Python projects.
Then, call `coveralls` command.
Make sure to pass the coverage report created by the JS project with the `merge` option.

- Please refer to the example config for more details.



```
jobs:
  include:
    - language: node_js
      node_js: 14
      python: "3.7.9"
      install:
        - source ~/virtualenv/python3.7/bin/activate
        - pip install -r requirements.txt
        - yarn install
        - pip install coveralls
        - yarn global add coveralls
        - gem install coveralls-lcov
      script:
        - cd frontend
        - ./node_modules/.bin/eslint src
        - yarn test --coverage --watchAll=false
        - coveralls-lcov -v -n coverage/lcov.info > coverage.json
        - cd ../backend
        - pylint **/*.py --load-plugins pylint_django
        - coverage run --source='.' manage.py test
        - coverage xml
        - sonar-scanner
        - coveralls --merge=../frontend/coverage.json
```

How to integrate CoverAlls

STEP 3. Add CoverAlls badge in the README.md. Copy and paste the following line.

```
[![Coverage Status](https://coveralls.io/repos/github/swsnu/swpp2021-teamX/badge.svg?branch=main)](https://coveralls.io/github/swsnu/swpp2021-teamX?branch=main)
```

- You can check the details by clicking the badge or by getting into the CoverAlls page (https://coveralls.io/github/swsnu).
- At the end of each sprint (starting from sprint3), your **coverage should be above 80%**, meaning both Travis and CoverAlls should work properly.
- There will be no team task for setting CoverAlls. Let TAs know if you have any problems with setting Travis and CoverAlls.

build
passing
quality gate
passed
coverage
87%

Team task for Mid Presentation

Today’s Team Task

- Create a new page in your team repository’s wiki of name ***MidPresentation***
  - Write short (less than 1 page) mid presentation demo plans