

POLITECNICO DI TORINO

Master's Degree in Engineering and Management



**Politecnico
di Torino**

Master's Degree Thesis

Design and Implementation of an Object-Oriented Python Library for Project Management

Supervisor:

Prof. Demagistris Paolo Eugenio

Candidate:

Chiemerie Ezechukwu

Academic Year 2021/2022

Abstract

When developing Project management backends, one often must plan for data types, ways to process the data and actions to take. This thesis explores how OOP methodologies benefit project management systems and software when the building blocks of the software or system are represented in terms of “self-sufficient” objects.

To improve the ease and speed with which applications are developed, there is the need for pre-developed libraries or framework. As such, a lot is abstracted from the library consumer, and they can come up with solutions quicker. A project management application is no exception to this and can benefit from a library that already implements or provides an interface for most project management objects and processes to be represented in the form of objects.

This thesis will produce a python library that models project management objects and processes that will be essential when developing a Python-based thesis project management application. For example, in a thesis setting, objects can be created to represent the project charter, the student involved, the professor and any other external stakeholders etc. These building blocks can then be aggregated, according to requirements, to form the core of the given project which should be self-sufficient (i.e., implement all the methods and attributes necessary). The benefits this will provide are many folds from appropriating the benefits of OOP to PM to reduced efforts in the development process due to code reusability to increased flexibility due to modularity and polymorphism.

Table of Contents

Abstract	1
1 Introduction	5
1.1 Object oriented project management	5
1.1.1 What is project management	5
1.1.2 What is a project	6
1.1.3 Object oriented project management	6
1.2 Object oriented programming	8
1.2.1 What is Object oriented programming	8
1.2.2 Building blocks of OOP	10
1.3 Principles of OOP	13
1.4 Libraries	20
1.5 Overview of the thesis	21
2 Implementation	22
2.1 Tools and technologies used	22
2.1.1 Python	22
2.1.2 Git and GitHub (Version Control)	23
2.1.3 GitHub Actions	25
2.2 Library Implementation	28
3 Testing	34
4 Conclusion	35

List of Figures

1.1	Class Stakeholder implemented in Professor and Student class	9
1.2	Cat class blueprint implementation	10
1.3	Cat class instance	11
1.4	class methods	12
1.5	Generic Dog class	14
1.6	Inheritance illustration	15
1.7	Child class instance	15
1.8	Method overriding	19
2.1	Components of GitHub Actions	26

Acronyms

O2PM Object oriented project management

OOP Object oriented programming

PM Project management

Chapter 1

Introduction

1.1 Object oriented project management

1.1.1 What is project management

Project management is the use of specific knowledge, skills, tools and techniques to deliver something of value to people [1].

Project management is the discipline of initiating, planning, executing, controlling and closing the work of a team to achieve specific goals and meet specific success criteria. A project is temporary in that it has a defined beginning and end in time, and therefore defined scope and resources [2].

1.1.2 What is a project

A project, on the other hand, is distinct in that it is not a routine action, but rather a specific group of operations aimed at achieving a single goal. As a result, a project team frequently consists of people who don't normally collaborate - perhaps from separate businesses and across several continents. Examples of projects can be the creation of software to improve a corporate process, the construction of a building or bridge, the relief effort following a natural disaster, expansion into a new geographical market.

1.1.3 Object oriented project management

Object oriented project management (O2PM) is based on concepts of Object oriented programming such as objects and attributes, classes and members. It consists of five major activities;

- finding classes and objects.
- identifying structures.
- identifying subjects.
- defining attributes.
- defining services.

Most importantly, O2PM is all about applying the object-oriented approach to project management [3].

Coming from the standpoint of O2PM, every aspect of a project is an object. In a typical project setting, the project manager constructs a project plan using a tool,

defines tasks, dates, and effort, and assigns them to team members; however there are possibly no adequate mechanisms to encapsulate work and define boundaries. This often will lead to accountability issues, challenges with effectively quantifying project milestones and producing inaccurate reports.

Many of these issues are addressed by O2PM. The object-oriented approach to project management is the focus of O2PM, and the key characteristics are as follows:

- **Encapsulation:** Team members can work well within defined bounds when work deliverables are encapsulated meaning that boundaries are well defined. Better control and, more crucially, accountability will result from this type of job encapsulation. It also aids in the easy identification of issues and their resolution.
- **Inheritance:** will specify the project's architecture, rules, standards, and processes, which must be "inherited" by all team members and their work deliverables.
- **Polymorphism:** Describes the concept that a singular object/interface can have several other implementations. For example, in a thesis project, the professor and student are different entities but both implement the same stakeholder class.
- **Communication:** This defines a known and standard medium through which each encapsulated work object communicates with each other.

1.2 Object oriented programming

1.2.1 What is Object oriented programming

Object oriented programming (OOP) is a programming paradigm that relies on the concept of classes and objects. It is used to structure a software program into simple, reusable pieces of code blueprints (called classes), which are used to create individual instances of objects. There are many object-oriented programming languages including JavaScript, C++, Java, and Python to name a few [4].

OOP allows programmers to handle software development as if they were dealing with actual real-world objects. People, in everyday life, have the knowledge and ability to do a variety of duties. Objects in OOP have fields to hold knowledge, state, and data, as well as the ability to execute numerous methods.

A class in OOP serves as a blueprint for creating more specific objects. They represent broad categories that share attributes. They enforce what attributes their instances can have but the values for any specific instance is set by that instance itself. For example, in project management, a stakeholder is a party who has an interest in a specific activity and can either affect or be affected by the said activity. Now there are several types of stakeholders but in the OOP world, they are all different implementation of the parent class/blueprint, stakeholder. In other words, any specific type of stakeholder, will be represented as an object, which itself is a specific example of the rather generic class, stakeholder and will have unique values to the properties defined in the class.

The above illustration visually depicts what has been explained so far. A Stakeholder class to contain all the properties a stakeholder must have and then from it are derived instances of a Stakeholder type object, Professor and Student to represent very specific stakeholders. Their attributes are isolated one from another and

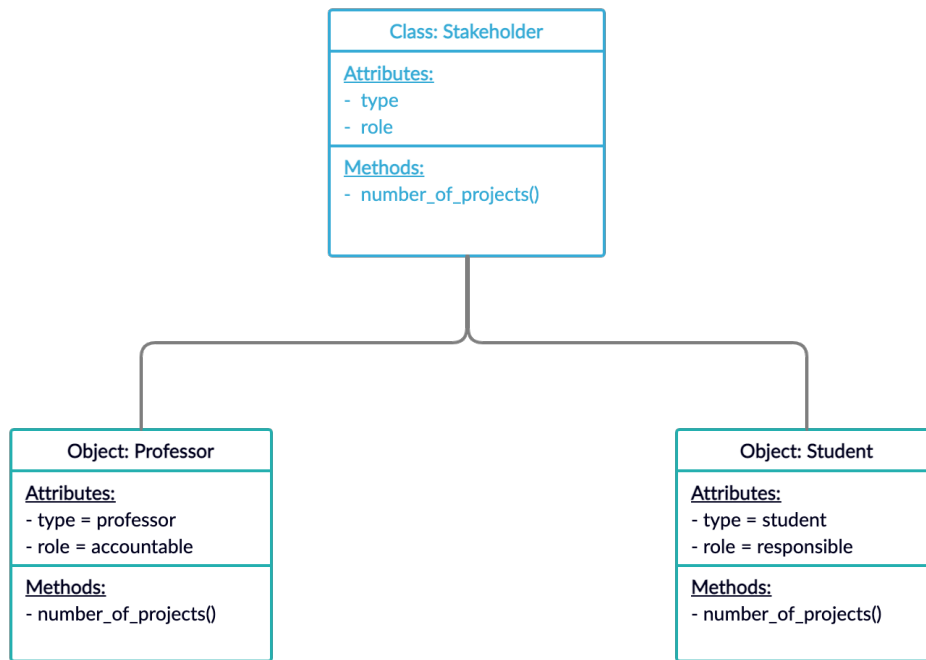


Figure 1.1: Class Stakeholder implemented in Professor and Student class

can be modified without affecting the original class or other objects. This means that the singular blueprint, Stakeholder can be reused to represent any number of stakeholders. The “role” field in the image above is according to RACI model which describes a responsibility assignment matrix.

The benefits that OOP presents are manyfold, from modelling complex things as simple reproducible structures to implementing objects that can be reused across programs to modifying the behavior of specific objects through polymorphism to their ability to protect information due to encapsulation. It will be beneficial to explain some basic terminologies in OOP in the next section.

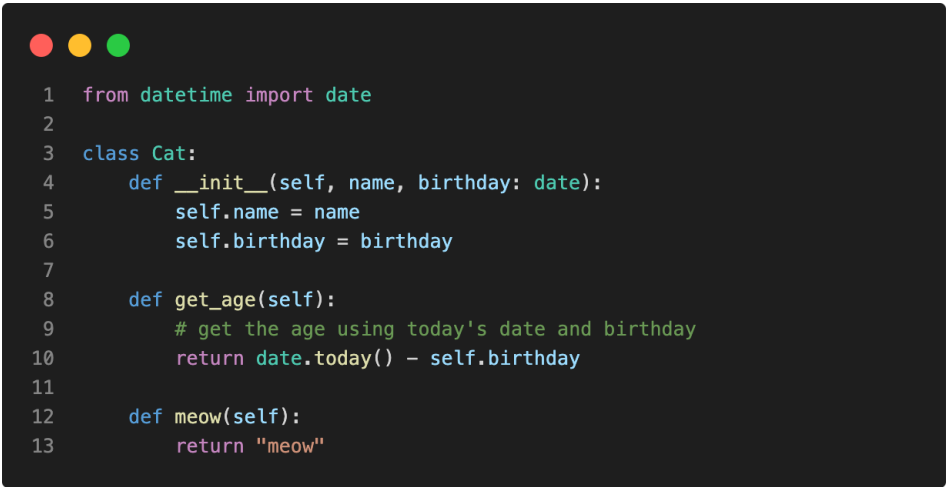
1.2.2 Building blocks of OOP

This section deals with elucidating the basic building blocks of an OOP program.

Classes

Classes are user-defined data types which is where the blueprint for the structure of attributes and methods are created. Individual objects can then be created or instantiated from this blueprint.

Classes have fields to hold attribute values and methods to model behavior. To illustrate, below is a code snippet in Python demonstrating how a generic class, `Cat` can be represented.




```
1  from datetime import date
2
3  class Cat:
4      def __init__(self, name, birthday: date):
5          self.name = name
6          self.birthday = birthday
7
8      def get_age(self):
9          # get the age using today's date and birthday
10         return date.today() - self.birthday
11
12     def meow(self):
13         return "meow"
```

Figure 1.2: Cat class blueprint implementation

The `Cat` class has attributes `name` and `birthday` and methods `get_age()` and `meow()`. With the knowledge that the class is only a blueprint for modelling a cat, a cat object can be instantiated from it to represent an individual real-world thing.

Objects

OOP is all about objects. Objects are instances of a class created with specific data. In code snippet, `milana` is an instance of the `Cat` class

A code editor window with a dark background and three colored window control buttons (red, yellow, green) in the top-left corner. The code is written in Python and defines a `Cat` class with methods `__init__`, `get_age`, and `meow`. It also creates two instances of the class: `milana` and `masha`.

```
1 from datetime import date
2
3 class Cat:
4     def __init__(self, name, birthday: date):
5         self.name = name
6         self.birthday = birthday
7
8     def get_age(self):
9         # get the age using today's date and birthday
10        return date.today() - self.birthday
11
12    def meow(self):
13        return "meow"
14
15    # a new instance of the Cat class and an individual cat named Milana
16    milana = Cat(name="Milana", birthday=date(2019, 5, 15))
17
18    # another instance of the Cat class and another individual cat named Masha
19    masha = Cat(name="Masha", birthday=date(2020, 7, 24))
```

Figure 1.3: Cat class instance

When the `Cat` class is called like on lines 16 and 19:

- A new object is created each time and stored in the variables `milana` and `masha` respectively.
- The constructor (`__init__`) method runs with the `name` and `birthday` arguments and assigns values.

Attributes

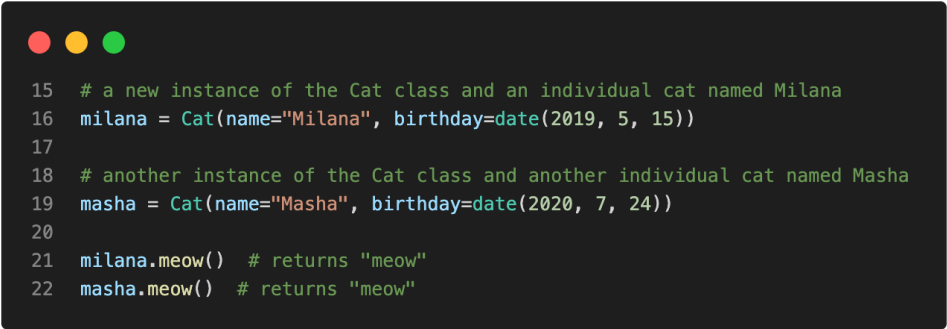
The information that is stored is referred to as attributes. The Class template defines the attributes. Individual objects have data stored in the attributes field when they are instantiated.

The data in the object's attributes fields define the object's state. The `birthday` attribute could define the state of an object, allowing software to manage cats of various ages differently.

Methods

The behaviors of objects are represented by methods. Methods carry out operations, such as returning information about an object or updating its data. The code for the method is specified in the class definition.

Objects can invoke the methods described in the class when they are instantiated. In the code snippet below, the `meow()` method is defined in the `Cat` class and the `meow()` method is called on either the `milana` or `masha` object on lines 21 and 22 respectively.



```
15 # a new instance of the Cat class and an individual cat named Milana
16 milana = Cat(name="Milana", birthday=date(2019, 5, 15))
17
18 # another instance of the Cat class and another individual cat named Masha
19 masha = Cat(name="Masha", birthday=date(2020, 7, 24))
20
21 milana.meow() # returns "meow"
22 masha.meow() # returns "meow"
```

Figure 1.4: class methods

Methods are often employed to modify, update or delete data but they don't have to. As a matter of fact, the `meow()` method does not modify any data because meowing does not modify any of the attributes of the `Cat` class, `name` or `birthday`. Methods are how programmers keep functionality encapsulated within an object and promote reusability. When debugging, its reusability comes in handy. If an error occurs, there is only one location to look for it and correct it rather than numerous.

1.3 Principles of OOP

There are four main pillars of OOP;

- **Inheritance:** Child classes inherit data and behavior from their parent classes.
- **Encapsulation:** The process of enclosing information in an object and exposing only a necessary subset.
- **Abstraction:** exposing high-level public methods for accessing an object.
- **Polymorphism:** many methods can do the same task.

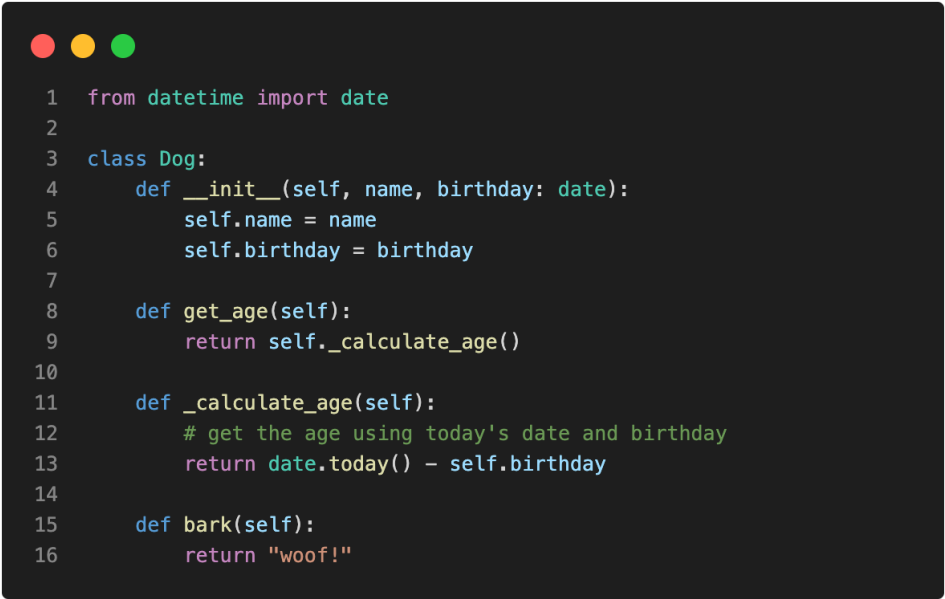
Inheritance

Objects typically have a lot in common. They have some similar logic. However, they are not comparable. Inheritance allows to reuse the common logic and extract the unique logic into another separate class. It implies that a (child) class can be created by deriving from another (parent) class. The child class derives all the parent class's fields and methods (common part) and can implement its own (unique part).

Herding dogs, for example, have a one-of-a-kind capacity to herd animals. To put

it another way, all herding dogs are dogs, but not all dogs are herding dogs. This distinction can be illustrated by establishing a `HerdingDog` child class from a parent `Dog` class, and then adding the unique `herd()` behavior to it.

Inheritance has the advantage of allowing programs to build a generic parent class and then create more specific child classes as needed. This simplifies the overall development because, rather than duplicating the `Dog` class's structure repeatedly, child classes can have automatic access to their parent class's features.



```
1  from datetime import date
2
3  class Dog:
4      def __init__(self, name, birthday: date):
5          self.name = name
6          self.birthday = birthday
7
8      def get_age(self):
9          return self._calculate_age()
10
11     def _calculate_age(self):
12         # get the age using today's date and birthday
13         return date.today() - self.birthday
14
15     def bark(self):
16         return "woof!"
```

Figure 1.5: Generic Dog class

In the code snippet in Figure 1.6, the child class `HerdingDog` inherits the method `bark` from the parent class `Dog` in Figure 1.5 and adds a new method `herd()`.

```
18 # Child class HerdingDog, inherits from parent class Dog
19 class HerdingDog(Dog):
20     def __init__(self, name, birthday: date):
21         super().__init__(name, birthday)
22
23     def herd(self):
24         # additional method for HerdingDog child class
25         return "Stay together!"
```

Figure 1.6: Inheritance illustration

The `bark()` method is not duplicated in the `HerdingDog` class; instead, it inherits the `bark()` method defined in the parent `Dog` class defined in Figure 1.5.

```
18 # Child class HerdingDog, inherits from parent class Dog
19 class HerdingDog(Dog):
20     def __init__(self, name, birthday: date):
21         super().__init__(name, birthday)
22
23     def herd(self):
24         # additional method for HerdingDog child class
25         return "Stay together!"
26
27 # instantiate a new HerdingDog object
28 jasper = HerdingDog("Jasper", date(2017, 12, 19))
29 jasper.bark()
```

Figure 1.7: Child class instance

When the code in Figure 1.3 calls `jasper.bark()` method, the `bark()` method traverses up the chain to the parent where the method has been defined.

Encapsulation

Encapsulation is the process of enclosing all critical information within an object and only presenting part of it to the outside world. Code within the class template defines attributes and behaviors.

Encapsulation hides the internal software code implementation inside a class and hides internal data of inside objects.

Encapsulation is achieved when each object keeps its state private, inside a class. Other objects don't have direct access to this state. Instead, they can only call a list of public methods.

As an example of encapsulation, consider an automobile. Public interfaces are how the automobile communicates with the outside world by employing blinkers to signal turns. Under the hood, on the other hand, the engine is hidden.

Security is improved via encapsulation. Private attributes and methods can be defined to prevent access from outside the class. Public methods and properties are used to obtain or alter data in an object.

Using the `Dog` class example, encapsulation helps here so that access to private information on other `Dog` instances is not possible. Consider the `get_age()` method in Figure 1.5, the calculation is hidden inside the `Dog` class and each instance of the class will use the `get_age()` method to access the `age` data. Since methods can also update an object's data, the developer controls what values can be changed through public methods.

Abstraction

Abstraction can be thought of as a natural extension of encapsulation. It refers to the user's interaction with only a subset of an object's attributes and operations. To access a complex object, abstraction use simplified, high-level tools. When employing abstraction, each object should only disclose a high-level mechanism for interacting with it. Internal implementation details should be hidden behind this mechanism. Using the automobile example again, you don't have to know the details of the inner workings of the engine to drive it. The driver utilizes only a few tools, such as the gas pedal, brake, steering wheel, and blinker. The driver is not privy to the engineering. A lot of elements must function together under the hood to make an automobile run but revealing that knowledge to the driver would be a risky distraction.

In the `Dog` class example in Figure 1.5, the calculation of the age has been abstracted away into a private method `_calculate_age()` and a simple public method `get_age()` is exposed as a way to get the age information

The benefits of abstraction are summarized below:

- Simple, high level user interfaces.
- Complex code is hidden.
- Security.
- Easier software maintenance.
- Updates to code will rarely change abstraction.

Polymorphism

Polymorphism refers to the design of items that have similar behavior. Objects can override shared parent behaviors with specific child behaviors through inheritance. Method overriding and method overloading are two ways that polymorphism allows the same method to perform various actions.

- **Method Overriding**

In method overriding, the child class will provide its own implementation of a method already in the parent class effectively overriding it. If for the `Dog` class, there's the need to create another child class `TrackingDog` which for some reason barks differently, the `bark()` method established in the parent class will have to be overridden like in the code snippet in Figure 1.8 to produce a different bark sound.

- **Method Overloading**

This kind of polymorphism happens at compile or code execution time. Methods may have the same name but behave differently based on the number of parameters passed to them.

```
1  from datetime import date
2
3  class Dog:
4      def __init__(self, name, birthday: date):
5          self.name = name
6          self.birthday = birthday
7
8      def get_age(self):
9          return self._calculate_age()
10
11     def _calculate_age(self):
12         # get the age using today's date and birthday
13         return date.today() - self.birthday
14
15     def bark(self):
16         return "woof!"
17
18 # Child class TrackingDog, inherits from parent class Dog
19 class TrackingDog(Dog):
20     def __init__(self, name, birthday: date):
21         super().__init__(name, birthday)
22
23     def track(self):
24         # additional method for TrackingDog child class
25         return "Searching...!"
26
27     def bark(self):
28         return "Found It!"
29
30 # instantiate a new TrackingDog object
31 buddy = TrackingDog("Buddy", date(2018, 8, 11))
32 buddy.bark() # returns "Found It!"
```

Figure 1.8: Method overriding

The benefits of Polymorphism are:

- Objects of different types can be passed through the same interface.
- Method overriding.
- Method overloading.

1.4 Libraries

A "library" is a collection of program parts that do common and/or specialized things that save the programmer from needing to "reinvent the wheel" when writing software. A class library is a pre-coded OOP template collection. It usually consists of functions to call and object classes you can instantiate [5].

Libraries encourage code reuse and decrease redundancy by providing implementation of repetitive jobs. Developing programs from the ground up can be a time-consuming and costly task. Class libraries include all necessary classes in a previously written coded format, making programming easier while also improving code quality. Customizing the class template is done in accordance with the programming requirements. To minimize programming language limitations, class libraries are regularly updated, and time verified for stability.

Programmers can decide to build everything from the ground up, but this is incredibly unsustainable and ill-advised for the reasons below;

- **Expertise in the domain covered by the library:** Authors are usually experts in the domain covered by the library. This will ensure that users of the library get the best implementation possible.
- **Stability:** Libraries have the benefit of being used by hundreds, if not thousands, of developers around the world. Most of the early issues have already been encountered and resolved by authors.
- **Knowledge:** By having to use code pre-written by others, developers gain knowledge in the process. Many well-known libraries are built by top developers and serve as excellent examples of solid coding and design.
- **Finances:** For commercial libraries, the equivalent of hundreds, if not thousands, of man days of work can be got for free thanks to open source.

1.5 Overview of the thesis

To be done

Chapter 2

Implementation

The implementation of this thesis project focuses on the library to be developed with the Python programming language.

2.1 Tools and technologies used

2.1.1 Python

Python is a programming language that is commonly used to create websites (server-side) and applications, automate operations, and perform data analysis. Python is a general-purpose programming language, which means that it can be used to develop a wide range of applications and isn't specialized for any particular problem. Because of its versatility and beginner-friendliness, it has become one of the most widely used programming languages today.

Python was designed for readability and has some similarities to the English language

with influence from mathematics. Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly braces for this purpose.

Python is popular for a variety of reasons from its simple syntax that mimics natural language to its versatility to the large and incredibly active community that contributes to Python's pool of modules and libraries.

2.1.2 Git and GitHub (Version Control)

Git

From web developers to app developers, Git is useful to anyone who writes code or track changes to files. Git is the most used version control system. It keeps track of the changes you make to files so you can see what has been done and go back to previous versions if you need to. Git also facilitates cooperation by allowing several people's modifications to be merged into a single source. Initialize Git locally and your files and their history are stored on your computer. Git repositories contains all the project files and the entire revision history. It can be created by running the `git init` command in the location you wish to create this repository. This creates a `.git` subfolder, which contains all of the Git metadata for tracking changes.

GitHub

GitHub, on the other hand, is a Git hosting repository that makes it easier for developers to work together. Developers can work on the same code repository without conflicts using features like pull requests, code review, and issues (ticketing system).

Developers can work on separate branches of the same repository, commit code changes to store them, create a pull request to the main branch, debate and peer-review code changes before merging the pull request.

A repository is sometimes known as a repo. Repos are Git projects that contain files and directories associated with a development project, as well as the revision history of each file. Within a repository, new branches can be created, pull requests can be made, and merges can take place. The changelog displays the history of file changes in the repository.

Git and GitHub has been employed to store and version all code modifications for this project.

Why use Git

- **Simplifies the process of contributing to open source:** GitHub is used by nearly every open-source project to administer their project. If your project is open source, GitHub is free to use, and it contains a wiki and issue tracker that makes it simple to provide more detailed documentation and receive feedback. To contribute, simply fork a project, make your changes, and submit a pull request via the GitHub web interface.
- **Documentation:** You make it easy to get quality documentation by using GitHub. They include articles on practically every issue related to git that you can think of in their help section and tutorials.
- **Showcase your work:** Do you want to attract recruiters as a developer? The finest tool you can use for this is GitHub. Most firms now check at GitHub profiles while looking for fresh hires for their projects. Even if you did not attend a prestigious university or college, if your profile is available, you will have a better chance of being hired.

- **Markdown:** Markdown allows you to write styled texts using a simple text editor. Everything on GitHub is written in Markdown, including the issue tracker, user comments, and everything else. With so many other computer languages to master in order to build up projects, having your content inputted in a format without having to learn yet another system is a huge benefit.
- **Repository** This has previously been noted, but it's worth repeating: GitHub is a repository. This means that it is possible for your work to be seen by the general audience. Furthermore, GitHub is one of the largest coding communities in the world right now, giving your project a lot of exposure.
- **Keep track of changes in your code over time:** It's difficult to maintain track of modifications when numerous people work on a project—who modified what, when, and where those files are stored. This problem is solved by GitHub, which keeps track of all the modifications that have been pushed to the repository. You can keep a version history of your code, much like you can with Microsoft Word or Google Drive, so that earlier versions aren't lost with each iteration.
- **Options for integration:** GitHub integrates with popular cloud platforms like Amazon and Google Cloud, as well as feedback tracking services like Code Climate, and can highlight syntax in over 200 different programming languages.

2.1.3 GitHub Actions

For continuous integration (CI), GitHub Actions has been employed.

GitHub Actions is a continuous integration and continuous delivery (CI/CD) platform that allows you to automate your build, test, and deployment pipeline. It allows for creating workflows that build and test your source code on events you configure

happening in your repository like pull requests or even merge pull requests to production [6].

GitHub Actions extends beyond DevOps by allowing you to execute workflows in response to other events in your repository. For example, anytime someone opens a new issue in your repository, you may execute a workflow to automatically add the required labels.

You can execute your workflows on GitHub’s virtual machines running Linux, Windows, and macOS, or you can host your own self-hosted runners in your own data center or cloud infrastructure.

The components of GitHub Actions

When an event occurs in your repository, such as when a pull request is opened or an issue is raised, you may set up a GitHub Actions workflow to be executed. Your workflow includes one or more jobs that can execute sequentially or concurrently. Each task contains one or more steps that either run a script you create or run an action, which is a reusable addition that can streamline your workflow.

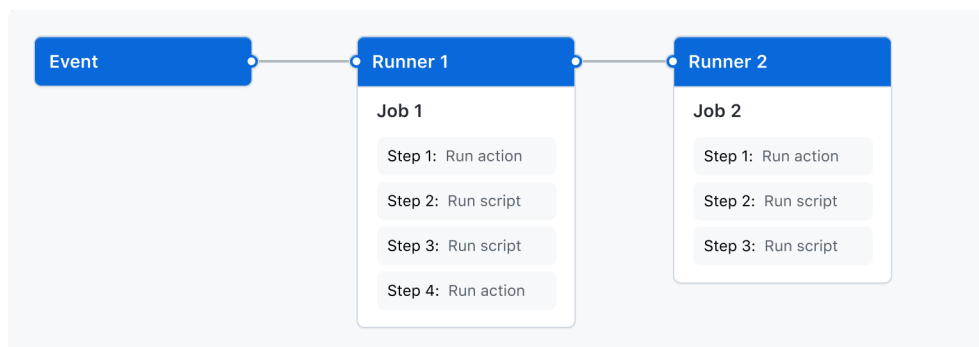


Figure 2.1: Components of GitHub Actions [6]

Workflows

A workflow is a programmable automated procedure that executes one or more tasks. Workflows are specified in a YAML file that is checked into your repository. They can be triggered by an event in your repository, manually, or on a set timetable.

A repository can contain numerous workflows, each of which can carry out a separate set of tasks. For example, you may have one workflow to build and test pull requests, another to deploy your application whenever a release is made, and still another to add a label if someone reports a new issue.

Events

A workflow run is triggered by an event, which is a specific activity in a repository. For example, GitHub can be an activity when someone creates a pull request, opens an issue, or pushes a commit to a repository. A workflow can also be started manually on a timetable, or by triggering a REST API.

Jobs

A job is a series of workflow stages that all run on the same runner. Each step consists of either a shell script that will be run or an action that will be performed. The steps are carried out in a sequential order and are interdependent. Because each step is run on the same runner, data can be shared from one step to the next. A step that builds your application, for example, could be followed by a step that tests the application that was built.

You can configure a job's dependencies with other jobs; by default, jobs execute in

parallel and have no dependencies. When a job becomes reliant on another job, it will not run until the dependent job has completed. You might have many build tasks for different architectures with no dependencies and a packaging job that is dependent on those jobs, for example. The build jobs will run in parallel, and the packaging job will begin once they have all completed successfully.

Actions

An action is a GitHub Actions platform custom application that performs a sophisticated but regularly repeated activity. To assist in limiting the amount of repetitive code in your workflow files, use an action. An action can grab your git repository from GitHub, configure your build environment's toolchain, or set up your cloud provider's credentials.

Runners

When your workflows are triggered, a runner is a server that executes them. Each runner is limited to one job at a time. To run your workflows, GitHub provides Ubuntu Linux, Microsoft Windows, and macOS runners; each workflow run takes place in a new, freshly provisioned virtual machine. You can host your own runners if you need a different operating system or a specific hardware setup.

2.2 Library Implementation

In this section, the implementation of the library is showcased and explained. The library will have 8 distinct project management objects each represented as a class in the library namely;

- `ProjectCharter` class
- `Stakeholder` class
- `Deliverable` class
- `ProjectGovernance` class
- `Risk` class
- `WBSItem` class
- `WBS` class
- `Project` class

The `ProjectCharter` class

A project charter is a formal, typically short document that describes your project in its entirety — including what the objectives are, how it will be carried out, and who the stakeholders are. It is a crucial ingredient in planning the project because it is used throughout the project lifecycle [7].

The `ProjectCharter` class presents a few methods and properties.

The class's methods and signatures are represented below in a very simplified interface to abstract away the implementation.

```
class ProjectCharter(project_title = None):  
    # properties  
    property executive_summary: str  
    property project_stakeholders: list  
    property project_title: str
```

```
# methods
method add_stakeholder(self, stakeholder)

# static methods
static is_class_stakeholder(obj) -> bool
```

The constructor is called anytime a new `ProjectCharter` instance is created. It takes an optional argument, `project_title` and goes ahead to initialize an empty list `_project_stakeholders` private property that will hold and track `Stakeholder` objects

The `executive_summary` property is a string field that holds summarized information about project.

There's a restriction to the `ProjectCharter` class in that only objects initialized from the `Stakeholder` class can be added to the `project_stakeholders` list. Adding a stakeholder must be done through the method `add_stakeholder`. A static method `is_class_stakeholder` is available solely for verifying objects before they are accepted to the stakeholder list. The static method simple returns a boolean `True` or `False` and is used in the method `add_stakeholder` like below

```
def add_stakeholder(self, stakeholder):
    assert self.is_class_stakeholder(stakeholder)
    if isinstance(stakeholder, (list, tuple)):
        for s in stakeholder:
            self.__finalize_add_stakeholder(s)
    else:
        self.__finalize_add_stakeholder(stakeholder)
```

```
@staticmethod
def is_class_stakeholder(obj) -> bool:
    """
    Checks that the stakeholder about to be added to the project
    is instance of the Stakeholder class
    """
    if isinstance(obj, list):
        return all(isinstance(s, Stakeholder) for s in obj)
    return isinstance(obj, Stakeholder)
```

The Stakeholder class

This class provides a template for modelling stakeholder objects. Its methods, properties and signatures are below in a simplified interface.

```
class Stakeholder(first_name, last_name, email):
    # methods
    method add_to_involved_projects(self, value)
    method get_full_name(self) -> str
    method get_number_of_projects(self) -> int

    # properties
    property projects_involved: List[Project]
    property responsibility: str
    property role
    property stakeholder_title
    property stakeholder_type
```

The method `add_to_involved_projects` is designed to be used only by the `__finalize_add_stakeholder()` private method defined in the `ProjectCharter` class. It is simply used to add new `ProjectCharter` instances to the list of projects the stakeholder in question is involved with.

`get_number_of_projects()` simply returns an integer depicting the number of projects that stakeholder instance is involved with at any point in time.

The property `projects_involved` returns a list of objects of the `ProjectCharter` charter class readily available should the library consumer require it.

For the `role` property of the various stakeholders, a simple enum following the RACI responsibility models is defined.

```
class RoleEnum(Enum):
    """
    * R = Responsible (Those who do the work to complete the task)
    * A = Accountable (also approver or final approving authority)
    * C = Consulted (Those whose opinions are sought)
    * I = Informed (Those who are kept up-to-date on progress)
    """

    RESPONSIBLE = "R"
    ACCOUNTABLE = "A"
    CONSULTED = "C"
    INFORMED = "I"
    UNSET = "UNSET"
```

The Deliverable class

Deliverables are very important to the success of any project and this library provides an implementation of a deliverable model. The methods and properties it provides are showcased in the simple interface below

```
class Deliverable(doc_title):
    # methods
    method mark_as_accepted(self)
    method mark_as_rejected(self)
```

```
method set_reviewer(self, value)

# properties
property accepted: bool
property reviewer
property acceptance_criteria
property description
property expected_result
```

The `mark_as_accepted` and `mark_as_rejected` methods serve to set the `accepted` property to `True` or `False` depending on whether the deliverable has been accepted or not. `set_reviewer` sets the `reviewer` property to the "responsible" who is accountable for the deliverable. `acceptance_criteria` property serves to define what determines when a deliverable can be accepted or not.

The ProjectGovernance class

The Risk class

The WBSItem class

The WBS class

The Project class

Chapter 3

Testing

Chapter 4

Conclusion

References

- [1] Project Management Institute. What is project management?
<https://www.pmi.org/about/learn-about-pmi/what-is-project-management>.
- [2] Harelimana JB. *Basic Concepts of Project Management*. Austin Publishing Group, 2017.
- [3] K.R. Chandrashekar. Object-oriented project management (o2pm) objectizing work. *Project Management Institute*, 2010.
- [4] Erin Doherty. What is object-oriented programming? oop explained in depth.
<https://www.educative.io/blog/object-oriented-programming>.
- [5] Steven Parker. What exactly is a library in programming?
<https://teamtreehouse.com/community/what-exactly-is-a-library-in-programming>.
- [6] GitHub Docs. Understanding github actions.
<https://docs.github.com/en/actions/learn-github-actions/understanding-github-actions>.
- [7] Wrike. What is a project charter in project management?
<https://www.wrike.com/project-management-guide/faq/what-is-a-project-charter-in-project-management/>.