Homework

by

Christopher Arias

carias1@stevens.edu

September 20, 2024

Homework

Christopher Arias
carias1@stevens.edu

This document provides the requirements and design details of the PROJECT. The following table (Table 1) should be updated by authors whenever major changes are made to the architecture design or new components are added. Add updates to the top of the table. Most recent changes to the document should be seen first and the oldest last.

Table 1: Document Update History

Date	**Updates**
9/19/2024	DDM: • Removed all unnecessary pages from template • Added new chapter for Homework Two (Chapter 4) • Inserted all necessary figures into the Homework Two chapter (Chapter 4)

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

– Christopher Arias

This project is for completing the homework in SSW 345.

Chapter 2

Team

– Christopher Arias

Hello, I am Christopher Arias. I am a 4/4 computer science major with a minor in software engineering. I aspire to work for a big tech company like Google or Amazon.

Chapter 3

Git Homework

– Christopher Arias

Link to repo

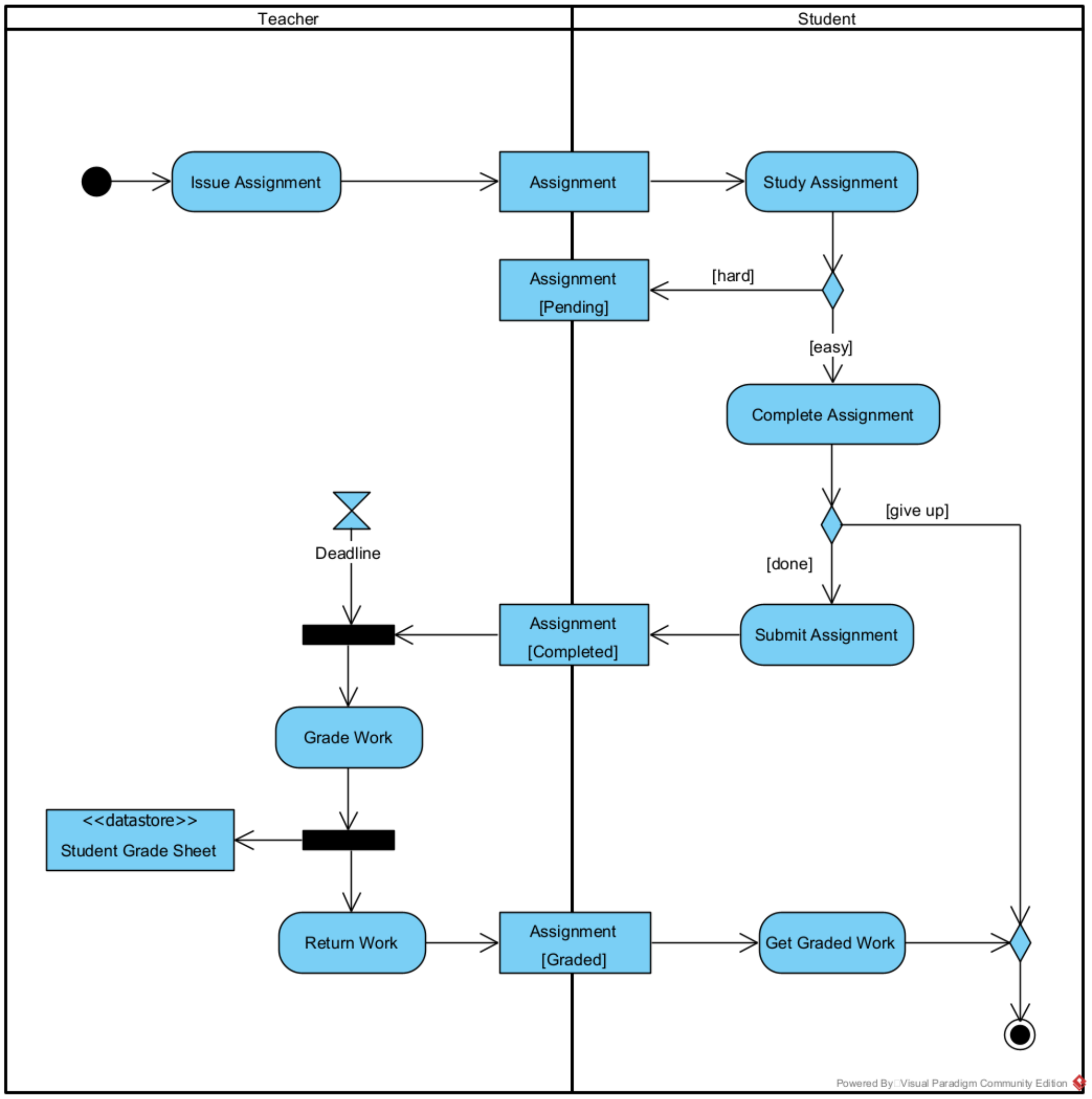


Figure 3.1: First test UML diagram from visual paradigm

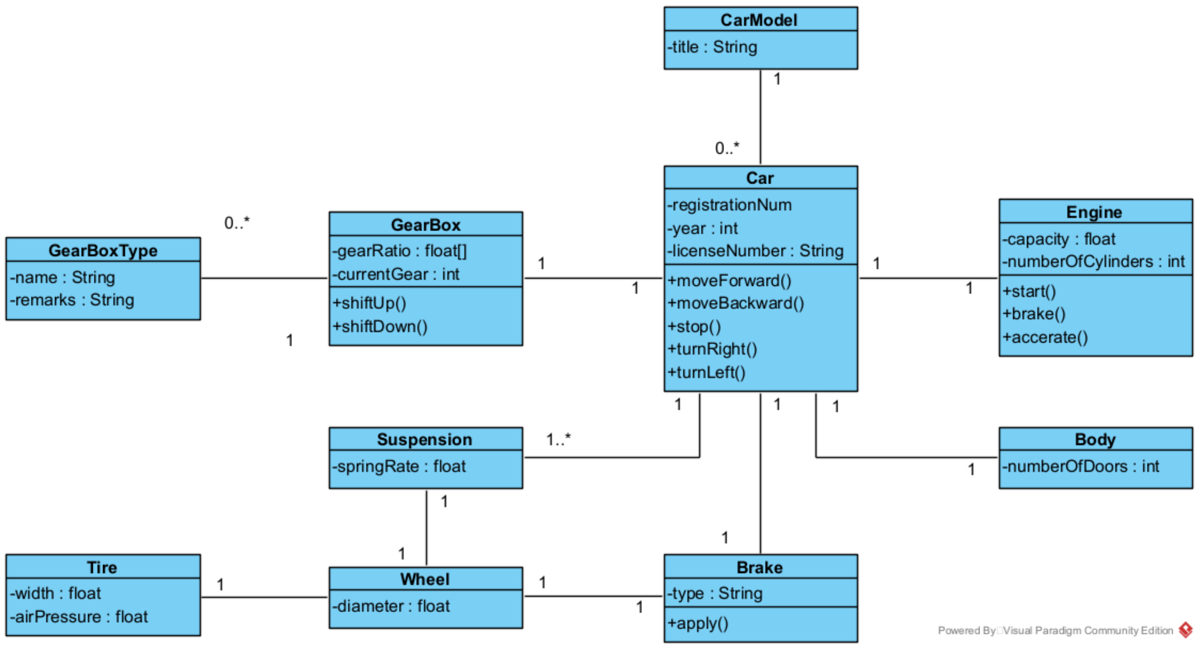
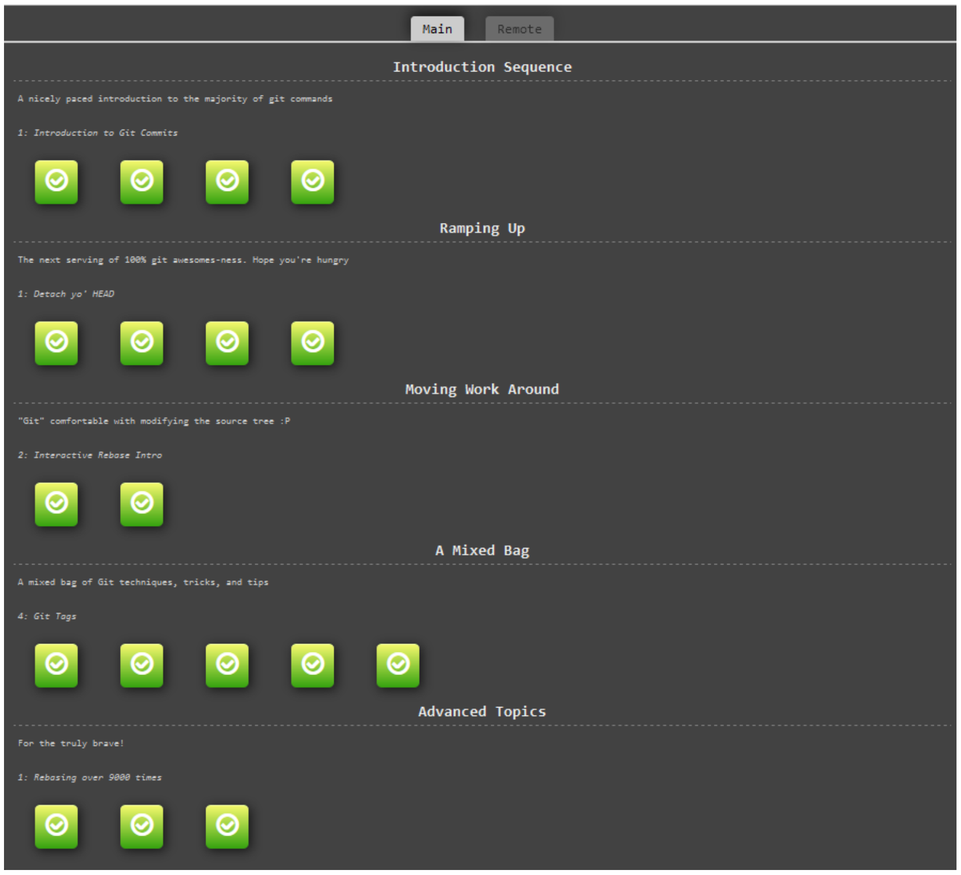


Figure 3.2: Second test UML diagram from visual paradigm



Introduction Sequence

A nicely paced introduction to the majority of git commands

1: Introduction to Git Commits

Ramping Up

The next serving of 100% git awesomes-ness. Hope you're hungry

1: Detach yo' HEAD

Moving Work Around

"Git" comfortable with modifying the source tree :P

2: Interactive Rebase Intro

A Mixed Bag

A mixed bag of Git techniques, tricks, and tips

4: Git Tags

Advanced Topics

For the truly brave!

1: Rebasing over 9000 times

Figure 3.3: Demonstrating completion of part 2 of this assignment

Chapter 4

UML Class Modeling

– Christopher Arias

4.1 Exercise 2.1

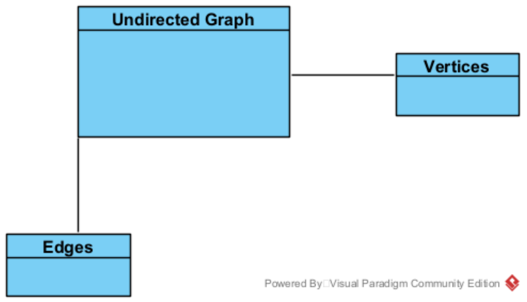


Figure 4.1: UML Diagram describing the undirected graph from Exercise 2.1

4.2 Exercise 2.2

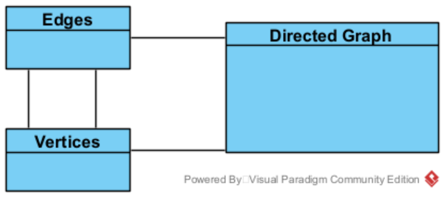


Figure 4.2: UML Diagram describing the directed graph from Exercise 2.2

4.3 Exercise 2.3

- Window to Canvas Association:

  A single Window can be linked to multiple Canvas objects. A Window is responsible for displaying content, and it can host multiple Canvas objects. Each Canvas serves as an individual drawing surface within the Window.

- Canvas to Shape Association:

  A Canvas can include zero or more Shape objects, with each Shape associated with only one Canvas. Canvas acts as a container for graphical elements like Shapes (such as Lines, ClosedShapes, etc.), which are drawn within the Canvas.

- Shape to Polygon, Line, ClosedShape, Ellipse Association:

  The Shape class can be specialized into subtypes such as Line, Polygon, ClosedShape, and Ellipse. Shape serves as a generic class for specific graphical elements like Lines, Polygons, and Ellipses, which inherit Shape's properties and override its draw functionality.

- Canvas to ScrollingCanvas, TextWindow Association:

  A Canvas can be specialized into either a ScrollingCanvas or a TextWindow. The Canvas can either represent a simple drawing area like a TextWindow or one with additional scrolling functionality like a ScrollingCanvas, allowing for varied content management.

- Panel to PanelItem Association:

  A Panel can contain either zero or one PanelItem, and each PanelItem is linked to exactly one Panel. The Panel serves as a container for a single PanelItem, which represents interactive elements like buttons or choice items.

- PanelItem to Button, ChoiceItem, TextItem Association:

  The PanelItem class can be specialized into different interactive types like Button, ChoiceItem, or TextItem. PanelItem generalizes elements that users can interact with, such as buttons for clicking, choice items for selection, and text items for input.

- ChoiceItem to ChoiceEntry Association:

  A ChoiceItem contains a set of ChoiceEntry objects. Each ChoiceItem can be associated with multiple ChoiceEntries, representing the different selectable options within the ChoiceItem.

- PanelItem to Event Association:

  A PanelItem is linked to one or more Event objects. PanelItems trigger Events based on user interactions, such as clicking a button or entering text, which result in specific actions being initiated.

- Event to PanelItem Association:

  Each Event is associated with one specific PanelItem. Events are directly triggered by interactions with specific PanelItems, ensuring the right action is executed when a user interacts with a given PanelItem.

- Polygon to Point Association:

  A Polygon is made up of multiple Point objects in a specific order. Polygons are defined by an ordered collection of Points, which represent the vertices of the polygon, forming its geometric structure.

4.4 Exercise 2.4

- Customer to MailingAddress Assocation:

  A Customer can have several MailingAddresses, indicating a one-to-many relationship. The MailingAddress contains relevant details such as the address and phone number associated with the customer.

- Customer to CreditCardAccount Assocation:

  A Customer can possess multiple CreditCardAccounts, but each account is tied to only one Customer. This relationship is represented by the ”accountHolder” attribute, with each CreditCardAccount storing information such as maximum credit, current balance, and the statement date.

- CreditCardAccount to MailingAddress Assocation:

  A CreditCardAccount is connected to exactly one MailingAddress. This association reflects the mailing address linked to the account for billing and correspondence.

- CreditCardAccount to Institution Assocation:

  Each CreditCardAccount is related to one Institution. The Institution maintains details about the bank or organization responsible for managing the account, including its name, address, and phone number.

- CreditCardAccount to Statement Assocation:

  A CreditCardAccount can be associated with multiple Statements. Each Statement provides detailed information such as the payment due date, finance charge, and minimum payment, and is linked to a single CreditCardAccount.

- Statement to Transaction Assocation:

  A Statement may include multiple Transactions. Each Transaction contains specific details like the transaction date, explanation, and amount, and is associated with only one Statement.

- Transaction to CashAdvance, Interest, Purchase, Fee, Adjustment Assocation:

  A Transaction can represent different types, including CashAdvance, Interest, Purchase, Fee, or Adjustment. Each transaction type refers to a specific kind of financial operation performed on the account.

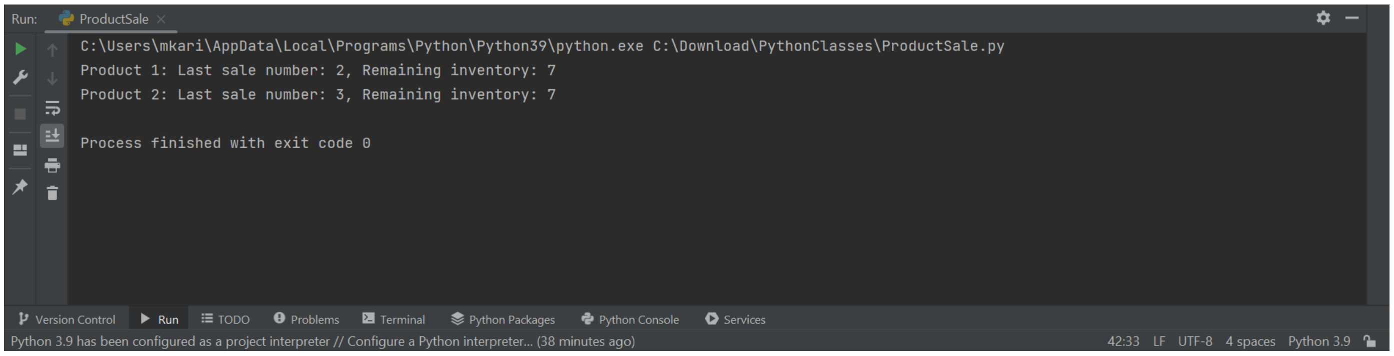
- Purchase to Merchant Assocation:

  A Purchase transaction is linked to a Merchant. This relationship indicates that every purchase transaction may involve details about the Merchant from whom the purchase was made.

- Fee to FeeType Assocation:

  A Fee transaction is related to a particular FeeType. The FeeType defines the nature of the fee applied in a transaction, such as late fees or service charges.

4.5 Code Related Figures and Images



```
C:\Users\mkari\AppData\Local\Programs\Python\Python39\python.exe C:\Download\PythonClasses\ProductSale.py
Product 1: Last sale number: 2, Remaining inventory: 7
Product 2: Last sale number: 3, Remaining inventory: 7

Process finished with exit code 0
```

Python 3.9 has been configured as a project interpreter // Configure a Python interpreter... (38 minutes ago) 42:33 LF UTF-8 4 spaces Python 3.9

Figure 4.3: Output of the modified ProductSale.py code

ProductSale.py

```python
# needed for forward reference of Sale in Product,
# since Sale is not yet defined.
from __future__ import annotations
from typing import List

# forward reference used for class Sale
class Product:
    __lastSale: Sale = None
    __inventory: int = 0

    def __init__(self, sale: Sale, inventory: int):
        self.__lastSale = sale
        self.__inventory = inventory

    def setLastSale(self, lastSale: Sale):
        self.__lastSale = lastSale

    @property
    def getLastSale(self) -> Sale:
        return self.__lastSale

    @property
    def getInventory(self) -> int:
        return self.__inventory

    def reduceInventory(self, quantity: int):
        if quantity > self.__inventory:
            raise ValueError("Not enough inventory to fulfill sale.")
        self.__inventory -= quantity

    def __getitem__(self, item):
        return self

# no forward reference needed since Product is defined
class Sale:
    __saleTimes = 0
    __productSold: List[Product] = None
    __saleNumber: int = 0

    def __init__(self, product: List[Product], quantities: List[int]):
        Sale.__saleTimes += 1
        self.__product = product
        self.__saleNumber = Sale.__saleTimes

        for index, product in enumerate(product):
            if index < len(quantities):
                product.reduceInventory(quantities[index])
            product.setLastSale(self)

    def setProductsSold(self, productSold: List[Product]):
        self.__productSold = productSold

    @property
    def getSaleNumber(self) -> int:
        return self.__saleNumber


productOne = Product(sale=None, inventory=10)
productTwo = Product(sale=None, inventory=15)

saleOne = Sale([productOne, productTwo], [2, 3])
saleTwo = Sale([productOne], [1])
saleThree = Sale([productTwo], [5])

print(f"Product 1: Last sale number: {productOne.getLastSale.getSaleNumber},
    Remaining inventory: {productOne.getInventory}")
print(f"Product 2: Last sale number: {productTwo.getLastSale.getSaleNumber},
    Remaining inventory: {productTwo.getInventory}")
```

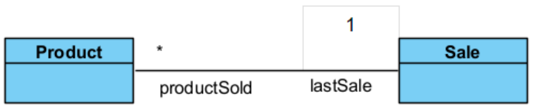


Figure 4.4: Updated UML diagram for code

Bibliography

Index