Homework Manual

by

Kyle Ponte

kponte@stevens.edu

November 22, 2024

© Kyle Ponte
kponte@stevens.edu
ALL RIGHTS RESERVED

Homework Manual

Kyle Ponte
kponte@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**
08/22/2023	DDM: • Updated dsnManual.tex with *newcommand(s)*{} for easier references of requirements, figures, and other labels.
10/25/2023	DDM: • Added chapters on use cases (Chapter **??**) and user stories (Chapter **??**).
10/11/2023	DDM: • Added chapters on requirements (Chapter **??**) and glossary.
09/18/2023	DDM: • Added chapter on development plan (Chapter **??**).
9/12/2024	Homework 1: • Added chapter called Team (Chapter 1) and a chapter called Git Homework (Chapter 2).
9/19/2024	Homework 2: • Added chapter called UML Class Modeling (Chapter 3).
10/10/2024	Homework 4: • Added chapter called Software Execution Model (Chapter 4).

Table 1: Document Update History

Date	Updates
11/1/2024	Exam One Extra Credit: • Added chapter called Exam One Extra Credit (Chapter 5).
11/1/2024	Decorator Pattern: • Added chapter called Decorator Pattern (Chapter 6).

Table of Contents

1 Team
– Kyle Ponte **1**

2 Git Homework
– Kyle Ponte **2**

3 UML Class Modeling
– Kyle Ponte **4**
3.1 Exercise 2.1 . . . 4
3.2 Exercise 2.2 . . . 5
3.3 Exercise 2.3 . . . 5
3.4 Exercise 2.4 . . . 8
3.5 Class - to - Code Modeling . . . 10
3.6 Code Results . . . 11

4 Software Execution Model
– Kyle Ponte **13**
4.1 Exercise 5.1 . . . 13
4.2 Exercise 5.2 . . . 14
4.3 Exercise 5.3 . . . 15
4.4 Link to the Updated Github Repository . . . 16

5 Exam One Extra Work
– Kyle Ponte **18**
5.1 Problem 5.6 . . . 18
5.1.1 Human.py Code . . . 18
5.1.2 PairGame.py code . . . 18
5.1.3 TeacherStudentGame.py (code driver) code . . . 19

6 Decorator Pattern
– Kyle Ponte **20**
6.1 Exercise 14.1 . . . 20
6.1.1 MargheritaPizza file code . . . 20

6.1.2 MushroomDecorator file code 20
6.1.3 PepperoniDecorator file code 21
6.1.4 Output Results 21
6.1.5 UML Diagram 21

Bibliography **23**

List of Tables

1 Document Update History iii
1 Document Update History iv

List of Figures

1.1 Example Diagram 1 1
1.2 Example Diagram 2 1

2.1 Part II HW 3

3.1 Sample Undirected Graph 4
3.2 Sample Directed Graph 5
3.3 Code Results 11
3.4 UML Diagram for Product and Sale 12

4.1 eStore Sequence Diagram 15
4.2 eStore Software Execution Diagram 17

6.1 Output Results 21
6.2 UML Diagram 22

Chapter 1

Team

– Kyle Ponte

My name is Kyle Ponte and I am a software engineer at Stevens Institute of Technology. I'm from Bloomfield, New Jersey, and I love both playing and watching basketball. I am also very fond of traveling, and so far have ventured to places in South and Central America like Peru and the Dominican Republic. In terms of other hobbies, I also enjoy listening to music and going to the gym.

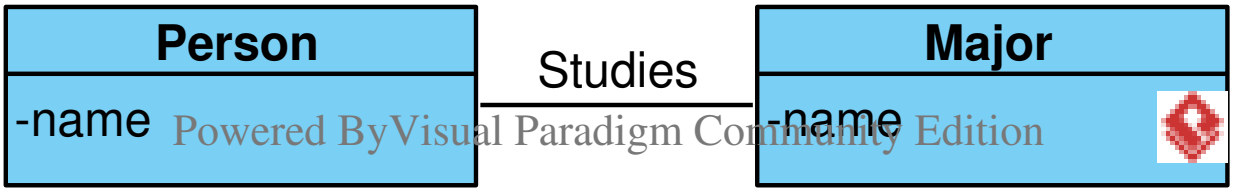

Figure 1.1: Example Diagram 1

Figure 1.2: Example Diagram 2

Chapter 2

Git Homework

– Kyle Ponte

Part II:

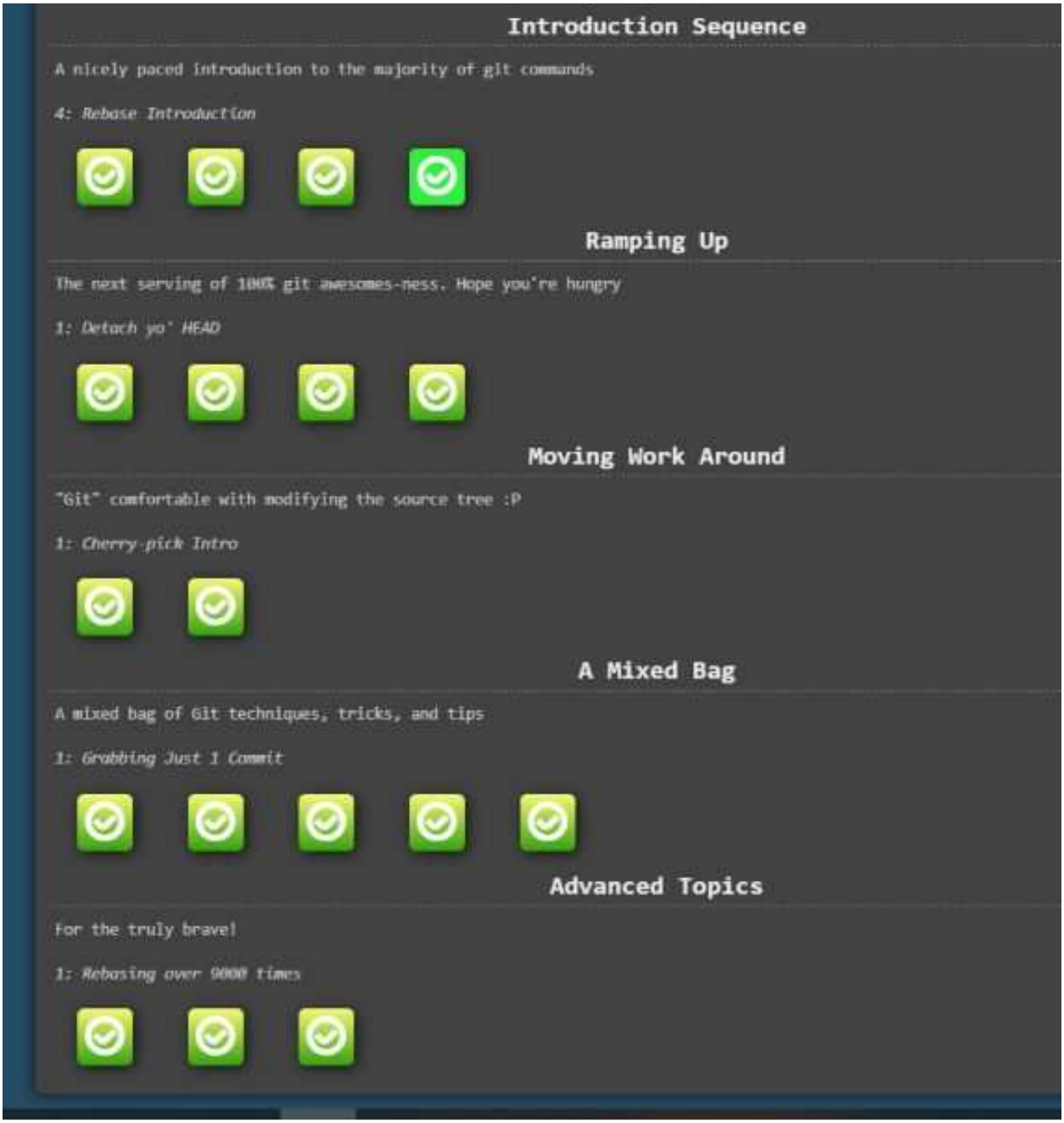

Figure 2.1: Part II HW

Chapter 3

UML Class Modeling

– Kyle Ponte

3.1 Exercise 2.1

Figure 3.1: Sample Undirected Graph

3.2 Exercise 2.2

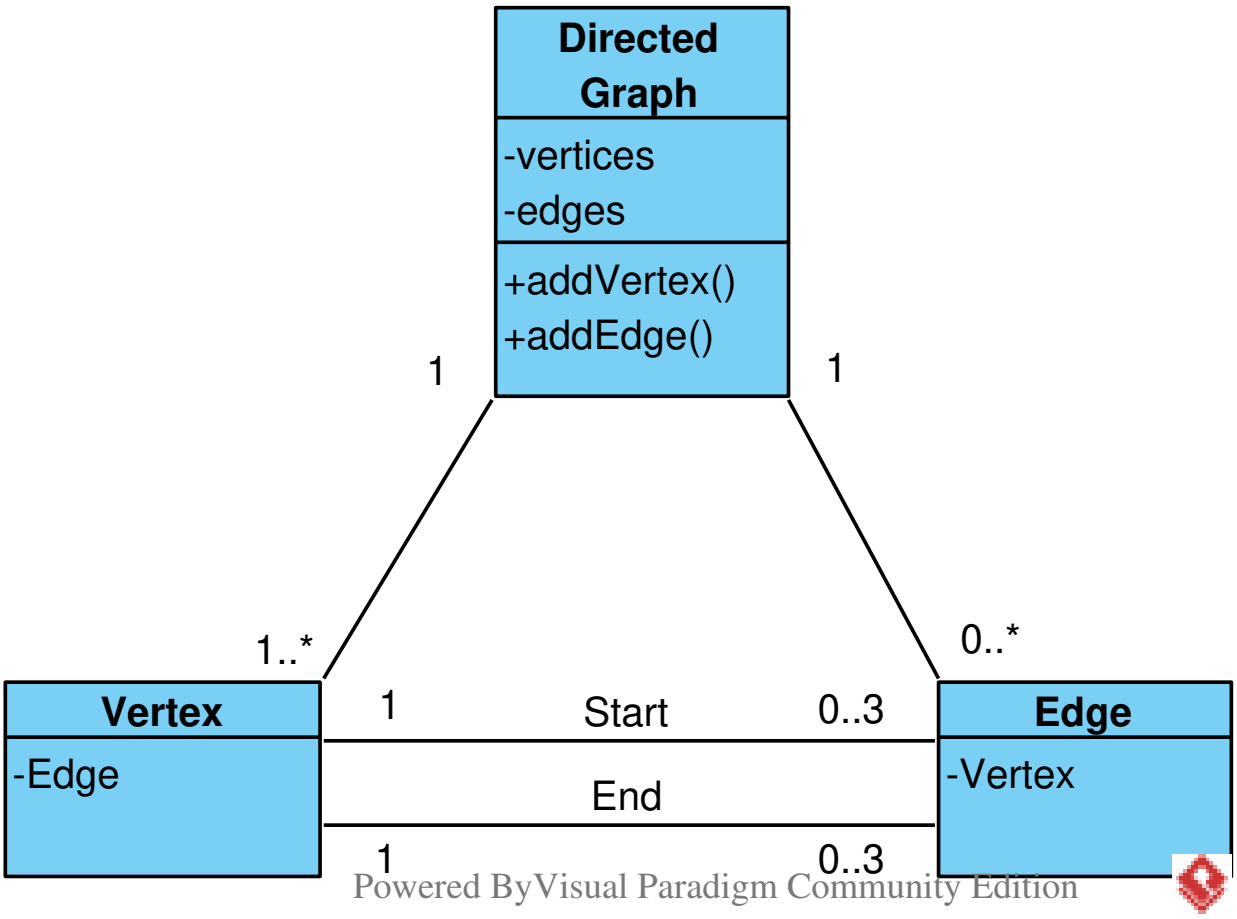

Figure 3.2: Sample Directed Graph

3.3 Exercise 2.3

1. Window Class

   First, the Window class is associated with the ScrollingWindow, Canvas, and Panel class by being their parent. This is an example of composition association, as they can't exist without the Window class. Furthermore, the Window class has attributes x1, y1, x2, y2. Additionally, it has multiple methods, those being display, undisplay, raise, and lower.

2. Canvas Class

   Since the Canvas class is a child of the Window class, it inherits the Window class' attributes and methods. Moreover, the Canvas class is the sole parent to the Shape class, indicating that

it shares a composition association with it. To be more specific in its relationship with the Shape class, one Canvas instance associates with numerous Shape elements. It also shares an aggregation association with the ScrollingCanvas class as it is one of multiple parents. In terms the attributes, the Canvas class has cx1, cy1, cx2, and cy2. It also has two methods, being addElement and deleteElement.

3. ScrollingWindow Class

   The ScrollingWindow class is also the child of the Window class, so it inherits its methods and attributes. Furthermore, the ScrollingWindow class has two attributes, xOffset and yOffset. It also has a scroll method. It shares a composition association with the TextWindow class as it's its only parent. On the other hand, it shares a aggregation association with the ScrollingCanvas class since it has multiple parents.

4. Panel Class

   The Panel class is another child of the Window Class so it inherits its methods and attributes. It is one of the parents of the PanelItem class, making it a aggregation association. Additionally, an instance of the Panel class can be associated with either 0 or 1 instances of PanelItem. Lastly, it has one attribute, being itemName.

5. Event Class

   The Event class is a parent to both the PanelItem and TextItem class. It has an aggregation association with both as they could exist without the Event class. Furthermore, it contains an attribute called action. To be more specific in it's association with the PanelItem class, for every event it can be associated with numerous PanelItem instances. Similarly, for every event it can be associated with numerous TextItem instances. It also has a role, notifyEvent, that connects to the PanelItem class, alongside another behavior, keyboardEvent, that connects to the TextItem class.

6. Shape Class

   The Shape class is a child of the Canvas class, so it inherits all of its attributes and methods. Furthermore, Each shape instance can only associate with one Canvas instance. The Shape class is the sole parent of both the Line and ClosedShape class. This means that it shares a composition association with both of them. Finally, it contains two attributes, color and lineWidth.

7. PanelItem Class

   The PanelItem class is a child of both the Panel and Event class, so it inherits attributes and methods from both of them. Moreover, each PanelItem instance sees one Panel instance and one Event instance. It has three attributes, those being x, y, and label. It is also the only parent to the Button and ChoiceItem class, indicating that it has a composition association with both. Lastly, it is one of the parents of the TextItem class, meaning that it has an aggregation association with it.

8. TextWindow Class

   The TextWindow class is a child of the ScrollingWindow class, so it inherits each of its attributes and methods. It also has one attribute, being string. Lastly, it has insert and delete methods.

9. ScrollingCanvas Class

   The ScrollingCanvas class is a child of both the ScrollingWindow and Canvas class, meaning that it inherits attributes and methods from both of them. Apart from that, it has no unique attributes or methods.

10. Line Class

    The Line Class is a child of the Shape class so it inherits its attributes and methods. It also has four attributes, being x1, y1, x2, and y2. Lastly, it has a draw method.

11. ClosedShape Class

    Similarly, the ClosedShape class is a child of the Shape class so it also inherits its attributes and methods. It is also the sole parent to the Ellipse class, making it a composition association. On the other hand, it is one of the Polygon class' parents, meaning that they share an aggregation association. Finally, the ClosedShape class has two attributes, being fillColor and fillPattern.

12. Button Class

    The Button class is a child of the PanelItem class, indicating that it inherits all of its methods and attributes. Furthermore, it has two attributes, string and depressed.

13. ChoiceItem Class

    Next is the ChoiceItem class, which is also a child of the PanelItem class, meaning that it inherits its attributes and methods. It doens't have any unique attributes or methods itself, but has a child ChoiceEntry. It shares a composition association with the ChoiceEntry class as the child can't exist without the parent. There are two subsets of the class, as they are separated by roles/behaviors. One role is the currentChoice, which always sees one instance of the ChoiceEntry class. On the other hand, the choices role sees numerous instances of the ChoiceEntry class.

14. TextItem Class

    The TextItem class is a child of both the PanelItem and Event class, so it inherits attributes and methods from both of them. It also has two attributes, being maxLength and currentString. To be more specific in it's association with the Event class, one instance of TextItem always sees one instance of the Event class.

15. Point Class

The Point class shares an association with the Polygon class, as any instance of Point always sees one instance of the Polygon class. Furthermore, the Point class has two attributes, being x and y.

16. Polygon Class

    The Polygon class is a child of the ClosedShape class so it inherits all of its attributes and methods. As mentioned before, it shares an association with the Point class. Furthermore, any instance of the Polygon class sees numerous instances of the Point class. Lastly, this class has a draw attribute.

17. Ellipse Class

    The Ellipse class is also a child of the ClosedShape class, meaning that it inherits its attributes and methods. It has four attributes, being x, y, a, and b. It also has a draw method.

18. ChoiceEntry Class

    The ChoiceEntry class is a child of the ChoiceItem class. As mentioned earlier, there are two subsets for potential roles, being currentChoice and choices. The currentChoice role can see either zero or one ChoiceItem instances, and the choices role can always see one instance of the ChoiceItem class.

3.4 Exercise 2.4

1. MailingAddress Class

   The MailingAddress class is the sole parent of the Customer class, meaning that they share a composition association. In terms of multiplicity, an instance of the MailingAddress class could be connected to numerous instances of the Customer class. Furthmore, the MailingAddress class has an association with the CreditCardAccount Class. Additionally, one instance of the MailingAddress class can see numerous instances of the CreditCardAccount class. Lastly, this class has two attributes, being address and phoneNumber.

2. CreditCardAccount Class

   The CreditCardAccount class is the only parent of the Statement class, indicating that there is a composition association between the two. When it comes to multiplicity, an instance of this class can see either zero or one instance of the Statement class. Moreover, this class shares an association with the Institution class. Furthermore, an instance of this class will always be correlated to one instance of the Institution class. Finally, this class has maximumCredit and currentBalance as attributes.

3. Institution Class

   The Institution Class is only associated with the CreditCardAccount class, and each instance of this class can see either zero or one instances of the CreditCardAccount Class. It has also three attributes, those being name, address, and phoneNumber.

4. Customer Class

   The Customer class is the child of the MailingAddress class, indicating that it inherits all of its attributes. For every instance of the Customer class, it can see numerous instances of the MailingAddres class. Lastly, it has an attribute called name.

5. Statement Class

   The Statement Class is the child of the CreditCardAccount class, so it inherits all of its attributes. Each instance of this class correlates to one instance of the CreditCardAccount class. Furthermore, this class is associated with the Transaction class. For every instance of the Statement class, it either sees zero or one instances of the Transaction class. This class has multiple attributes, those being paymentDueDate, financeCharge, and minimumPayment.

6. Transaction Class

   The Transaction class is associated with the Statement class. Each instance of this class always connects to one instance of the Statement class. This class is also the parent to the CashAdvance, Interest, Purchase, Fee, and Adjustment class. It shares a composition association with all of these classes as well. Finally, it has three attributes, being transactionDate, explanation, and amount.

7. CashAdvance Class

   The CashAdvance class is the child of the Transaction class, meaning that it inherits its attributes.

8. Interest Class

   Similarly, the Interest class is another child of the Transaction class, resulting in it inheriting its attributes.

9. Purchase Class

   The Purchase Class is also a child of the Transaction class and inherits all of its attributes. It also has a child, the Merchant class, and shares a composition association with it. Furthermore, every instance of the Purchase class correlates to one instance of the Merchant class.

10. Fee Class

    The Fee Class is another child of the Transaction class, meaning that it inherits its attributes. It also has a unique attribute, being feeType.

11. Adjustment Class

    The Adjustment Class a child of the Transaction class, indicating that it inherits all of its attributes.

12. Merchant Class

    The Merchant Class is the child of the Purchase class, resulting in it inheriting whatever attributes and methods the Purchase class might have. Lastly, it has one unique attribute, being name.

3.5 Class - to - Code Modeling

```
# 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, inventory: int = 0):
        self.__inventory = inventory

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

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

    def addInventory(self, amount: int):
        self.__inventory += amount

    def removeInventory(self, amount: int):
        if self.__inventory >= amount:
            self.__inventory -= amount
        else:
            raise ValueError('There is not enough inventory to remove this amount.')

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


# no forward reference needed since Product is defined
class Sale:
    __saleTimes = 0
    __saleNumber: int = 0

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

```
43          self.__saleNumber = Sale.__saleTimes
44          for product in self.__product:
45              product.removeInventory(1)
46              product.setLastSale(self)
47
48      @property
49      def getSaleNumber(self) -> int:
50          return self.__saleNumber
51
52
53  productOne = Product(inventory=15)
54  productTwo = Product(inventory=10)
55
56  saleOne = Sale([productOne, productTwo])
57  saleTwo = Sale([productOne])
58  saleThree = Sale([productTwo])
59
60
61  print(f"Product One inventory: {productOne.getInventory}, last sale number: {
        productOne.getLastSale.getSaleNumber}")
62  print(f"Product Two inventory: {productTwo.getInventory}, last sale number: {
        productTwo.getLastSale.getSaleNumber}")
```

3.6 Code Results

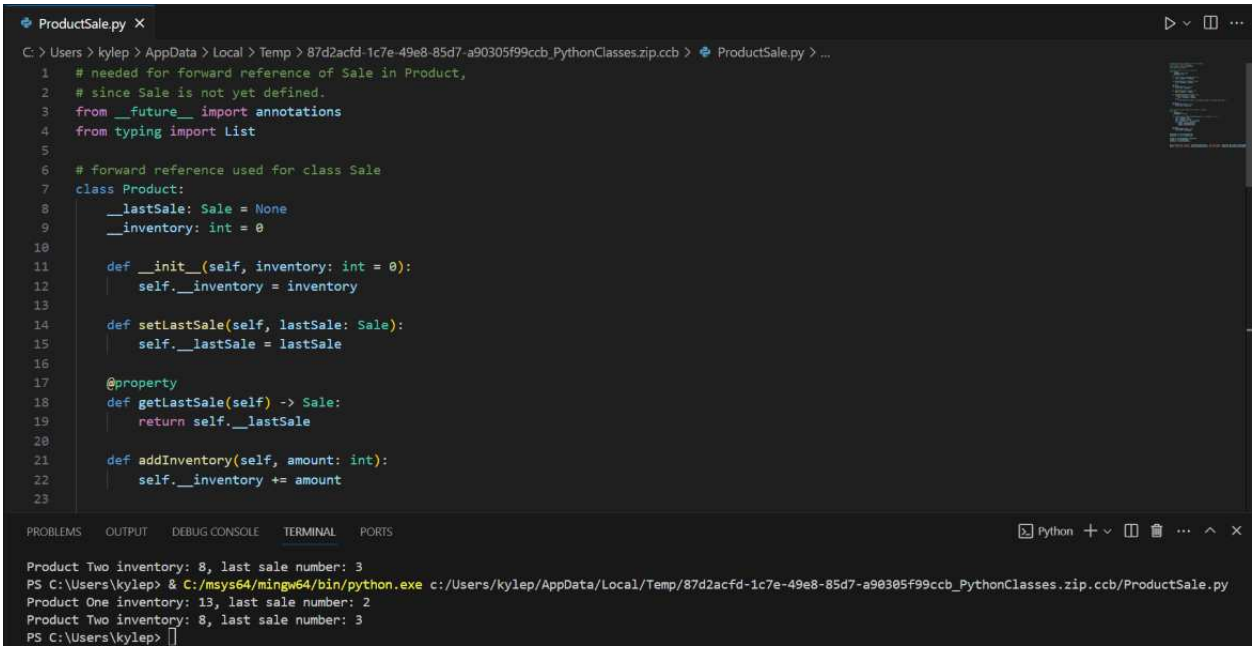

Figure 3.3: Code Results

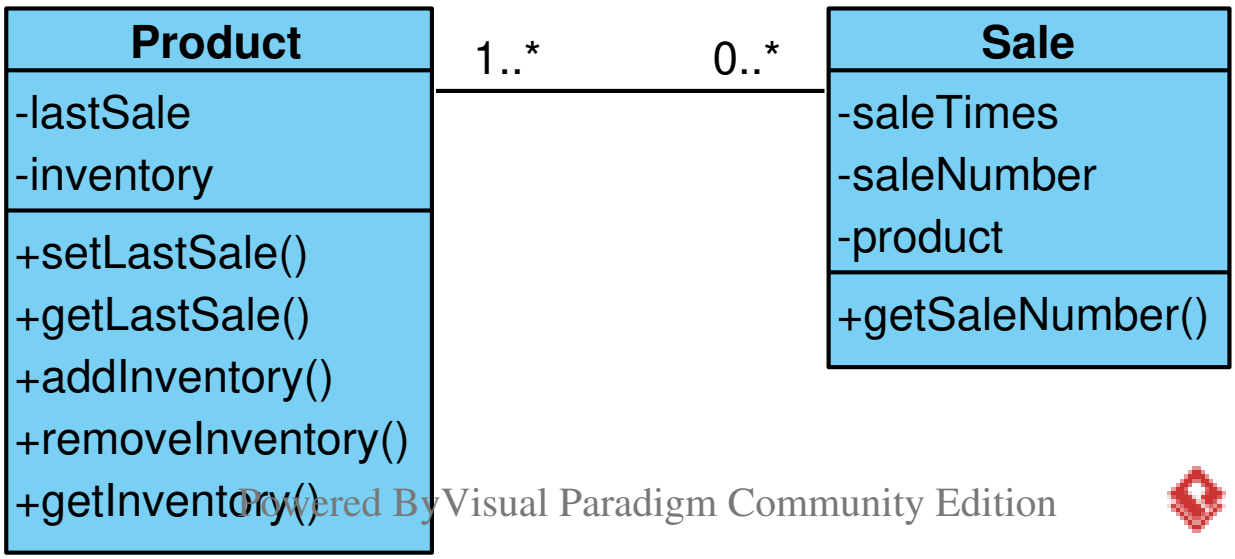

Figure 3.4: UML Diagram for Product and Sale

Chapter 4

Software Execution Model

– Kyle Ponte

4.1 Exercise 5.1

I copied all of the information from the tables onto an Excel and utilized formulas to determine the CPU time, Disk time, and Network time. After completing some calculations, I found the CPU, Disk, and Network times to be 0.0775s, 0.28s, and 2.44s. The total time of P0 is the sum of these values, which comes to 2.7975s. I also needed to get the demand times per unit for Work, DB, and Msg. I completed some more calculations in the excel and determined them to be 0.12s/unit, 6.47s/unit, and 0.013s/unit.

Now calculating the times for all processes (Work is in excel):
P0 = 2.7975s, P1 = 0.493s, P3 = 0.013s, P5 = 13.099s, P6 = 0.013s, P7 = 6.47s, P10 = 7.683s, P11 = 0.013s, P12 = 13.352s, P13 = 20.383s

Prior to any path divergences, P0, P1, and P0 are all run. Therefore, we can assume that every path will start with this time: P0 + P1 + P0 = 2.7975 + 0.493 + 2.7975 = 6.088s.

Demand time calculations for each path:
Path 1 = P5 + P7 = 19.569s (Take P7 because it is the longest process in a parallel operation), Path 2 = P0 = 2.7975s, Path 3 = P1 + P11 = 0.506s (Take P11 because it is the shortest process in a series operation), Path 4 = P13 = 20.383s

The shortest demand path is Path 3 and the longest demand path is Path 4. Using these two paths we can find solve for the best and worst demand times.

Since the loop runs 4 times and experiences P3 and P0 every time, we can use this formula: 6.088 + 4(P3 + Path + P0)

Overall Best Demand Time = 6.088 + 4(P3 + Path 3 + P0) = 19.354s

Overall Worst Demand Time = 6.088 + 4(P3 + Path 4 + P0) = 98.862s

Next, we need to consider the probabilities to determine the average time it would take to go through the paths.

Average time for taking the paths = (Path 1 * 0.1 + Path 2 * 0.2 + Path 3 * 0.3 + Path 4 * 0.4) = 10.8214s

Overall Average Demand Time = 6.088 + 4(P3 + 10.8214 + P0) = 60.6156s

I then went ahead and calculated the total CPU, Disk, and Network demand times by simply adding P0's values to the sum of their Software values multiplied by their Hardware and Service

Time values.

Total CPU Demand Time = 0.0775 + 3 * (400 + 15000 + 10) * 0.0001 = 0.6505s, Total Disk Demand Time = 0.28 + 2 * (0 + 150 + 0) * 0.02 = 6.28s, Total Network Time = 2.44 + 1 * (0 + 2 + 1) * 0.01 = 2.47s

4.2 Exercise 5.2

I first calculated the demand time per unit for Work, DB, and msg in Excel. I solved these to be 0.0075s, 0.58s, and 0.1575s. I then went ahead and calculated the demand times for each process. This was done by multiplying their software resources values by the demand time per unit for each software resource (Work found in excel):
Initialize = 0.7525s, Switch Operation = 0.0075s, updateMirrorNY = 1.5125s, updateMirrorLA = 1.505s, queryMirrorNY = 0.9175s, queryMirrorLA = 0.925s, writeMirrorNY = 0.9175s, writeMirrorLA = 0.925s

Using this, we can determine the demand time for query and write:
Query = min(queryMirrorNY, queryMirrorLA) = 0.9175s (Take the shortest time because of series operation)
Write = max(writeMirrorNY, writeMirrorLA) = 0.925s (Take the longest time because of parallel operation)

Additionally, initialize and switch are executed in every path, so we can assume every path will start with this time: 0.7525 + 0.0075 = 0.76s.

After calculations in excel, I determined the Path 3 (through query) to be the fastest while Path 1 (through updateMirrorNY) to be the slowest.

Overall Best Demand Time = 2 * (0.76 + Path 3) = 3.355s

Overall Worst Demand Time = 2 * (0.76 + Switch operation + Path 1) = 4.56s

Before calculating the average demand time for the entire process, I need to calculate the average time for both update paths.

Update Average Path Time = 0.7 * updateMirrorNY + 0.3 * updateMirrorLA + Switch Operation = 1.51775s

Using this, we can then calculate the average demand time.

Overall Average Demand Time = 2 * (0.76 + 0.5 * Update Average Path Time + 0.4 * Path 3 + 0.1 * Path 4) = 3.95675s

4.3 Exercise 5.3

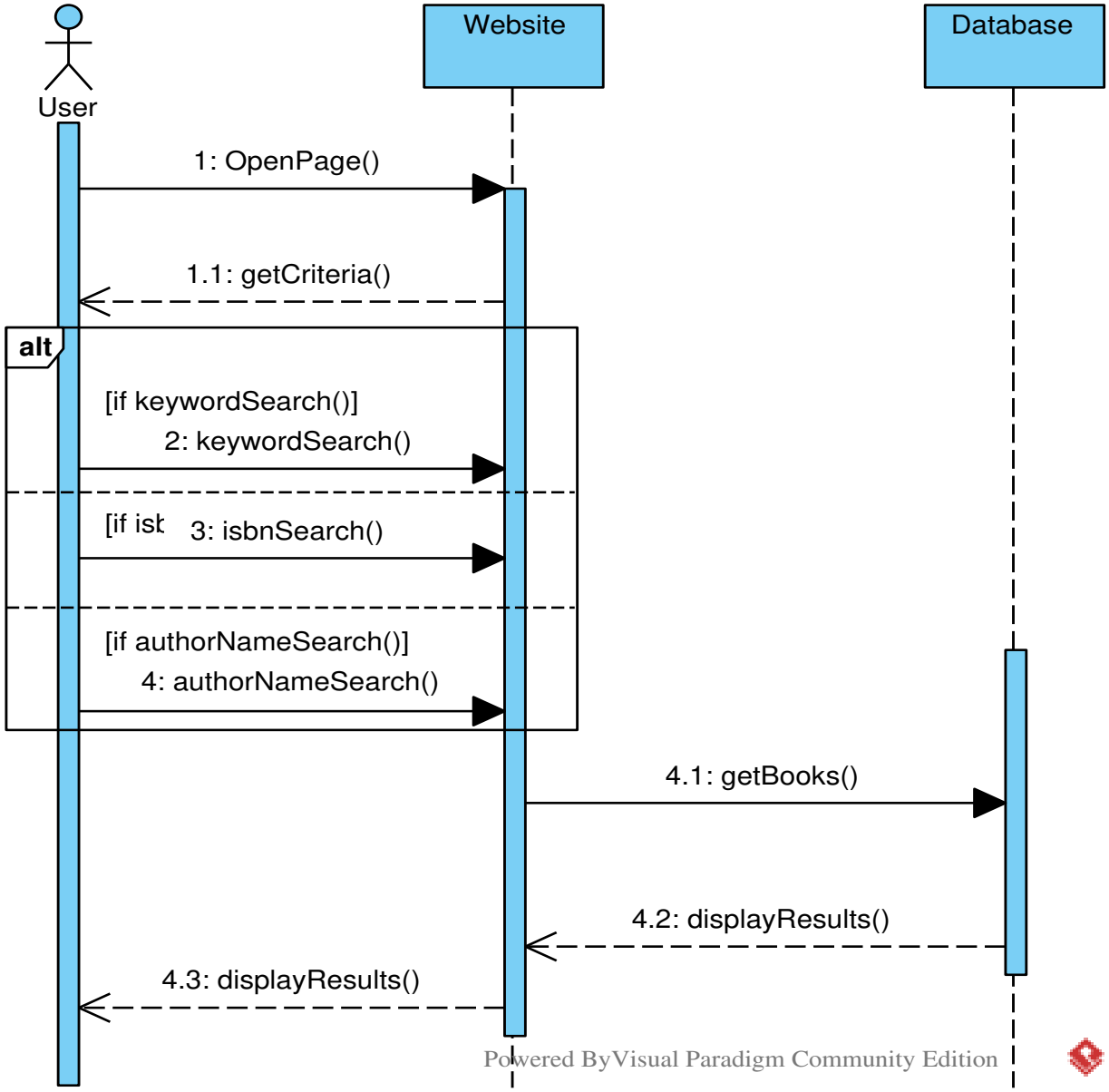

Figure 4.1: eStore Sequence Diagram

In the above figure I utilized an alternative combined fragment to demonstrates the user's ability to instantiate three types of searches. Furthermore, I selected 0.6, 0.1, and 0.3 as my probabilities for the keywordSearch, isbnSearch, and authorNameSearch. The constraint is clearly that the probabilities must add up to 1.

Using the given information, I went ahead and calculated the demand times for each process. (Work shown in excel):

OpenPage() = 0.2s, getCriteria() = 2.22s, keywordSearch() = 40s, isbnSearch() = 0s, authorNameSearch() = 0s, getBooks() = 0s, displayResults() = 0.2s

Overall Best Demand Time = OpenPage + getCriteria + min(keywordSearch, isbnSearch, authorNameSearch) + getBooks + 0.2 * displayResults = 2.82s

Overall Worst Demand Time = OpenPage + getCriteria + max(keywordSearch, isbnSearch, authorNameSearch) + getBooks + 0.2 * displayResults = 42.82s

Prior to calculating the overall average, I solved for the average path time at the switch operator using my probabilities.

Average Path Time = 0.6 * keywordSearch + 0.1 * isbnSearch + 0.3 * authorNameSearch = 24s

Now we can find the average demand time:

Overall Average Demand Time = OpenPage + getCriteria + 24 + getBooks + 2 * displayResults = 26.82s

4.4 Link to the Updated Github Repository

Github Repository

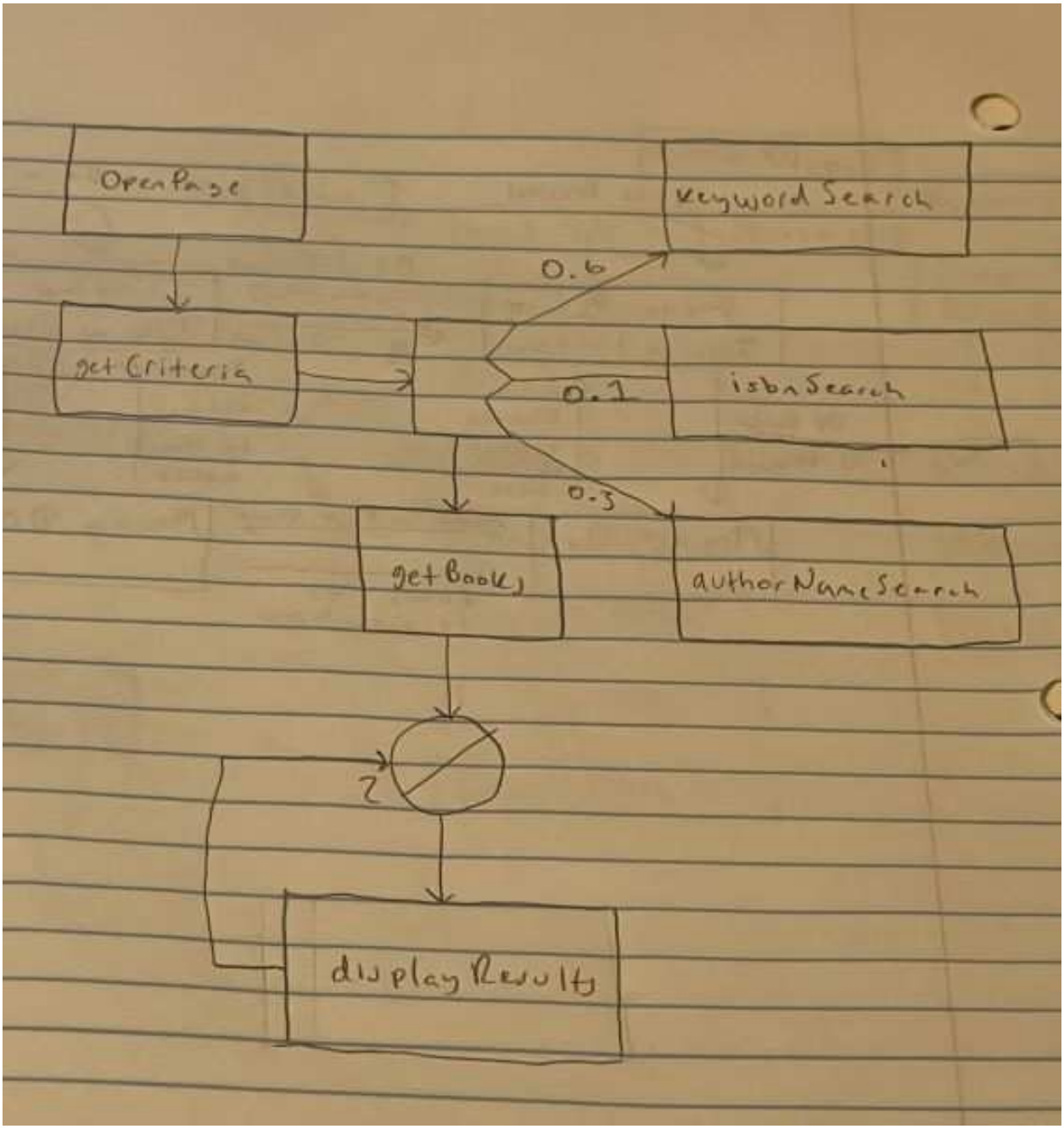

Figure 4.2: eStore Software Execution Diagram

Chapter 5

Exam One Extra Work

– Kyle Ponte

5.1 Problem 5.6

5.1.1 Human.py Code

```
from abc import ABC, abstractmethod

class Human(ABC):
    def __init__(self, profession):
        self._profession = profession

    @abstractmethod
    def _SetProfession(self, profession):
        pass

    def GetProfession(self):
        return self._profession

class Student(Human):
    def __init__(self):
        super().__init__("Student")

    def _SetProfession(self, profession):
        self._profession = profession

class Teacher(Human):
    def __init__(self):
        super().__init__("Teacher")

    def _SetProfession(self, profession):
        self._profession = profession
```

5.1.2 PairGame.py code

```python
from abc import ABC, abstractmethod

class PairGame(ABC):
    def __init__(self):
        self.player_one = None
        self.player_two = None

    def SetPlayerOne(self, player):
        self.player_one = player

    def SetPlayerTwo(self, player):
        self.player_two = player

    @abstractmethod
    def Play(self):
        pass

class Chess(PairGame):
    def Play(self):
        print(f"{self.player_one.GetProfession()} and {self.player_two.GetProfession()} are playing chess.")

class Checkers(PairGame):
    def Play(self):
        print(f"{self.player_one.GetProfession()} and {self.player_two.GetProfession()} are playing checkers.")
```

5.1.3 TeacherStudentGame.py (code driver) code

```python
from Human import Student, Teacher
from PairGame import Chess, Checkers

if __name__ == "__main__":
    # Instantiate Student and Teacher
    student = Student()
    teacher = Teacher()

    # Instantiate games
    chess_game = Chess()
    checkers_game = Checkers()

    # Set players for both games
    chess_game.SetPlayerOne(teacher)
    chess_game.SetPlayerTwo(student)

    checkers_game.SetPlayerOne(teacher)
    checkers_game.SetPlayerTwo(student)

    # Execution of games being played
    chess_game.Play()
    checkers_game.Play()
```

Chapter 6

Decorator Pattern

– Kyle Ponte

6.1 Exercise 14.1

6.1.1 MargheritaPizza file code

```
public class MargheritaPizza implements Pizza {
    @Override
    public String getDescription() {
        return "Margherita Pizza";
    }

    @Override
    public double getCost() {
        return 6.99;
    }
}
```

6.1.2 MushroomDecorator file code

```
public class MushroomDecorator extends ToppingDecorator {
    public MushroomDecorator(Pizza pizza) {
        super(pizza);
    }

    @Override
    public String getDescription() {
        return pizza.getDescription() + ", mushrooms";
    }

    @Override
    public double getCost() {
        return pizza.getCost() + 1.25;
    }
}
```

6.1.3 PepperoniDecorator file code

```
1  public class PepperoniDecorator extends ToppingDecorator {
2      public PepperoniDecorator(Pizza pizza) {
3          super(pizza);
4      }
5
6      @Override
7      public String getDescription() {
8          return pizza.getDescription() + ", pepperoni";
9      }
10
11     @Override
12     public double getCost() {
13         return pizza.getCost() + 1.50;
14     }
15 }
```

6.1.4 Output Results

Figure 6.1: Output Results

6.1.5 UML Diagram

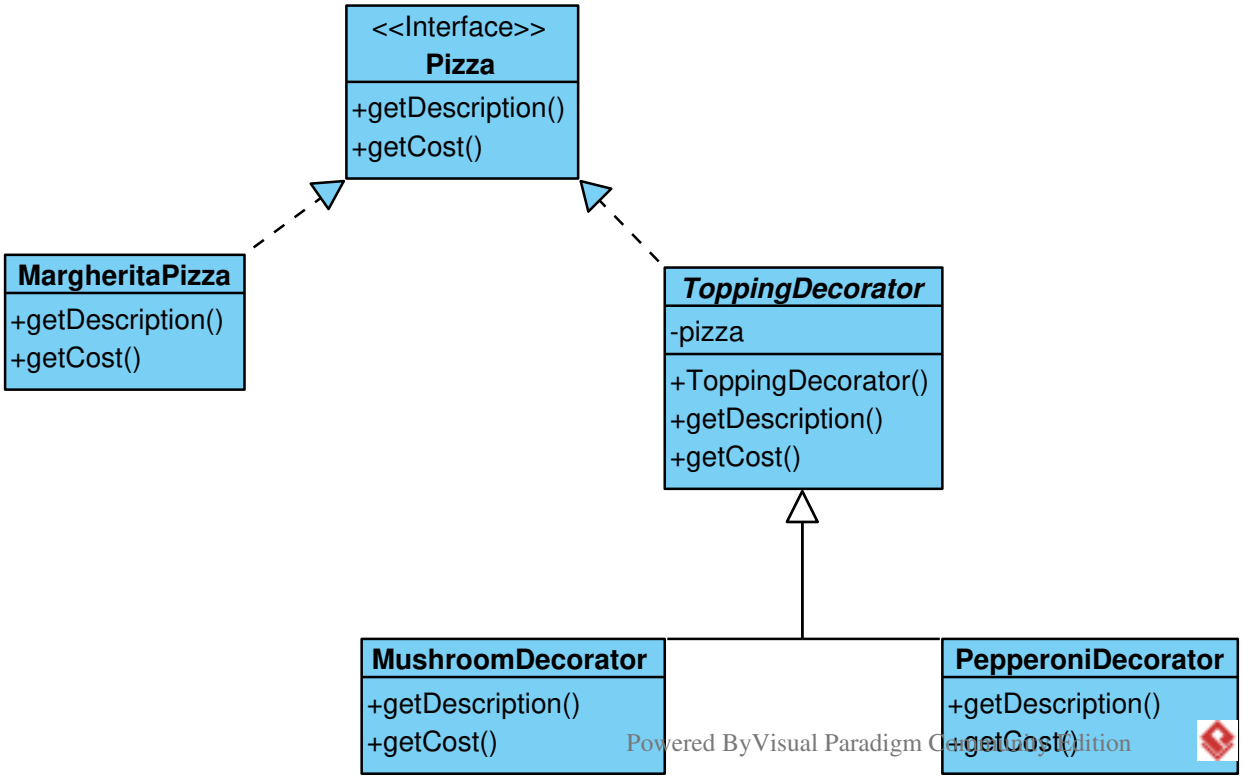

Figure 6.2: UML Diagram

Bibliography

Index

Chapter
    Decorator Pattern, 20
    Exam One Extra Work, 18
    Git Homework, 2
    SoftwareExecutionModel, 13
    Team, 1
    UML Class Modeling, 4

Decorator Pattern, 20

Exam One Extra Work, 18

introduction, 1, 2, 4

Software Execution Model, 13