

Homework

by

Jason McCauley

Stevens.edu

December 7, 2024

© Jason McCauley
Stevens.edu
ALL RIGHTS RESERVED

Homework

Jason McCauley
Stevens.edu

Table 1: Document Update History

Date	Updates
09/11/2024	JM: <ul style="list-style-type: none">• Created dsnTeam.tex, which introduces the "Team" chapter of the book. Includes two paragraphs about the author, as well as two sample EPS images.• Furthermore, updated dsnManual.tex to include this file.
09/12/2024	JM: <ul style="list-style-type: none">• Created dsnGitHomework.tex, which introduces the "Git Homework" chapter of the book. Includes a screenshot showing the completion of LearningGit-Branching. Updated dsnManual.tex to include this file.
09/17/2024	JM: <ul style="list-style-type: none">• Cleaned up the book by removing any unnecessary chapters or references from the original provided template.• Created a new chapter, dsnHomeworkTwo.tex, which will include all of the needed diagrams, responses, and code for the "UML Class Modeling" assignment.
09/18/2024	JM: <ul style="list-style-type: none">• Created class diagrams to describe the undirected graph and directed graph provided in the Class Modeling pdf.• Added their EPS figures to "dsnHomeworkTwo.tex" under the Class Diagrams section.

Table 1: Document Update History

Date	Updates
09/19/2024	<p>JM:</p> <ul style="list-style-type: none"> • Under the Description of Associations section of "dsnHomeworkTwo.tex", added a paragraph to explain the Class Diagrams for the windowing system and the credit card system, describing all of the associations for each. • Under the Class-to-Code Modeling section of "dsnHomeworkTwo.tex", embedded the code for ProductSale_modified.py, which now keeps track of the inventory for each product instance as it is sold. Added a screenshot to show the output of this code working as expected. • Created a UML Class Diagram for the updated ProductSale code, detailing the member variables and functions for the Product and Sale classes, along with their association multiplicities.
10/10/2024	<p>JM:</p> <ul style="list-style-type: none"> • From Figure 5.6 of the spec sheet, calculated the best, worst, and average demand times of the execution path. Provided the table of demand time for each software component and for each node that I put together to help with these calculations. Explained my thought process as I performed my calculations at different nodes. Also provided a table of total demand times for each of the three hardware components, explaining my process. • From Figure 5.7 of the spec sheet, calculated the best, worst, and average demand times of the execution model. Provided a table of demand time for each of three software components and each node, which I put together to help in my calculations. Also explained my thought process when for stepping through the different nodes of the execution model, and how I adjusted my calculations accordingly. • Created a simplified sequence diagram based on the three given objects. From the sequence diagram, put together a software execution model, showing the corresponding path of nodes along with the demand weights assigned to each node. Provided a chart of demand time for each node, which was used to assist in calculating the worst, best, and average execution time of the diagram. Explained my thought process in moving from node to node, and how I handled each calculation time accordingly.
11/01/2024	<p>JM:</p> <ul style="list-style-type: none"> • Created chapter for Exam One Extra Work, which includes an EPS file of the UML diagram to be implemented and my C++ code for TeacherStudentGame.

Table 1: Document Update History

Date	Updates
11/19/2024	<p>JM:</p> <ul style="list-style-type: none"> • Created chapter for Decorator Pattern, which includes Java code for the concrete classes MargheritaPizza, MushroomDecorator, and PepperoniDecorator. • Added EPS files for the screen-shot of the output results and the UML class diagram for the complete code.
12/06/2024	<p>JM:</p> <ul style="list-style-type: none"> • Added Java code for singleton class SingleObject and SingletonPatternDemo, and EPS screenshot of output when running SingletonPatternDemo in terminal • Added Java code for interface ComputerPart, concrete classes Keyboard, Monitor, Mouse, and Computer, interface ComputerPartVisitor, concrete visitor ComputerPartDisplayVisitor, and VisitorPatternDemo, along with EPS screenshot of output when running VisitorPatternDemo in terminal • Added JavaScript code for visitorDemo along with EPS screenshot of output when running visitorDemo in terminal

Table of Contents

1	Team	
	– <i>Jason McCauley</i>	1
2	Git Homework	
	– <i>Jason McCauley</i>	3
3	UML Class Modeling	
	– <i>Jason McCauley</i>	4
3.1	Class Diagrams	4
3.2	Description of Associations	5
3.3	Class-to-Code Modeling	5
4	Software Execution Models	
	– <i>Jason McCauley</i>	9
4.1	Exercise 5.1	9
4.2	Exercise 5.2	11
4.3	Exercise 5.3	13
5	Exam One Extra Work	
	– <i>Jason McCauley</i>	16
5.1	Exercise 5.6	16
6	Decorator Pattern	
	– <i>Jason McCauley</i>	19
6.1	Exercise 14.1	19
7	Visitor and Singleton Patterns	
	– <i>Jason McCauley</i>	22
7.1	Write a Program to Implement Singleton Pattern	22
7.2	Write a Program to Implement Visitor Pattern	23
7.3	Setting up the Visitor Pattern in Javascript	25

Chapter 1

Team

– Jason McCauley

Hey! I can't believe I actually got this to work... anyways, my name is Jason McCauley. I am a 4/4 Software Engineering student here at Stevens Institute of Technology. While I do not have any industry experience yet, I did take part in research this past summer. Working under Dr. Jacob Gissinger in the Chemical Engineer Department, I preprocessed a large dataset which encompassed the interaction between two polystyrene molecules. Utilizing relevant features, such as the position and velocity vectors of the molecules, I developed a robust machine learning model to predict whether or not a reaction would occur between them, achieving 94% accuracy.

While I fully recognize that this semester will be one of the most rigorous academically, I am eager to explore the valuable knowledge that each course has to offer. I am certain that those tools will be beneficial when I hopefully work in the industry. However, until then, it is time to buckle up, lock in, and prepare for the journey that lies ahead these next few months! Outside of the classroom, I am most passionate about playing guitar, basketball, and going to the gym.

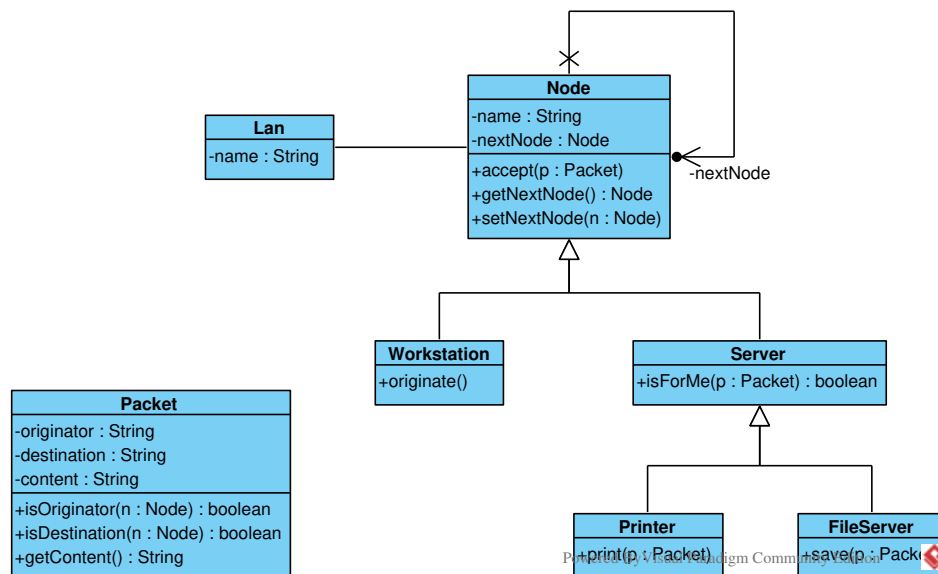


Figure 1.1: First EPS figure, sample Class Diagram generated through Visual Paradigm.

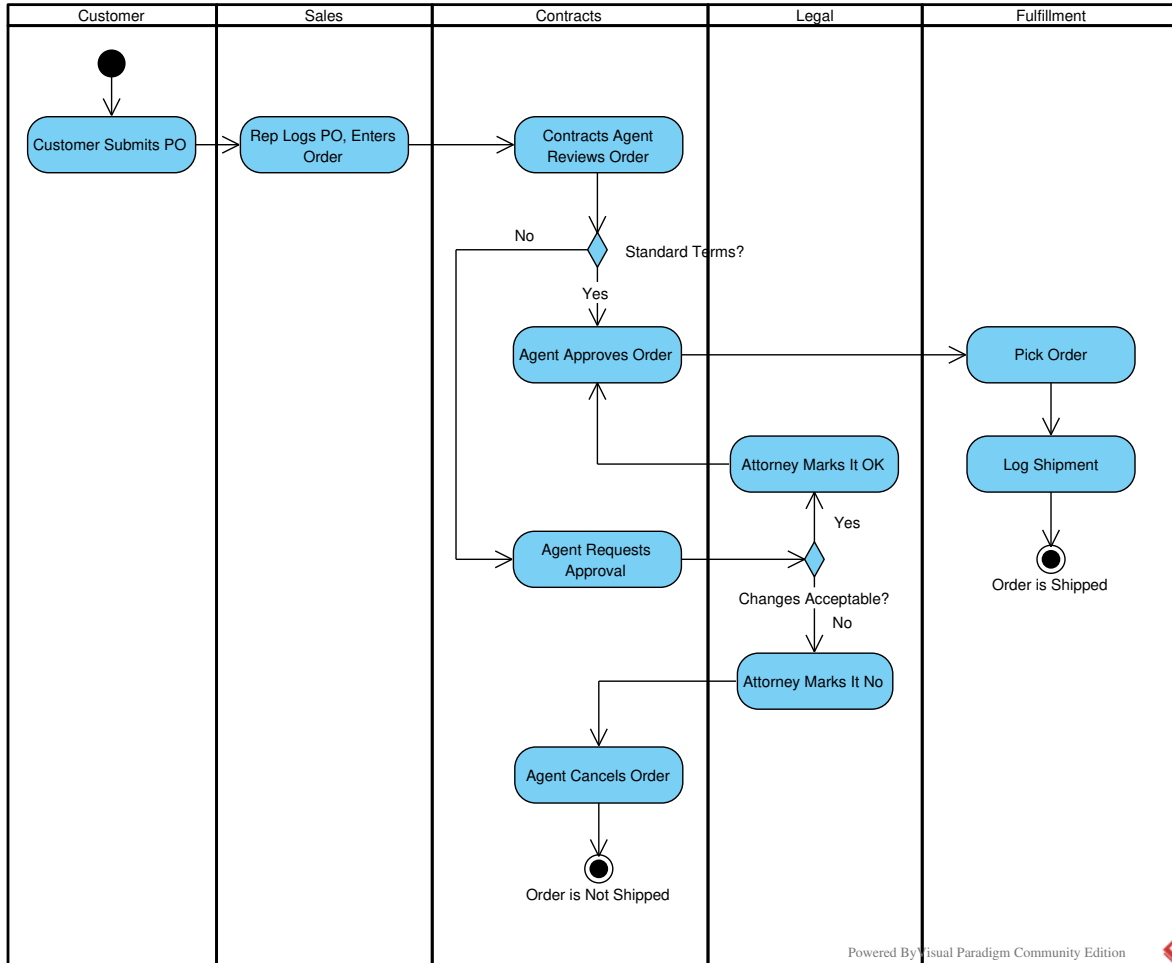


Figure 1.2: Second EPS figure, sample Activity Diagram generated through Visual Paradigm.

Chapter 2

Git Homework

– Jason McCauley

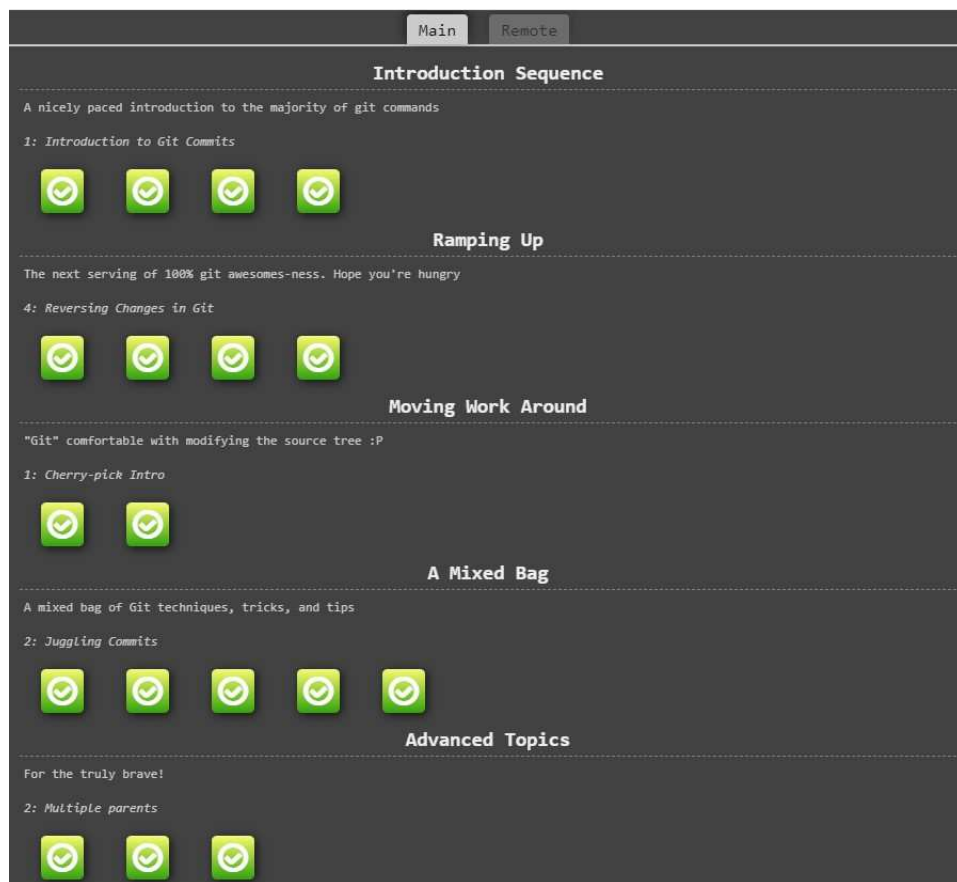


Figure 2.1: Screenshot showing that all five levels of LearningGitBranching have been solved.

Chapter 3

UML Class Modeling

– Jason McCauley

3.1 Class Diagrams

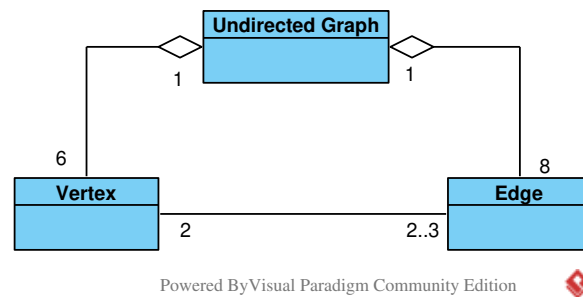


Figure 3.1: Class model describing the connectivity of the provided undirected graph, consisting of edges and vertices.

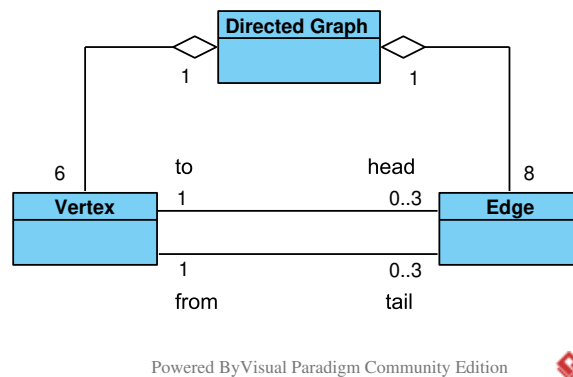


Figure 3.2: Class model describing the connectivity of the provided directed graph, consisting of oriented edges and vertices.

3.2 Description of Associations

For the windowing system, the Window class serves as a superclass, with the classes ScrollingWindow, Canvas, and Panel each inheriting its functions. Furthermore, the ScrollingWindow class acts as a superclass, with classes TextWindow and ScrollingCanvas each inheriting its functions. ScrollingCanvas actually exhibits multiple inheritance from disjoint classes, as it inherits the functions from the Canvas class as well. To elaborate on the Canvas class, it sees multiple instances of the Shape class, each called elements. The Shape class sees exactly one instance of the Canvas class called window. Furthermore, this Shape class acts as a superclass, with classes Line and ClosedShape each inheriting its functions as subclasses. Additionally, ClosedShape serves as a superclass, with the Polygon and Ellipse classes each inheriting its functions. Moreover, the Polygon class sees multiple instances of the Point class called vertices, where these instances are ordered. On the other hand, the Point class sees exactly one instance of the Polygon class. Moving back to the top of the diagram, the Panel class, via the member variable itemName, sees either zero or one instance of the PanelItem class. The PanelItem class sees exactly one instance of the Panel class via itemName. Furthermore, the PanelItem class sees exactly one instance of the Event class, called notifyEvent. The Event class sees many instances of the PanelItem class. Additionally, the Event class sees many instances of the TextItem class, while the TextItem class only sees one instance of the Event class called keyboardEvent. This TextItem class, along with Button and ChoiceItem, each serve as subclasses to the aforementioned PanelItem class, inheriting its functions. The ChoiceItem class is unique, in that it will see either many instances of the ChoiceEntry class called choices, or as a subset, see one instance of the ChoiceEntry class called currentChoice. From the ChoiceEntry class, through the one currentChoice instance, it will see either zero or once instance of the ChoiceItem class. Otherwise, through the many instances of ChoiceEntry, called choices, it will see exactly one instance of ChoiceItem.

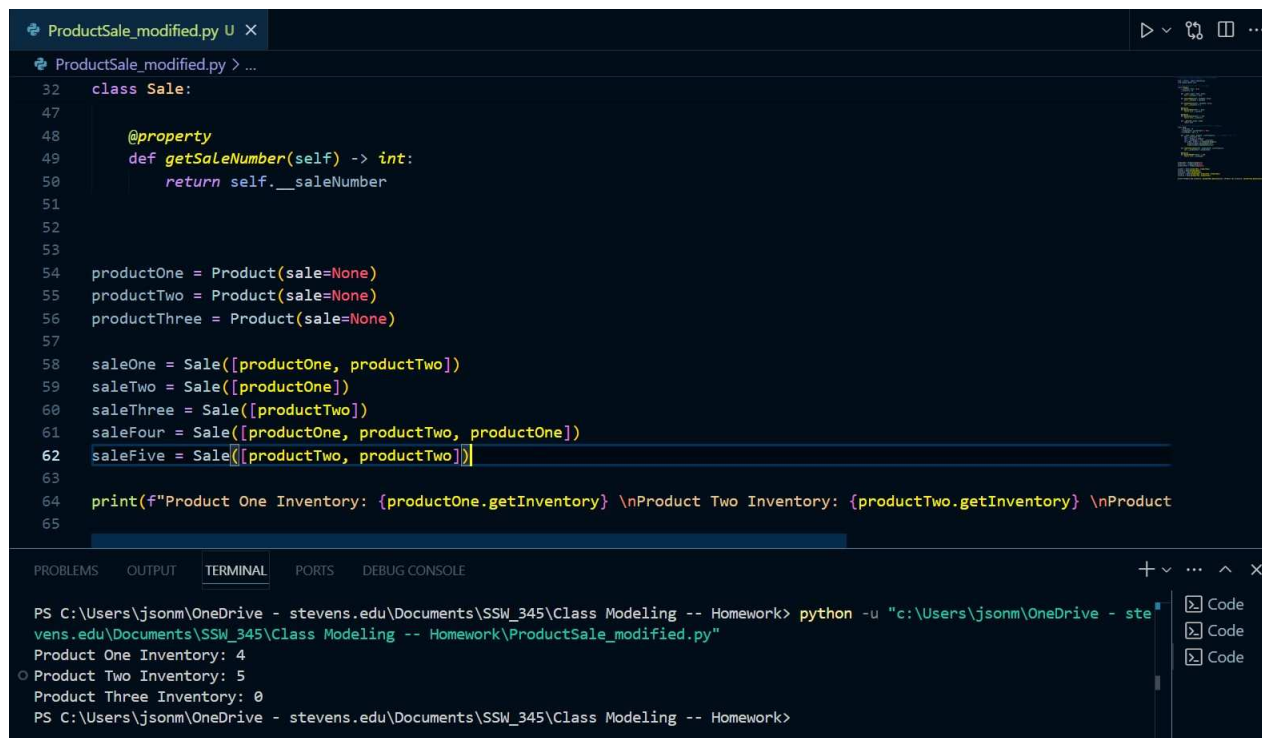
For the credit card system, the MailingAddress class sees many instances of the Customer class, called accountHolder. Likewise, the Customer class sees many instances of MailingAddress. Continuing with the MailingAddress class, it also sees many instances of CreditCardAccount, while the CreditCardAccount class only sees one instance of MailingAddress. Furthermore, the CreditCardAccount class only sees one instance of the Institution class, via the member variable accountNumber. Conversely, the Institution class, via accountNumber, sees either zero or one instance of the CreditCardAccountClass. Also, the CreditCardAccount class, via the member variable statementDate, sees zero or one instance of the Statement class. On the other hand, the Statement class sees one instance of CreditCardAccount via statementDate. Additionally, the Statement class, via member variable transactionNumber, sees either zero or one instance of the Transaction class. The Transaction class, on the other hand, sees exactly one instance of Statement via its member variable transactionNumber. The Transaction class also serves as a superclass, with classes CashAdvance, Interest, Purchase, Fee, and Adjustment each inheriting its methods. One of the subclasses, particularly the Purchase class, sees exactly one instance of the Merchant class, while the Merchant class sees many instances of the Purchase class.

3.3 Class-to-Code Modeling

```
1 # needed for forward reference of Sale in Product,
2 # since Sale is not yet defined.
3 from __future__ import annotations
4 from typing import List
5
6 # forward reference used for class Sale
7 class Product:
8     __lastSale: Sale = None
9     __inventory = 0
10
11     def __init__(self, sale: Sale):
12         self.__lastSale = sale
13
14     def setLastSale(self, lastSale: Sale):
15         self.__lastSale = lastSale
16
17     def setInventory(self, lastSale: Sale):
18         self.__inventory += 1
19
20     @property
21     def getLastSale(self) -> Sale:
22         return self.__lastSale
23
24     @property
25     def getInventory(self) -> int:
26         return self.__inventory
27
28     def __getitem__(self, item):
29         return self
30
31 # no forward reference needed since Product is defined
32 class Sale:
33     __saleTimes = 0
34     __productSold: List[Product] = None
35     __saleNumber: int = 0
36
37     def __init__(self, product: List[Product]): #, saleNumber: int = 1):
38         Sale.__saleTimes += 1
39         self.__product = product
40         self.__saleNumber = Sale.__saleTimes
41         for index, product in enumerate(product):
42             product[index].setLastSale(self)
43             product[index].setInventory(self)
44
45     def setProductsSold(self, productSold: List[Product]):
46         self.__productSold = productSold
47
48     @property
49     def getSaleNumber(self) -> int:
50         return self.__saleNumber
51
52
```

```
53 productOne = Product(sale=None)
54 productTwo = Product(sale=None)
55 productThree = Product(sale=None)
56
57 saleOne = Sale([productOne, productTwo])
58 saleTwo = Sale([productOne])
59 saleThree = Sale([productTwo])
60 saleFour = Sale([productOne, productTwo, productOne])
61 saleFive = Sale([productTwo, productTwo])
62
63
64 print(f"Product One Inventory: {productOne.getInventory} \nProduct Two
    Inventory: {productTwo.getInventory} \nProduct Three Inventory: {
    productThree.getInventory}")
```

Listing 3.1: ProductSale_modified.py Code



The screenshot shows a code editor with a file named 'ProductSale_modified.py'. The code defines a 'Sale' class with a 'getSaleNumber' property and creates several 'Product' and 'Sale' instances. The output in the terminal shows the inventory for each product instance after the sales are processed.

```
32 class Sale:
47
48     @property
49     def getSaleNumber(self) -> int:
50         return self.__saleNumber
51
52
53
54 productOne = Product(sale=None)
55 productTwo = Product(sale=None)
56 productThree = Product(sale=None)
57
58 saleOne = Sale([productOne, productTwo])
59 saleTwo = Sale([productOne])
60 saleThree = Sale([productTwo])
61 saleFour = Sale([productOne, productTwo, productOne])
62 saleFive = Sale([productTwo, productTwo])
63
64 print(f"Product One Inventory: {productOne.getInventory} \nProduct Two Inventory: {productTwo.getInventory} \nProduct
65
PS C:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Class Modeling -- Homework> python -u "c:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Class Modeling -- Homework\ProductSale_modified.py"
Product One Inventory: 4
Product Two Inventory: 5
Product Three Inventory: 0
PS C:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Class Modeling -- Homework>
```

Figure 3.3: Screenshot showing the output of the code. As seen, the inventory for each product instance is accounted for according to the sales.

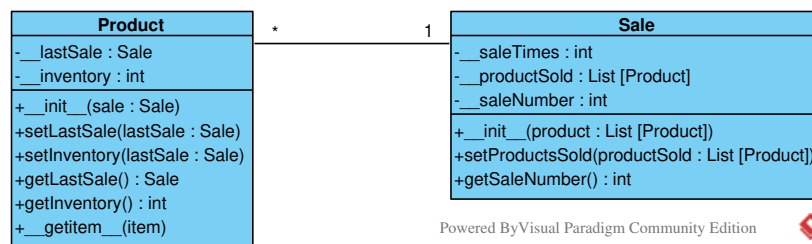


Figure 3.4: UML Class Diagram for the ProductSale code. Includes the Product and Sale classes, member variables for each class, updated functions, and the proper multiplicities.

Chapter 4

Software Execution Models

– Jason McCauley

4.1 Exercise 5.1

Demand for Each Software Component

Work	0.12
DB	6.47
Msg	0.013

Figure 4.1: Table showing the calculated demand time for each node.

To begin calculating the best, worst, and average demand times of the software execution model in Figure 5.6 of the spec sheet, I began by calculating the demand for each software component, namely Work, DB, and Msg. To do this, I referenced the table in Figure 5.6 of the spec sheet, and for each software component, multiplied the demand times by the corresponding number of hardware units and service times, summing the results.

Demand For Each Node

P_0	2.7975
P_1	0.493
P_3	0.013
P_5	13.099
P_6	0.013
P_7	6.47
P_10	7.683
P_11	0.013
P_12	13.352
P_13	20.383

Figure 4.2: Table showing the calculated demand time for each node.

From here, I was rather easily able to calculate the demand time for each node in the figure, as each node is merely calling each software component a number of times. For instance, node P_1 calls the Work component four times and Msg once, so to find the demand time of the node, I would just multiply the number of calls by the demand time of each software instance and sum. I did this process throughout the entire figure and found the demand time for each node P_1 through P_13. Node P_0 was only a slight exception, where I had to refer to the top of the table in Figure 5.6. From there, I added the demand times of each software function call together, multiplied by the corresponding number of hardware units and service time, and added them together.

Calculation Times

Worst	87.672
Best	19.354
Average	57.69301

Figure 4.3: Table showing the calculated worst, best, and average execution times.

Calculating the worst execution time was a rather straightforward process, and mostly a matter of following the path of software execution. At the case node, I would have to consider which path would lead me to the nodes with the highest total execution time, which ended up being the path to node P_13. With this path in mind, it was just a matter of iterating through the software execution model and adding the demand times of each node to find the worst execution time. Finding the best execution time was the exact same thought process, but instead at the case node, finding the path that would lead me to the nodes with the lowest total execution time. At the case node, this ended up being the path to node P_1 and then node P_11 at the split node. Again from here, it was just a matter of summing the demand times of each node on that optimal path. Finding the average execution time was a bit more involved, as I had to consider the probabilities of split nodes. I began by summing the demand times like normal, but at the case node, had to multiply the demand time of each branch by their corresponding probability. For the pardo node, I had to consider the

node that would take longer to execute, as that has to finish before proceeding. At the split node following P_1, the three processes run in parallel, so to find the average demand time of the split node, I multiplied the demand time of each process equally by 0.33 and added them together. Aside from that, calculating the rest of the node demand times was a relatively straightforward process, and just a matter of making sure that the probabilities at case nodes were being applied to the entire branches that followed, not just the subsequent node.

Total Hardware Times

CPU	53.8178
Disk	347.504
Network	18.74

Figure 4.4: Table showing the calculated total demand time for each hardware component.

Finding the total demand time for each hardware component was certainly a tedious process. I began with CPU, and for each node, I considered how many times each software component was called. If we were to take node P_1, for instance, the Work component was called four times, and the Msg component was called once. Then, for each software component, I referenced the table in Figure 5.6 to find the corresponding demand time separated by hardware instance. So for CPU, I would multiply the demand time for each software component by the number of hardware components, which in this case, was three, and then multiply by the service time, to get the CPU demand time for each software component. Now, I was able to traverse through the software execution model, and for each node, multiply the CPU demand time for each software component depending on how many times that software component was called, and add them together to get the total CPU demand time for that node. Additionally, since we are finding total, worst-case CPU time, we have to consider every possible node when traversing through the software execution model, and disregard the probability of reaching certain nodes. I then repeated this exact same process for the Disk and Network components, and after a lot of careful multiplication, successfully found their total hardware demand times as well.

4.2 Exercise 5.2

Demand for Each Software Component

Work	0.0075
DB	0.58
Msg	0.1575

Figure 4.5: Table showing the calculated demand time for each software component

To begin calculating the worst, best, and average demand times for the given software execution model, I wanted to reduce overhead as much as possible. To do this, I began by calculating the demand time for each software component. So for instance, a call to the database – I referenced the

table in Figure 5.7 of the spec sheet, and for DB software component, multiplied each value by the corresponding number of hardware instances and service time, summing them together. This same process would be repeated for the Work and Msg software components, finding the total demand time for each call.

Demand for Each Node	
initialize	0.7525
updateMirrorNY	1.5125
updateMirrorLA	1.505
queryMirrorNY	0.9175
queryMirrorLA	0.925
writeMirrorNY	0.9175
writeMirrorLA	0.925

Figure 4.6: Table showing the calculated demand time for each node.

Next, I found the demand time for each node – if we take the updateMirrorNY node for instance, it has 5 calls of Work, 2 calls of DB, and 2 calls Msg. Finding the demand time of each node was just a matter of multiplying the number of calls by the demand time for each software component, summing them together. This process was repeated for each node, multiplying by number of software calls by their demand times and summing.

•

Calculation Times	
Worst	4.53
Best	3.34
Average	3.93725

Figure 4.7: Table showing the calculated worst, best, and average execution times

From here, finding the worst, best, and average execution times was a pretty straightforward process. For the worst execution time, it was just a matter of following the path of nodes. Wherever the diagram split, I had to calculate the total demand time of that branch, and follow the one with the highest demand time, which ended up being the "update" node, which leads to another case node. This process was then repeated to find the best execution time, but following the path of the nodes that led to the lowest demand times. This ended up being the "query" node, as the split node it leads to allowed it to proceed just a tiny bit faster than the "write" node, which leads to a pardo node. Now for calculating the average demand time, the process was quite similar, traversing through the path of the nodes, however, we had to handle the splits different. At the first case node, we had to multiply the entire following branches by the probability, finding the weighted demand time of each branch. Additionally, we had to calculate the demand time of the contents

of the branch. For the "query" node, it was another case node, multiplying each node by the corresponding probability. For the split node, the two query nodes runs in parallel – therefore, we weigh them equally, multiplying their demand times by 0.5 and summing them to find an average. Finally, for the pardo node, we only consider the path leading to the node with the highest demand time, which is writeMirrorLA, as it must complete before the node proceeds. From here, it is just a matter of following the path of nodes, summing their demand times, and finding weighted demand times at the necessary splits to calculate the average execution time.

4.3 Exercise 5.3

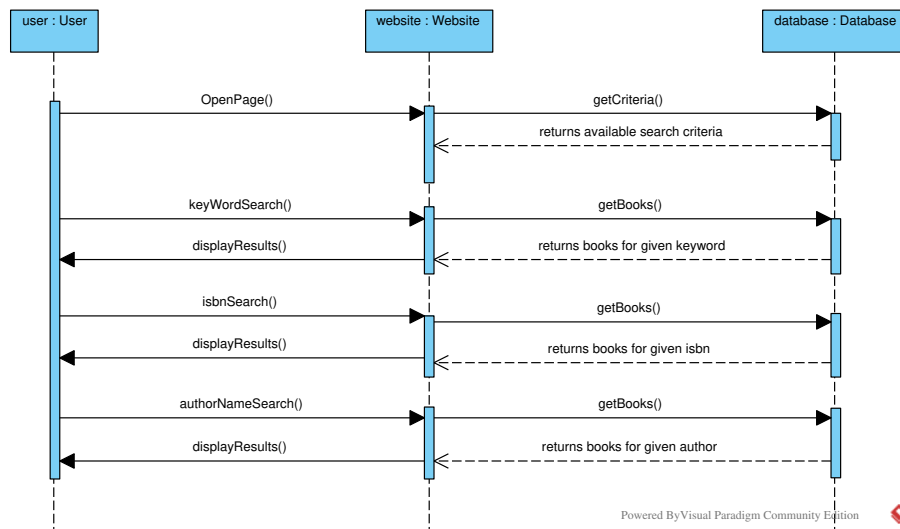


Figure 4.8: Sequence diagram showing how the three types of searches are conducted.

The sequence diagram has three lifelines, each being an instance of the User, Website, and Database classes respectively. Beginning at the user lifeline, the user can open the page, which will cause the website to query the database and return the possible search criteria. Then, on the website, the user can search by either keyword, isbn, or author name. Regardless of the method they chose, the website will then use the input to query the database through the `getBooks` function. The database will then return the books corresponding to search method, and the `displayResults` function will display the returned books for the user to see.

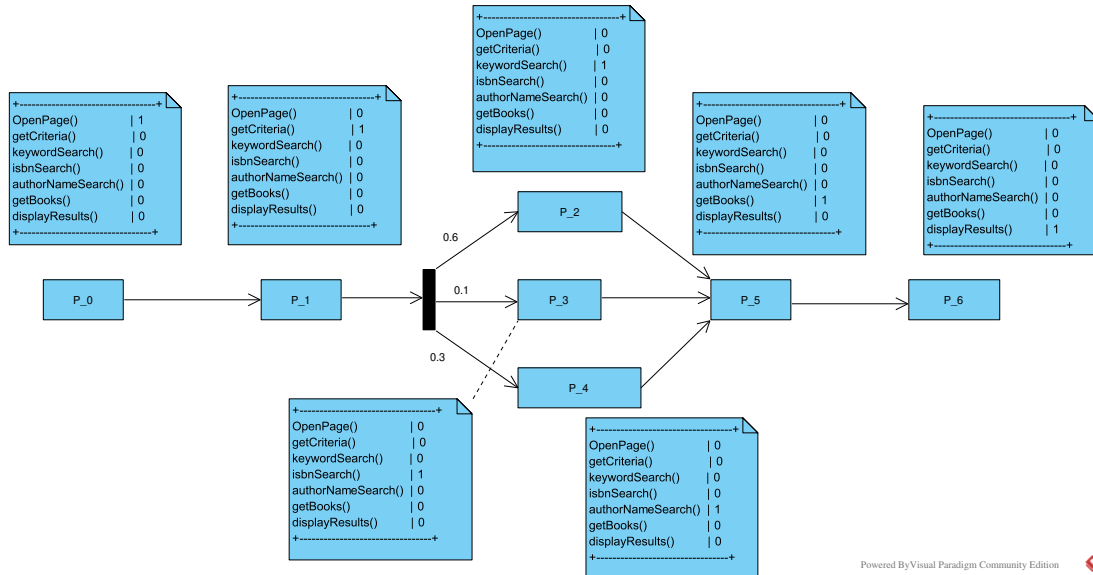


Figure 4.9: Corresponding software execution diagram for the sequence diagram, with demand weights assigned at each node.

The software execution diagram begins at node P_0, where there will be a call to open page. Subsequently, at node P_1, there will be a call to get criteria to be searched, in this case, the ways the book can be queried in the database. From there we have a case node, with probabilities that lead to three different nodes. These three nodes, P_2, P_3, and P_4 contain the calls to each of the possible search functions, such as search by isbn. After the books are queried, node P_5 contains the call to the getBooks function, which will return the books that match the entered search criteria. Finally, that node leads to P_6, which simply contains the call to displayResults, allowing the user to see which books match their entered search.

Demand for Each Node

P_0	0.2
P_1	2.22
P_2	40
P_3	0
P_4	0
P_5	0
P_6	0.2

Figure 4.10: Demand time for each node of the software execution diagram.

To begin calculating the worst, best, and average demand times, I calculated the demand time for each node, as seen in the table above. To do this, I had to consider which software components were being used in each node. For instance, P_0 utilizes the OpenPage component. From there, I referenced Table 5.1 in the spec sheet to multiply each demand time for that software component

by the corresponding number of hardware instances and service time. Summing those together, I found the total demand time for that node.

Calculation Times

Worst	42.62
Best	2.62
Average	26.62

Figure 4.11: Worst, best, and average calculation times for the software execution diagram.

After repeating this process for each node, I had to calculate the worst, best, and average execution times. To calculate the worst time, I followed the path of the nodes, and at the case node, selected the path that led to the node with the highest demand time. Adding the demand times for each node on this path, I found the worst execution time. This process was then repeated to find the best execution time, but at the case node, selecting the path that led to the node with the lowest demand time. Finding the average execution time is a very similar process, summing the demand time of each node – however, once we reach the case node, we multiply the demand time of each node by their corresponding probability to find each weighted demand time. From there, we continue adding the demand time of each node like normal, and find the average execution time.

Chapter 5

Exam One Extra Work

– Jason McCauley

5.1 Exercise 5.6

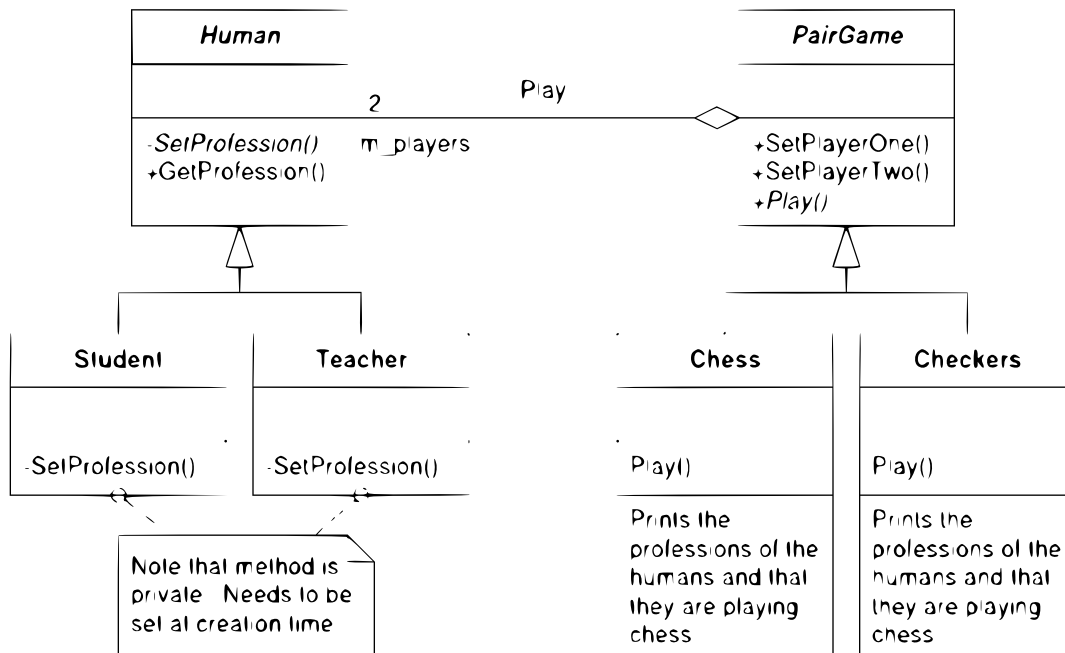


Figure 5.1: UML diagram for implementation.

```
1 #include <iostream>
2 #include <string>
3 using namespace std;
4
5 class Human {
6     public:
7         string GetProfession() {
8             return profession;
```

```
9     }
10    protected:
11        Human() {}
12        string profession;
13    private:
14        virtual void SetProfession() = 0;
15};
16
17class Student: public Human {
18    private:
19        void SetProfession() override {
20            profession = "Student";
21        }
22    public:
23        Student(): Human() {
24            SetProfession();
25        }
26};
27
28class Teacher: public Human {
29    private:
30        void SetProfession() override {
31            profession = "Teacher";
32        }
33    public:
34        Teacher(): Human() {
35            SetProfession();
36        }
37};
38
39class PairGame {
40    protected:
41        Human* playerOne;
42        Human* playerTwo;
43    public:
44        PairGame(): playerOne(nullptr), playerTwo(nullptr) {}
45
46        void SetPlayerOne(Human* player) {
47            playerOne = player;
48        }
49        void SetPlayerTwo(Human* player) {
50            playerTwo = player;
51        }
52        virtual void Play() = 0;
53};
54
55class Chess: public PairGame {
56    public:
57        void Play() override {
58            cout << "Playing Chess with " << playerOne->GetProfession() << "
59            and " << playerTwo->GetProfession() << endl;
60};
```

```
61
62 class Checkers: public PairGame {
63     public:
64         void Play() override {
65             cout << "Playing Checkers with " << playerOne->GetProfession() <<
66             " and " << playerTwo->GetProfession() << endl;
67         }
68 };
69
70 int main() {
71     Student student;
72     Teacher teacher;
73     Chess chessGame;
74     Checkers checkersGame;
75
76     chessGame.SetPlayerOne(&student);
77     chessGame.SetPlayerTwo(&teacher);
78     chessGame.Play();
79
80     checkersGame.SetPlayerOne(&student);
81     checkersGame.SetPlayerTwo(&teacher);
82     checkersGame.Play();
83
84     return 0;
85 }
```

Listing 5.1: Code for TeacherStudentGame.cpp, which includes all classes and code driver for instantiating student and teacher, instantiating chess and checkers, setting the student and teacher as players, and playing both games.

Chapter 6

Decorator Pattern

– Jason McCauley

6.1 Exercise 14.1

```
1 public class MargheritaPizza implements Pizza {
2     @Override
3     public String getDescription(){
4         return "Margherita Pizza";
5     }
6
7     @Override
8     public double getCost(){
9         return 6.99;
10    }
11 }
```

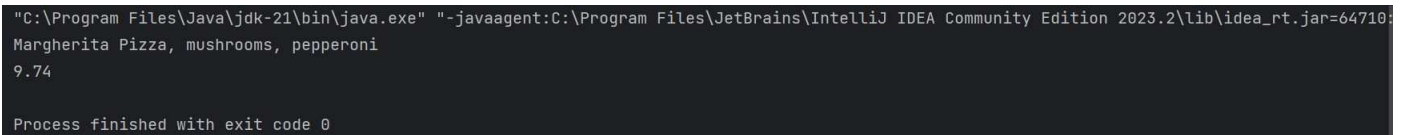
Listing 6.1: The concrete class MargheritaPizza, which prints "Margherita Pizza" and has a cost of 6.99.

```
1 public class MushroomDecorator extends ToppingDecorator {
2     public MushroomDecorator(Pizza pizza){
3         super(pizza);
4     }
5
6     @Override
7     public String getDescription(){
8         return pizza.getDescription() + ", mushrooms";
9     }
10
11    @Override
12    public double getCost(){
13        return pizza.getCost() + 1.25;
14    }
15 }
```

Listing 6.2: The concrete class MushroomDecorator, which appends the text "mushrooms" to the description and adds 1.25 to the price.

```
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 }
```

Listing 6.3: The concrete class `PepperoniDecorator`, which appends the text "pepperoni" to the description and adds 1.50 to the price.



```
"C:\Program Files\Java\jdk-21\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA Community Edition 2023.2\lib\idea_rt.jar=64710:  
Margherita Pizza, mushrooms, pepperoni  
9.74  
  
Process finished with exit code 0
```

Figure 6.1: Screen-shot of the output results for the above code.

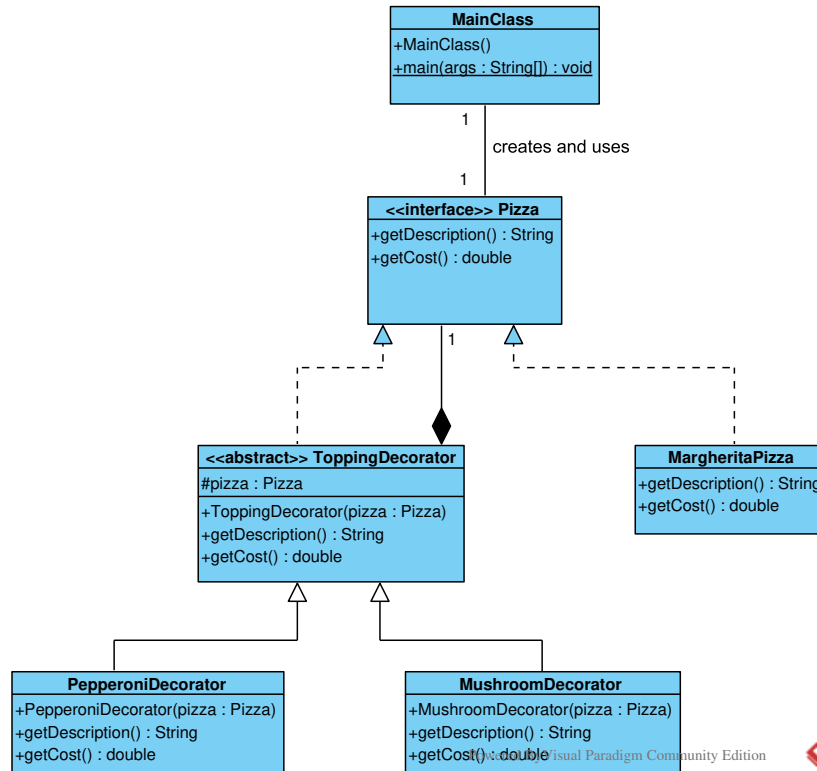


Figure 6.2: Complete UML class diagram for the above code.

Chapter 7

Visitor and Singleton Patterns

– Jason McCauley

7.1 Write a Program to Implement Singleton Pattern

```
1 public class SingleObject{
2     // create an object of SingleObject
3     private static SingleObject instance = new SingleObject();
4     // make the constructor private so that this class cannot be instantiated
5     private SingleObject(){}
6     // get the only object available
7     public static SingleObject getInstance(){
8         return instance;
9     }
10    public void showMessage(){
11        System.out.println("Hello World!");
12    }
13 }
```

Listing 7.1: SingleObject.java, the Singleton Class

```
1 public class SingletonPatternDemo{
2     public static void main(String[] args){
3
4         // illegal construct
5         // Compile Time Error: The constructor SingleObject() is not visible
6         // SingleObject object = new SingleObject();
7
8         // Get the only object available
9         SingleObject object = SingleObject.getInstance();
10
11        // show the message
12        object.showMessage();
13    }
14 }
```

Listing 7.2: SingletonPatternDemo.java, which gets the only object of the singleton class

```
C:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Singleton and Visitor Pattern -- Homework\Part1>javac *.java
C:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Singleton and Visitor Pattern -- Homework\Part1>java SingletonPatternDemo
Hello World!
```

Figure 7.1: Output after compiling all files and running SingletonPatternDemo.java

7.2 Write a Program to Implement Visitor Pattern

```
1 public interface ComputerPart{
2     public void accept(ComputerPartVisitor computerPartVisitor);
3 }
```

Listing 7.3: ComputerPart.java, which is an interface to represent element

```
1 public class Keyboard implements ComputerPart{
2     @Override
3     public void accept(ComputerPartVisitor computerPartVisitor){
4         computerPartVisitor.visit(this);
5     }
6 }
```

Listing 7.4: Keyboard.java, which is a concrete class extending ComputerPart.java

```
1 public class Monitor implements ComputerPart{
2     @Override
3     public void accept(ComputerPartVisitor computerPartVisitor){
4         computerPartVisitor.visit(this);
5     }
6 }
```

Listing 7.5: Monitor.java, which is a concrete class extending ComputerPart.java

```
1 public class Mouse implements ComputerPart{
2     @Override
3     public void accept(ComputerPartVisitor computerPartVisitor){
4         computerPartVisitor.visit(this);
5     }
6 }
```

Listing 7.6: Mouse.java, which is a concrete class extending ComputerPart.java

```
1 public class Computer implements ComputerPart{
2     ComputerPart[] parts;
3     public Computer(){
4         parts = new ComputerPart[]{new Mouse(), new Keyboard(), new Monitor()}
5     };
6
7     @Override
8     public void accept(ComputerPartVisitor computerPartVisitor){
```

```

9      for(int i = 0; i< parts.length; i++){
10         parts[i].accept(computerPartVisitor);
11     }
12
13     computerPartVisitor.visit(this);
14 }
15 }

```

Listing 7.7: Computer.java, which is a concrete class extending ComputerPart.java

```

1 public interface ComputerPartVisitor{
2     public void visit(Computer computer);
3     public void visit(Mouse mouse);
4     public void visit(Keyboard keyboard);
5     public void visit(Monitor monitor);
6 }

```

Listing 7.8: ComputerPartVisitor.java, which is an interface to represent the visitor

```

1 public class ComputerPartDisplayVisitor implements ComputerPartVisitor{
2     @Override
3     public void visit(Computer computer){
4         System.out.println("Displaying Computer.");
5     }
6
7     @Override
8     public void visit(Mouse mouse){
9         System.out.println("Displaying Mouse.");
10    }
11
12    @Override
13    public void visit(Keyboard keyboard){
14        System.out.println("Displaying Keyboard.");
15    }
16
17    @Override
18    public void visit(Monitor monitor){
19        System.out.println("Displaying Monitor.");
20    }
21 }

```

Listing 7.9: ComputerPartDisplayVisitor.java, which is a concrete visitor implementing ComputerPartVisitor.java

```

1 public class VisitorPatternDemo{
2     public static void main(String[] args){
3         ComputerPart computer = new Computer();
4         computer.accept(new ComputerPartDisplayVisitor());
5     }
6 }

```

Listing 7.10: VisitorPatternDemo.java, which uses ComputerPartDisplayVisitor.java to display parts of Computer

```
C:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Singleton and Visitor Pattern -- Homework\Part2>javac *.java
C:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Singleton and Visitor Pattern -- Homework\Part2>java VisitorPatternDemo
Displaying Mouse.
Displaying Keyboard.
Displaying Monitor.
Displaying Computer.
```

Figure 7.2: Output after compiling all files and running VisitorPatternDemo.java

7.3 Setting up the Visitor Pattern in Javascript

```
1 var CarVisitor = function(){
2     var visit = function(carVariable){
3         //do some operations on carVariable
4     }
5 }
6
7 var TruckVisitor = function(){
8     var visit = function(truckVariable){
9         //do some operations on truckVariable
10    }
11 }
12
13 var MonsterTruckVisitor = function(){
14     var visit = function(monsterTruckVariable){
15         //do some operations on monsterTruckVariable
16     }
17 }
18
19 var carVariable = function(){
20     var seats = 5;
21     var doors = 4;
22     this.accept = function(visitorObject){
23         visitorObject.visit(this);
24     }
25 }
26
27 var truckVariable = function(){
28     var towPackage = true;
29     var doors = 2;
30     this.accept = function(visitorObject){
31         visitorObject.visit(this);
32     }
33 }
34
35 var monsterTruckVariable = function(){
36     var looksLikeADragon = true;
37     var doors = 1.5;
38     this.accept = function(visitorObject){
39         visitorObject.visit(this);
40     }
41 }
```

```

41 }
42
43 var CarVisitor = function(){
44     this.visit = function(car){
45         if(car.seats > 2){
46             console.log("this is clearly a car for old people");
47         }
48         else{
49             console.log("My bet is this car has at least 2 cylinders");
50         }
51     }
52 }
53
54 var TruckVisitor = function(){
55     this.visit = function(truckVar){
56         if(truckVar.towPackage){
57             console.log("we need to buy a boat");
58         }
59     }
60 }
61
62 var MonsterTruckVisitor = function() {
63     this.visit = function(monsterTruckVar) {
64         if(monsterTruckVar.looksLikeADragon){
65             console.log("that is a badass monster truck");
66         }
67         else{
68             console.log("loser");
69         }
70     }
71 }
72
73 var myCar = new carVariable();
74 myCar.seats = 2;
75 myCar.accept(new CarVisitor());
76
77 var myMonsterTruck = new monsterTruckVariable();
78 myMonsterTruck.looksLikeADragon = false;
79 myMonsterTruck.accept(new MonsterTruckVisitor());
80
81 var myCar2 = new carVariable();
82 myCar2.seats = 2;
83 myCar2.accept(new MonsterTruckVisitor());

```

Listing 7.11: visitorDemo.js, which defines carVariable, truckVariable, and monsterTruckVariable, and adds functionality to the visitors


```
C:\Users\jsonm\OneDrive - stevens.edu\Documents\SSW_345\Singleton and Visitor Pattern -- Homework\Part3>node visitorDemo.js  
My bet is this car has at least 2 cylinders  
loser  
loser
```

Figure 7.3: Output after running visitorDemo.js in terminal