2020 USA Regional Round
The International Mathematical Modeling Challenge
Summary

It is the primary goal of any retail store to gain as much revenue as possible in the shortest period of time. A common solution to this is enticing customers with deals and discounts on popular products, meaning that events such as flash sales are considered highly effective and popular among retail stores. From Black Friday deals to store-specific events, flash sales are a natural staple of retail life. Each year, flash sales like these bring in thousands of customers seeking good discounts and quality products from their favorite stores. Whether it be the days before a major holiday or a store-exclusive promotion, flash sales are a crucial staple of brick-and-mortar retail (and, recently, even online shopping).

Although such sales can bring in a lot of money for any store, they can also get rather chaotic. Black Friday, for example, often fills the news with reports of breakout fights and stores filled to the brim with customers hoping to buy as many products for as little money as possible. This hectic environment means that product damage is almost inevitable. Damages can take a large chunk of potential revenue from a store, as damaged products cannot be sold, which becomes a major issue for any store hoping to host a flash sale. This means, for many stores, the question is: how can they maximize revenue with a flash sale while minimizing damage?

In this paper, the team first identified the methods by which a product could be damaged as dropping, package opening, and rough handling. Then, popularity of an item was quantified and the most popular products and product types identified. Using this, and a series of factors including population density, product density, obstruction from the view of cashiers, and total customers, a scoring system was developed that could predict the number of damages due to each method of damaging product. Finally, this model was put through two simulations. The first simulation uses Java to create a map of the store and a number of total products in a given department, then simulates customers buying and damaging items based on the calculations from the scoring system. The second simulation uses Python to create a rearrangeable model of the store, where each department can be divided, rearranged, or otherwise modified to create a new floor plan based on the amount of space taken up by products. The team used these simulations to determine the optimal layout for an electronics and appliances store during a flash sale in order to avoid product damages.

Introduction

In the retail world, flash sales are popular for their ability to bring in customers and, in turn, unusually high store revenue. Unusually low prices and popular products draw in many more customers than normal, meaning more purchasing of store products, but also more chaos. Whether a customer knocks over an item in the scramble to get what they want or takes a peek inside a package that they shouldn't in hopes of confirming the product is intact, flash sales also bring in unfortunate damages to products storewide alongside a spike in sales.

In this paper, a model is introduced that scores the floor plan of a store based upon its ability to deter damage to its product. These scores are based upon various factors, including popularity, how well a cashier can see each department, and multiple other factors. Then, using this scoring system, a floor plan can be run through two different simulations: one that simulates the sale and damages of products over a certain period of time, and one that allows the user to rearrange the floor plan and test how many damages are produced during the whole sale. Using this model and these two simulations, it is possible to find the optimal placement of product departments and the optimal store floor plan.

The Problem

The team is tasked with investigating the ability of the floor plan of an electronics and appliances retail store in helping to minimize damages in its upcoming flash sale. Having been provided with the original floor plan, a list of departments, and a list of products, the team is asked by the store's manager to first evaluate where each department should be located in the original floor plan, and then to create their own floor plan that results in the least accidental damage to goods in the store.

Assumptions

1. **Damaged goods do not include stolen or intentionally broken products.** The problem states that the floor plan is meant to account for accidental damages. Therefore, certain behaviors can be intentional, but the actual damaging of the product should not be.
2. **The only employees on the floor during the flash sale are the cashiers.** Due to the number of people that would be visiting the store during the flash sale, lines are likely to be much longer than normal. As such, as many employees as possible would be needed at cashier stations in order to keep customers from waiting too long to check out.
3. **The maximum number of people that can be looking at one product is fifteen.** Ten people looking at the same product would be a large number, likely preventing some of those ten from actually accessing the product they're looking for. However, flash sales will create a larger number of people in the store in general, so this number would become higher. Even then, not many others would be able to gain access

to the product anyway, so 150% of the typical maximum— that is, fifteen people instead of ten— is reasonable.

4. **Any item can be damaged in the same ways as any other item.** Typically, it would be harder to break a washer or dryer than it would be to break headphones or a cell phone. However, with a flash sale comes additional chaos as everyone runs around to get to the products they want, and so the larger number of people creates greater chances of damaging larger or sturdier products.
5. **The general size of a type of product is important, but the specific sizes of each product is irrelevant.** In general, the area needed to hold washers or dryers will be larger than the area needed to hold headphones, because these products are different sizes. However, the area for each specific product doesn't matter, because the model works upon a 2D floor plan. Therefore, the model doesn't include how many shelves are on each shelf, so it can be assumed that, while a smaller area would naturally fit headphones better than washers, the specific space required to fit every single pair of headphones isn't crucial to the problem.

Problem 1: Damage Types and Product Popularity

Describe the various ways in which products at the store might be damaged during the sale event due to careless and accidental actions of the customers.

Inventory shrinkage, or product loss, is responsible for over 1.3% loss in sales (1), and a deficit of over $44 billion globally. Although shrinkage is often associated with employee theft, shoplifting, and other forms of fraud or theft, accidental and careless product damage is part of 7-10% of inventory shrinkage not from these factors (2). Moreover,this contributes to 37% of inventory shrinkage between the mass flash sales Black Friday and Christmas Eve shopping (3). Although methods of reducing shrinkage due to theft or fraud are often well-addressed, shrinkage due to accidental mishandling and product damage are often discounted and underestimated.

There are several ways in which products can get damaged in a flash sale due to careless and accidental actions of the customers. A common way is damage of the product from accidentally knocking it over or knocking it out of place, like off of a shelf. This can happen often when the number of products is particularly dense in a certain area and when the product type is larger or composed of more fragile material. Heavier objects, like televisions and large laptops, experience larger forces upon impact with another surface because of their weight. On top of that, the screens of these products are made of glass, which is more likely to break than plastic or metal, although both of the latter can be dented, bent, or otherwise damaged. Lighter objects, like headphones, experience smaller forces upon impact with another surface for the same reason. Typically the casings of headphones are also made of plastic, which is less likely to break than glass. That means that placing smaller objects with sturdier materials at taller heights and larger objects with more fragile materials at lower heights would create less probability of product damage, while the inverse would create a higher probability.

Rough handling can damage products as well. This involves accidentally or carelessly holding or transporting objects in a way that can damage them. This can also happen if a product is stepped on, tossed around, stuck underneath a heavier object, or otherwise handled poorly. Smaller products, however, can usually

sustain more rough handling. This is because smaller products are often lighter than larger ones and will, therefore, sustain less force upon impact. For example, headphones, even if not packaged, can be roughly handled but still not damaged because they commonly have a smaller mass, so even if they are dropped or pushed around, the force on them would be less. However, if a large object like a TV, PC, or even a laptop is pushed around, because the mass of these objects is typically larger, these objects would experience a larger force with the same amount of rough handling. Moreover, in an electronics store, certain products have more fragile components than others, and they may be more prone to damage due to rough handling. Electronics with glass or loose-wire components are more prone to breakage. It then becomes important to ensure that larger objects or objects with fragile components are less likely to be rough-handled.

Another way a product can get damaged is if a customer opens the packaging. This often prevents the product from being used for display, as the product is more likely to be damaged when taken out of its packaging. This would be especially problematic for the sale of electronics and appliances, as is the primary focus of the store in question, because many of these products are already fragile, and once they are taken out of their packaging they are most likely deemed unsuitable for sale. Larger items would not only take more time to open and check, but would also likely come with more security in an electronics and appliances store. Also, people are more likely to open these products in an area where they generally cannot be seen by employees at the store and be stopped. Customers are not supposed to open the packaging on a product, so employee surveillance will deter them from opening products so as to not get in trouble, and it will additionally allow employees to prevent the customer from opening anything. Smaller items are more likely to be unpackaged, since larger objects are typically displayed in-store but not always accessible in packaging to customers without request, and thus in an electronics store items including headphones, cameras, small laptops, and occasionally larger devices, like microwaves, would be more likely to be unpackaged. Therefore, placing smaller items closer to the cashier and other employee-dense areas would lessen the number of opened products in the store.

Consider the items included in this flash sale event. Which sale items do you think will be most popular (most desired) by the shoppers and why?

In a flash sale event, popularity of a product depends on several factors. The team decided to incorporate three factors deemed most significant in determining the popularity of the products.

The team's model for the popularity of specific items can be broken down to three factors: the lifespan of specific products, the overall customer satisfaction, and the price reduction of the product. These three factors were incorporated into a scoring system that produced a "popularity percentage." The popularity percentage represented the percentage of people that would be willing to look into a certain product given its score. Although the popularity depends on lifespan, ratings, and markdown, these do not contribute to the popularity of a product equally. The team chose weights for each of the factors based on the importance of each. The lifespan was given the most weightage (60%) because a customer would be most likely to buy a certain type of product if they needed to replace it, regardless of that type of product's general price or ratings for each individual model. Next, the percent markdown was given a high weightage (30%) because in flash sales, most consumers are looking for products that minimize their total spending. Lastly, the customer rating was given the least weightage (10%) because although the rating of a product is an important factor in product popularity,

during a flash sale most customers are focused on buying products they need or products that are significantly marked down regardless of ratings. In addition, all the ratings were relatively high as they were above 4 out of 5 stars. A summary table of the weights used is given below:

Popularity Factors: Weights	
% weight customer rating	0.10
% weight markdown	0.30
% weight lifespan	0.60

Table 1: A summary table of the weights used for each of the factors in the popularity of a product.

In order to convert each of these factors into an individualized popularity percentage, the team scored each product as a weighted percentage of each of the factors, and combined the weighted percentages of each of the factors to calculate a final popularity score. The weighted percentage for customer rating was calculated by dividing the customer rating by 5, and is shown in and multiplying this number by the customer rating weight of 0.1. This is shown in Equation A:

$$RATING\ (\%) = \frac{RATING_{customer}}{5} * 100 * 0.1 \quad \textit{(Equation A)}$$

The weighted percentage for the markdown was calculated by dividing the markdown price by the regular price of the item, and multiplying this number by the markdown weight of 0.3. This is shown in Equation B:

$$MARKDOWN\ (\%) = \frac{PRICE_{retail} - PRICE_{flash\ sale}}{PRICE_{retail}} * 100 * 0.3 \quad \textit{(Equation B)}$$

The weighted percentage for lifespan was simply 1 divided by the total lifespan of the object in years multiplied by the lifespan weight of 0.6 (this represented the probability of a customer needing to replenish an item that had past its lifespan during the flash sale, as the team assumed that the flash sale, on the order of Black Friday, was an annual event). This is shown in Equation C:

$$LIFESPAN\ (\%) = \frac{1}{LIFESPAN_{average}} * 100 * 0.6 \quad \textit{(Equation C)}$$

Then, the weighted percentage for customer rating, the weighted percentage for markdown, and the weighted percentage for lifespan were combined to give the overall "popularity percentage" of the product. This popularity percentage was calculated for all products being sold on sale at the electronics store during the flash sale. This is shown in Equation D:

$$P_{Popularity} = RATING\ (\%) + LIFESPAN\ (\%) + MARKDOWN\ (\%) \quad \textit{(Equation D)}$$

The general popularity percentages in each department were summed up to obtain the total popularity percentage. The average popularity percentages were obtained by dividing the total popularity percentages by the number of products in each department. These average popularity percentages were then normalized with respect to each other so all percentages added to 100% and could be used to analyze the distribution of people in the flash sale. The normalized percentages were calculated by taking each average popularity percentage and dividing it by the sum of the average popularity percentages across all departments and then multiplying by 100. Table 3 shows the department (column 1), total popularity percentages (column 2), average popularity

percentages (column 3), and the normalized percentages (column 4). Table 2 shows a sample of 5 products and their popularity percentages are shown below:

Score Table				
Product Type	**Weighted Customer Rating (%)**	**Weighted Markdown (%)**	**Weighted Lifespan (%)**	**Popularity Percentage (%)**
Dryer	9.2	7.9	6	23.1
Washer	8.8	8.6	4.8	22.2
Cameras	9.8	10	15	34.8
Headsets	8.2	6	20	34.2
Gaming Desktop	9.4	7.1	24	40.5

Table 2: A sample table of the weighted percentage for customer rating, the weighted percentage for markdown, the weighted percentage for lifespan, and the popularity percentage for 5 randomly selected products from the electronics store.

Based on this model, the most popular product would be: *15.6" Gaming Laptop, AMD Ryzen 5, 8GB Ram, NVIDIA GeForce GTX 1050, 25* gaming laptop (popularity score of 47.1), and the least popular product would be the *26.2cu ft French Door Smart Wi-Fi Enabled Refrigerator, PrintProof, Black Stainless* refrigerator (popularity score of 19.6).

In general, the popularity of different categories seem to be as follows:

1. Headphones (42.4), PC Gaming (41.7)
2. Laptops (39.2), Cell Phones and Accessories (35.7)
3. Tablets (35.4)
4. Desktops and All-in-Ones (34.7)
5. Printers (32.2)
6. Vacuum Cleaners and Floor Care (32.1)
7. Video (31.5)
8. Mirrorless Cameras (30.7)
9. DSLR Cameras (29.9)
10. Monitors (28.3)
11. TVS 30” to 45” (28.0)
12. Console Game Systems (26.5)
13. TVS 50” to 55” (24.4)
14. TVS 65” (24.1)
15. TVS 85” (23.6)
16. Major Kitchen Appliances (23.4)
17. TVS 70” to 75” (23.0)
18. and Laundry Appliances (22.9).

Grouped Percentage			
Department	Total %	Average %	Normalized %
Appliances	781.6	25.21	10.87
Audio	84.8	42.40	18.29
Cameras	302.3	30.23	13.04
Cell Phones	107.0	35.66	15.38
Computers & Tablets	1420.5	35.51	15.32
TV & Home Theater	903.4	25.81	11.13
Video Gaming	481.3	37.02	15.97

Table 3: The average popularity percentages and normalized average popularity percentages for each department.

Problem 2: Store Layout

Use your responses to Requirement 1 to identify and describe the store layout factors that impact possible damage to products and other measures you deem important during a flash sale event.

The team determined six factors that would be most influential on the probability of product damage. These are shown below:

1. Total Visitors: The total number of customers visiting a given department in a given amount of time. This is used as a variable, since different stores can have varying foot traffic.
2. Population Density: The average number of people per square meter in a given department. This was determined to be a vital factor because higher population densities can exacerbate product damages due to crowded spaces and multiple people trying to gain access to certain items. The model accounts for the population density in a given hour, and so the population density is calculated as follows:
   First, the average number of visitors per hour given the total number of people at the sale, the average popularity percentage as found in Section 1, and the sale time are calculated as follows:

$$n_{avg} = \frac{N * P}{t} \quad \textit{(Equation 1)}$$

   Where N is the total number of people at the sale, P is the calculated popularity percentage for the department, t is the total sale time in hours, and n_{avg} is the average number of visitors per hour.
   Then, the population density is calculated by dividing n_{avg} by the area of the department as shown on the layout. This equation is shown below, considering that A is the area of the department, and $D_{population}$ is the population density:

$$D_{population} = \frac{n_{avg}}{A} \quad \textit{(Equation 2)}$$

3. Product density: The average number of products per square meter in a given department. This was deemed important because more availability of multiple models of product will allow for more accessibility to those products, and therefore there will be less urgency and less reckless behavior in trying to access them. This is calculated similarly to population density. The product density, or $D_{product}$ is calculated using the total number of products in the department ($n_{product}$) and the area of the department on the layout (A). This equation is given below:

$$D_{products} = \frac{n_{product}}{A} \quad \textit{(Equation 3)}$$

4. Obstruction of Cashier: This is the likelihood that a cashier will be able to stop accidental damage, as a function of the distance of the department's center to the cashier's center. This model was chosen because customers would be less likely to attempt to open a package or be careless with handling products if they were closer to the cashier. Moreover, this can be turned into a score from 0-1 if it is compared to the maximum possible distance the cashier and the department can have between each other. Multiple cashiers were modeled by taking the average score and subtracting a portion of every score to ensure that all other cashiers have an influence on the final lower score. This portion of every score was obtained by multiplying all other scores by a factor c. This was chosen to be 0.05 to ensure that all cashiers are incorporated in the score without overweighting a closer cashier. Equation 4.1 indicates the obstruction $O_{cashier}$ as a function of distance d, the length l of the store, and the width w of the store is given below. Equation 4.2 shows the obstruction score $O_{cashier}$ as a function of the maximum score O_{max} , the constant c, and a function of all other obstruction scores for each cashier O_1, O_2, O_3, etc.

$$O_{cashier} = \frac{d}{\sqrt{w^2 + l^2}} \quad \textit{(Equation 4.1)}$$

$$O_{cashier} = O_{average} - c(O_1 + O_2 + O_3 + \cdots + O_n) \quad \textit{(Equation 4.2)}$$

5. Popularity Percentage: The likelihood of a person being interested in a given item. A more popular product will attract more people than a less popular product, and there will be more urgency to gain access to the more popular product. The calculations for the popularity percentage is given in Section 1.
6. People per product ratio (R): This factor is a combination of the population density ($D_{population}$) and product density($D_{product}$), as well as the maximum number of people per product(R_{max}) at the store and is a flexible factor used to account for both in the model.

$$R = \frac{D_{population}}{R_{max} * D_{product}} \quad \textit{(Equation 5)}$$

In the model, the maximum number of people per product is set as 15. This is because it was determined that ten people would be a high number of people to be looking to buy a product at one time, because any higher would limit accessibility to the product. However, because of the flash sale, it is likely that

the maximum number of people would be unusually high, and so 150% of the typical maximum was used, which would be fifteen people.

Use the factors you identified in part (a) to develop a mathematical model or models to quantitatively predict both the behaviors of the flash sale customers that potentially result in damage to products in the store and the level of that damage. Your model(s) should incorporate floor plan characteristics, location of departments, specific flash sale products, and arrangement of cashier stations.

An overview of this process is shown here:

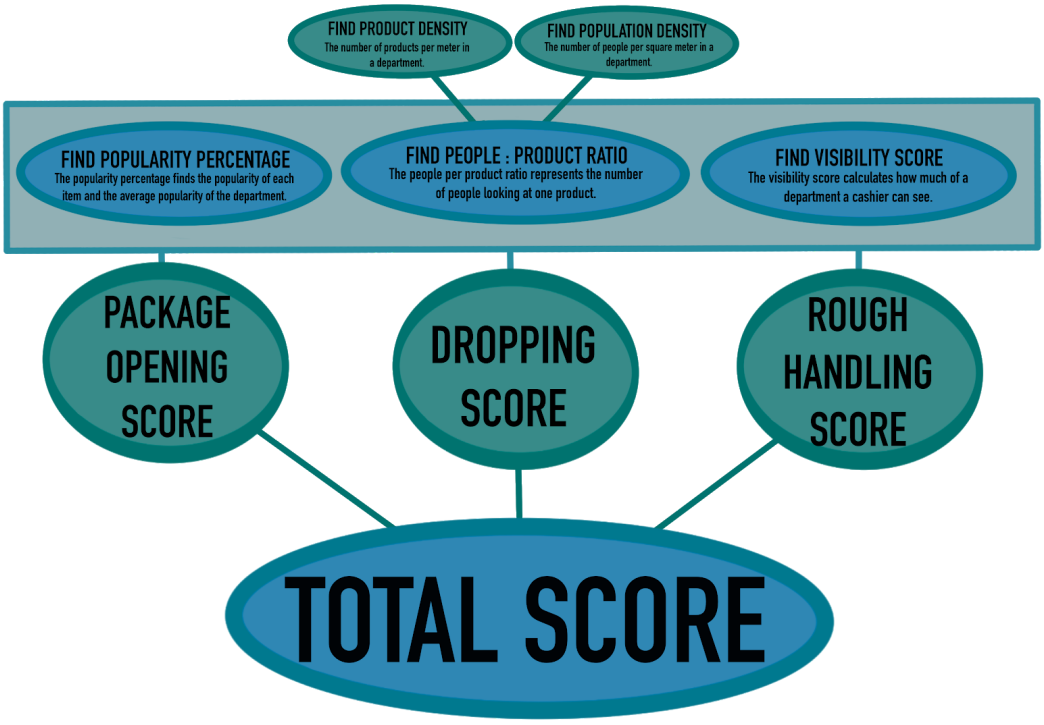

Figure 1: A flow chart representing the overall process of finding the total damageability score for a department.

Based upon the factors described in Section I, the model determines an overall "damageability" score for each department based upon three combined subscores: a dropping score, an open packaging score, and a rough handling score. Each of these represent methods by which products could be damaged, and use the given factors in calculations.

The dropping score determines the probability that products in a given department will be dropped or knocked over by customers. This score incorporates the population density, product density, total visitors during the sale, and total number of products in the department. This score was found by calculating the number of people per product, defined as the population density divided by the product density, by the maximum number

of people per product, defined as the total visitors divided by the total products in the department. The dropping score is a number from 0-1. Thus, the equation for the dropping score is given below:

$$DROPPING\ SCORE = R \quad \textit{(Equation 6)}$$

The open packaging score quantifies the likeliness of products being opened by customers in a given department. This score incorporates the population density, obstruction of cashiers, and popularity score. It is found by combining the number of people per product, as described previously in the dropping score, the popularity score, and the obstruction of cashiers in a weighted average. The open packaging score is a number from 0-1, and is a weighted average of the people to product ratio, the popularity percentage, and the obstruction factor. Obstruction is weighed at 70%, since if a person is in view of the cashiers, they're much less likely to act recklessly or open packaging. The people to product ratio is weighed at 20%, because more people trying to get access to a certain product increases the likelihood of package opening only because there are more people around to do so. Finally, the popularity score is weighed at 10%, because people who have a tendency to open products likely wouldn't be majorly swayed by popularity scores, since the goal of opening the product is commonly to ensure no unusual issues with the product, like factory defects, which can occur regardless of popularity. However, the popularity score still holds minor sway, since people are more likely to want and look at those products. The equation is shown below:

$$OPEN\ PACKAGING\ SCORE = w_{1\,OP} * R + w_{2\,OP} * P + w_{3\,OP} * V_{cashier} \quad \textit{(Equation 7)}$$

The rough handling score scores the likelihood that products in a given department will be rough handled by customers, as a score from 0-1. The rough handling score was calculated taking into various factors into account, which included the population density, obstruction of the department for the cashier, and the popularity score for the department. A higher population density would warrant a higher likelihood of rough handling, as there are more people who may cause accidents or careless handling of merchandise to take place. Moreover, products are more likely to be roughly handled if the cashier cannot see the customer. Thus, the rough handling score is a weighted average of the obstruction factor and the people to product ratio. The rough handling score is a number from 0-1. In this equation, the cashier obstruction is weighed at 30% because, like with the open packaging score, rough handling is less likely to occur in the presence of a cashier. Thus, the people to product ratio is weighed at 70%, because a more condensed population means it will be harder to access products. The The equation is shown below:

$$ROUGH\ HANDLING\ SCORE = w_{1\,RH} * R + w_{2\,RH} * V_{cashier} \quad \textit{(Equation 8)}$$

The total score was calculated as a weighted average of the dropping score (*DS*), the open packaging score (*PS*), and the rough handling score (*RHS*). The dropping and rough handling scores are both weighed at 42.5%, because both are more likely to occur at a flash sale. Open packaging, however, is weighed at 15%, because it is less common than the other two, and because the rush of a flash sale means that there is less time to do this. The equation is shown below:

$$TOTAL\ SCORE = w_{1\,T} * DS + w_{2\,T} * PS + w_{3\,T} * RHS \quad \textit{(Equation 9)}$$

The total score was used to calculate the number of products damaged. Because the scores have relative meaning (the scores were a number from 0-3 that could be used to compare to each other, and not the number of

products damaged), the total score was divided by 3 to obtain a percentage. However, this percentage was the expected damage of a department when compared to another department. In order to convert this number to a physical number of objects damaged, a simple proportion can be used. Every year, the number of damages from unknown reasons, which includes accidental damages, is from 7-10% (2). This means that the maximum percentage damage from careless or accidental damages is 10%. Because the maximum score is 3, you can use a conversion factor of 10% → 3 to calculate the percent damages with a given total score. The equation modeling this is shown below in Equation 10.1, and the formula to calculate the percent damages as derived from Equation 10.1 is shown in 10.2:

$$\frac{TOTAL\ SCORE}{3} = \frac{P_{damages}}{0.1} \quad \textit{(Equation 10.1)}$$

$$\frac{10 * TOTAL\ SCORE}{3} = P_{damages} \quad \textit{(Equation 10.2)}$$

The percent damages were used to calculate the total number of damages by multiplying the total number of people in the department by the percent damage. This was then iterated over all of the departments in the layout and summed to obtain the total number of products damaged. The number of products damaged per department is shown in Equation 11.1, and the total number of products damaged overall is shown in Equation 11.2.

$$PD_{department} = P_{damages} * n_{avg} \quad \textit{(Equation 11.1)}$$

$$TOTAL\ PRODUCTS\ DAMAGED = PD_{department\ 1} + PD_{department\ 2} + \cdots + PD_{department\ n} \quad \textit{(Equation 11.2)}$$

Analyzing Layouts: Java Simulation

The math model was then incorporated into a Java-based simulation of the store. The Java file system is organized into four main classes: the mainClass.java class, the person.java class, the map.java class, and the item.java class. These classes all work together to simulate the sale and damaging of items in the store performed by customers.

The person.java class controls how the simulated people will move, through the randomAction() method. randomAction() is split into two separate parts. In the first part, the direction that the person is facing is changed based on the value given by Math.random(). In the second part, whether or not the person will move is based on another value given by Math.random(). However, if the person would run into an item other than the specified path item, the person doesn't move, and waits at their current position until they can move to another path item. Additionally, if the person would run into an item representative of a department of items for sale, they stay in their current position, but the person has the ability to buy or damage the items in front of them. Whether or not they buy something is based on the overall department's average popularity score, as calculated earlier in section 1B. Whether or not the person damages an item through rough handling, opening the packaging, or dropping it depends on the corresponding scores as calculated in section 2A. Additionally, if a person reaches a cash register item, that person is removed from the simulation, as it is assumed that they paid for the items they bought and left.

The map.java class is mainly composed of a 2D array of "item" objects that are continually adjusted when "person" objects interact with them through the randomAction() method. Each department, such as

Appliances or Video Gaming, has its own item that contains its average popularity score and the number of overall products. Since each part of the department is a single item, when any part of the department decreases in the overall number of items, the availability of items for all of the other parts of the department also decreases. map.java also keeps a running count of how many individual items have been bought or damaged in various ways (rough handled, opened, or dropped).

The item.java class contains the int numberOfItems, a total number of smaller items that it keeps track of; the string itemType, which serves as an identifier for the item in various other classes; and the double popularity, a constant that holds the overall popularity of the item. For path items, numberOfItems is representative of the total number of people on that item at a time, itemType simply identifies the item as a path, and the popularity score is not utilized. For the items representative of different departments, numberOfItems is the total number of items on the shelves of that department, and itemType and popularity are both used as mentioned.

The mainClass.java class is the main class of the simulation, and incorporates the elements of the other three classes. mainClass.java starts off by making a new map, currentMap, and then printing the initial state of this map out. Then, mainClass.java iterates through a for loop 200 times, with each time counting for approximately one minute in real life. Each time the for loop runs, all of the existing people perform a random action, based on the randomAction() method in the person.java class. Additionally, a random number of up to five people “enter” the simulation, at the coordinates of the door, with the total number of people in the simulation never exceeding 500.

An example of what the map of the store looks like is given below.

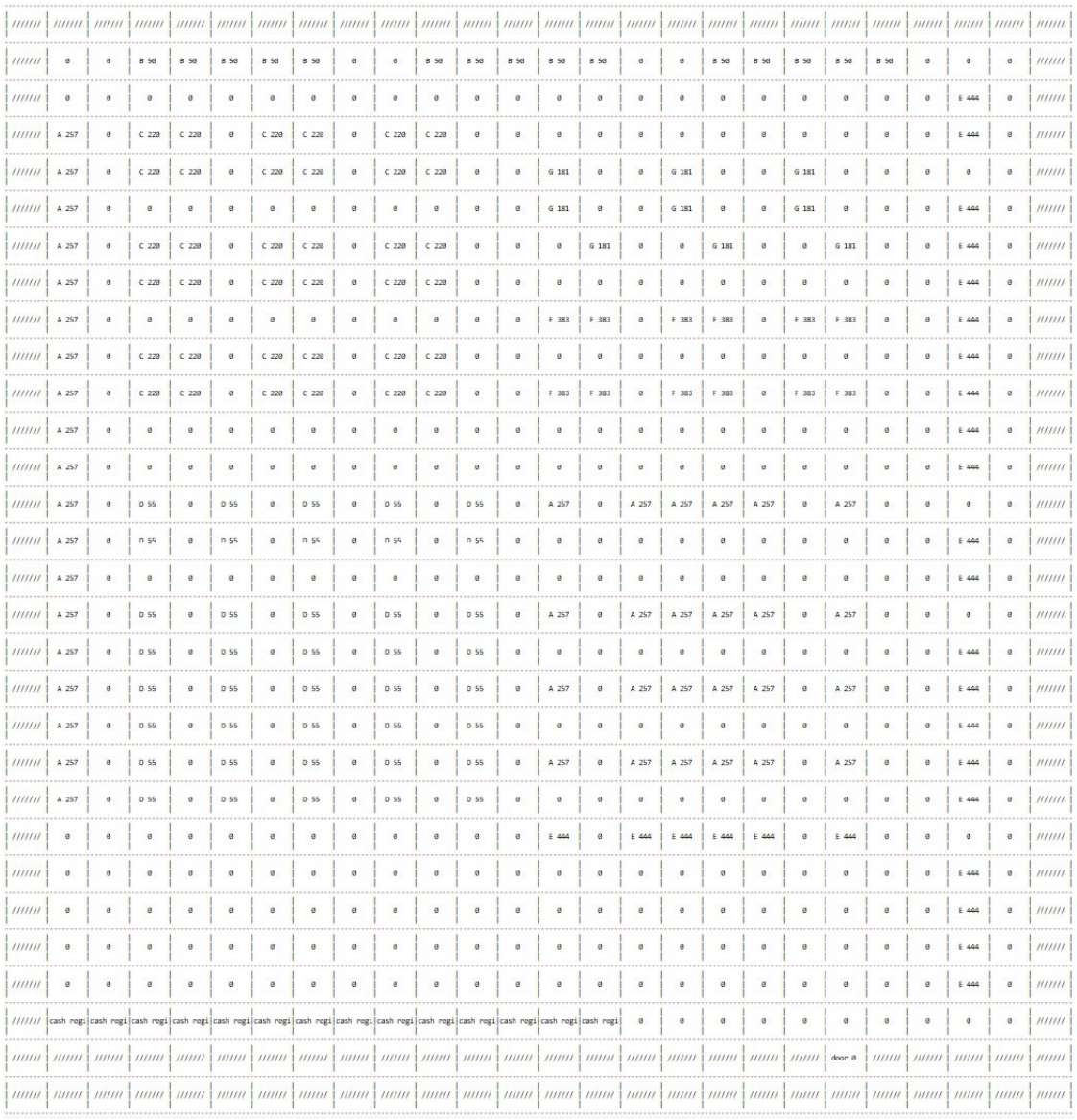

Figure 2: An example map produced by mainClass.java.

This map is modified through each iteration of the loop, representing the movement of people, the sale of products, and the damaging of products.

Based on your factors and model, discuss the optimal locations of the store's departments and the most popular/desired sale items. Indicate these locations on the floor plan. In other words, label the various areas of the floor plan with your team's choice for the locations of departments and displays of the most popular sale products.

Based on the given model, a sectioned map of each department was created using the original floor plan, including total department area and the area in which products are displayed, which is shown below:

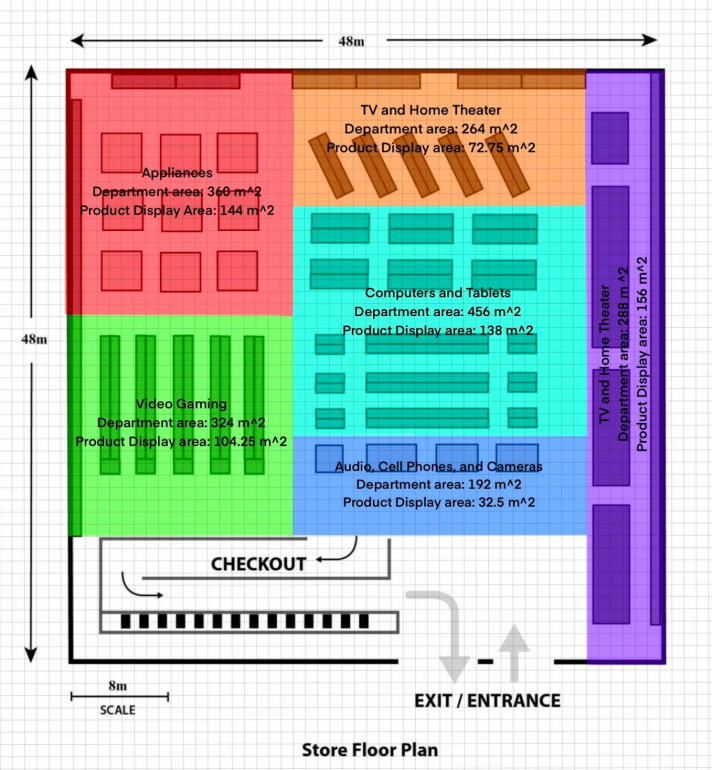

Figure 3: A color-coded version of the original floor plan. Each color represents a certain section of the store corresponding to the department(s) included in that section, except for TV and Home Theater. This exception is explained below.

Closest to the cashier stations are the Video Gaming, Audio, Cell Phones, and Cameras departments. The Audio department had the highest overall popularity score, and the size of its products (speakers and headphones are typically smaller than many of the products in other departments) means that that section can be shared with cell phones and cameras, which are also generally smaller products. This is the smallest section, and one of the closest to the cashier station, and so one of the strengths of this placement is that it is highly visible to the cashier due to its smallness and lack of obstruction from cashier view. The condensed nature of all three departments means that the product density is high, making it less likely that a product will be dropped or knocked over, since a higher product density means a smaller people per product ratio. Meanwhile, Video Gaming is a larger section and closer to the cashier because it is a large department with the second-highest popularity score. This means that the department is easily visible for cashiers, but that product density is lower, meaning that the people per product ratio is higher and therefore dropping is more likely.

In the middle of the store is the Computers and Tablets department. This department has the most products, and so it was necessary for it to have a large space. Generally, the popularity score of products in this department was medium-to-high, which means that it needs to be visible to a cashier to prevent rough handling despite the availability of product because it will sell fast. The large space lessens product density, meaning that there is more likeliness of dropping or knocking over product and rough handling as people may scramble to access products, and it is partially obstructed by the section in front of it, so there is a higher percentage of

package opening. However, placing products with higher popularity scores in sight of the cashiers would lessen the likeliness of package opening, since those are more likely to be opened and checked by customers.

In the back left corner of the store is the Appliances department. This department includes products like washers and dryers, which are typically large, and so it requires a large space despite its low number of products. Overall, appliances had generally low popularity scores, and so cashier visibility was less necessary. Putting this department in such a large space allows all the products to fit with less product density, and so it is more likely that products will be roughly handled, dropped, or knocked over despite product size. However, lower popularity scores also means that these likelihoods are slightly leveled out.

In the back right corner, split into two sections, is the TV and Home Theater department. These, like appliances, are typically large products, and so they required a significant amount of space. However, the popularity score of products in this department is also generally low-to-medium, meaning that it isn't imperative for the products to be visible to the cashier. Thus, it is possible to put lower-scored products in the back (orange) section and higher-scored products in the (purple) section closer and less obstructed to the cashier stations, minimizing the likelihood of package opening and rough handling. The large area of the department, however, means that the people to product ratio would be lower, and so rough handling and dropping are more likely.

After having run the Java simulation ten times, with a maximum of 500 people, 200 iterations, and with the map based on *Figure 3*, the averages for the calculated values are shown in the table below. See *Appendix 1* for the full program.

	Items bought	Items damaged	Remaining Items	Rough handled	Opened	Dropped
Run 1	397	14	1179	8	1	5
Run 2	419	14	1157	3	3	8
Run 3	453	23	1114	14	1	8
Run 4	433	19	1138	5	3	11
Run 5	450	24	1116	11	2	11
Run 6	439	15	1136	3	1	11
Run 7	419	15	1156	5	1	9
Run 8	410	13	1167	5	0	8
Run 9	344	17	1229	5	0	12
Run 10	474	22	1094	10	1	11
Average	**423.8**	**17.6**	**1148.6**	**6.9**	**1.3**	**9.4**

Table 3: A table of the results after ten runs of the simulation. Based on this model, the average number of damaged items is between 17 and 18.

Using your analysis and the model you developed, create and evaluate a new and better floor plan for this flash sale scenario. The store dimensions, scale, location of the entrance/exit, and the items for sale remain the same, but your team can now create its own layout. Justify why your floor plan is better than the one given the problem.

Optimizing Layouts: Pygame Simulation

Using the model provided, a Python simulator was developed that allows for the movement of individual departments and the cashier stations and calculates the consequent number of damaged products throughout the store. This simulator creates a platform for objects, and the user can move the objects around the floor using a drag and drop mechanism. Cashiers and departments can be added and taken out of the floor. Based on the locations and properties of the departments and cashiers, the number of products damaged increases and decreases.

The program models the display area, department area, dimensions, position, population percentage, and total quantity of items for each department. Each department is also color-coded using the same colors as the sectioned map in *Figure 3*. It is important to note that unlike the original layout map given, the program models each department display area using one box and not individual boxes. The program uses the model to determine the likelihood of products being damaged through the damageability score, then uses that to determine a percentage of products damaged and calculates the total damaged products. This program is split into five sections: utils.py, information.py, department.py, calculate_score.py, and main.py.

The first of these classes, utils.py, calculates the people per product ratio (referred to in the code as the people per product metric), population density, product density, and obstruction. This class also calculates the population density and product density, both of which are necessary to calculate the people per product ratio. It includes the Point class as well, which is used to calculate the distance between departments and cashiers. The information.py class includes all constants necessary for further calculations. The user is allowed to move each department, through department.py, which initializes the screen representative of the store where 10 pixels is 1 meter. Each time a department is moved in department.py, calculate_scores.py calculates the total score (represented by a number between 0 and 3) and uses it to find the percentage and, therefore, the total number of damages in the store during the entire sale.

Using this program, the team found that the best floor plan, in its simplest form, is shown below:

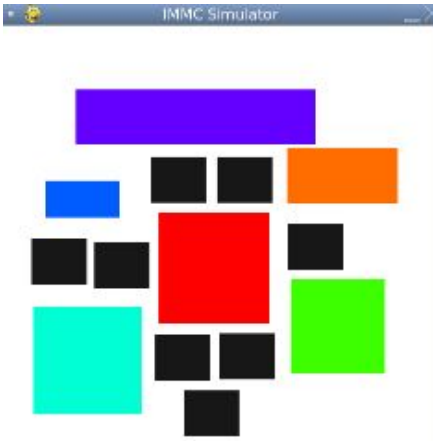

Figure 4: A simple form of the new floor plan, color coded by department, including each department as its product display area.

Based on this layout, the total number of items damaged was approximately 17.13 (about 17 items). The simulation of the original floor plan provided by the manager resulted in approximately 19 damaged items. A table of the damages for the given layout versus the damages for the developed layout is shown below:

Layouts: Damage Scores		
	Num People	Damages
Given Layout	500	20.5
Developed Layout	500	17.1

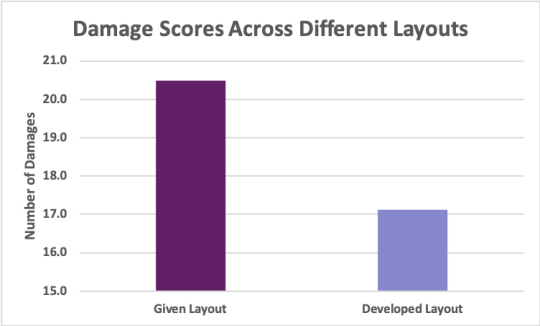

Table 4: A table of the number of people at the flash sale (set to the maximum capacity of an arbitrary retail store) versus the number of damages in the layouts. Figure 5: A pictorial representation of this data. The given layout is plotted versus the number of damaged items for each layout.

See *Appendix 2* for the full program.

Putting this through the Java simulation results in the following:

	Items bought	Items damaged	Remaining Items	Rough handled	Opened	Dropped
Run 1	178	36	1376	11	2	23
Run 2	177	27	1386	14	0	13
Run 3	183	33	1374	13	2	18
Run 4	188	27	1375	9	6	12

Run 5	182	27	1381	9	1	17
Run 6	194	26	1370	11	2	13
Run 7	185	28	1377	14	3	11
Run 8	189	21	1380	10	1	10
Run 9	175	29	1386	11	2	16
Run 10	179	24	1387	10	2	12
Average	**183.0**	**27.8**	**1379.2**	**11.2**	**2.1**	**14.5**

Table 4: A table of the results after ten runs of the simulation. Based on this model, the average number of damaged items is between 27 and 28.

As shown above, the amount of items bought in this simulation and the number of items damaged is higher than the original floor plan. This could potentially be due to the nature of the simulation: as the person objects move through the simulation, they are removed when they come across a cash register item. Additionally, because the person objects act randomly (that is, they do not favor a particular route to another) and thus would neither try to avoid the cash registers, nor actively seek out the items for sale, the bottom right corner of the simulation is heavily favored, and the probabilities calculated there hold the most impact in the overall simulation, as the simulation is only over the course of 200 loops, and does not recognize the full sawy of popularity scores on a person's path.

Conclusion

Given the model, which scores an electronics and appliances store's floor plan upon its ability to prevent the dropping, rough handling, and opening of items, a floor plan was determined (Figure x) that minimized damages to products during a flash sale. It was determined that:

- Less popular items should be placed further from cashier view, while more popular items should be included in areas less obstructed for cashiers.
- Larger items or items larger in quantity should be placed in the largest sections of the store, while smaller items or items smaller in quantity should be placed in smaller sections of the store.
- Cashier stations are most effective in layouts where cashiers can see the most possible products, and so multiple cashier stations is more effective than one large cashier station.

This meant that, despite the center of the new floor plan being surrounded by cashier stations, it is best for a low-medium popularity department to be placed there because other departments either wouldn't fit or would leave too much space to reasonably justify not placing an additional department in the center, which would case significant obstruction of view for some of the surrounding cashiers. This way, there is also space on the outside of the area surrounded by cashiers where the only obstructions come from other cashiers, in which case visibility is high and more popular departments are more available.

The benefits of this model are that it is extremely flexible. The programs and equations used are designed to take into account different numbers of products, different layouts, and different numbers of customers, meaning that this layout could be applied to any brick-and-mortar retail store. The model is also able to predict the number of products damaged, and in what manner, and so additional precautions can be implemented by the store alongside the floor plan changes.

The downsides to this model, however, are mainly that specific products are not accounted for. Since all products are addressed in groups, individual models of different popularities or sizes are not accounted for, meaning that the foot traffic in a certain department could vary from what the model presents or the area provided for a department could be unnecessarily large or too small. It also does not take into account the total spending of a customer, which could sway how that customer spends their money throughout the store: if they have already bought a $1500 product, they are less likely to buy another of the same price, but this is not accounted for. The calculations between simulations are not equivalent, either, which creates discrepancies between each simulation as one may determine that a floor plan is more effective, but another may not.

If given additional time, the team would have implemented methods that would modify damages based on prior spending, product size, and the specific popularity of each product. This would have created a more precise model. Including employees not stationed at cash registers would also strengthen the model, since while not as many employees are likely to be roaming the store, there would probably still be some walking around to reorganize items or even to further prevent damages. The addition of cameras and product displays that do not allow customers to take the item, as is often seen with laptops or cell phones, would also lessen the likelihood of damage, since there would be more view of the store for employees and less direct access to purchasable items for reckless customers. It also could have been effective to divide more departments, as was done with TV and Home Theater, as it could possibly lessen population density and, therefore, damageability scores. Accounting for multiple cashier stations in the Java simulation would also create more accurate calculations.

Letter

Write a one-page letter to the store manager supporting your floor plan layout and discussing any additional strategies for a successful flash sale.

Good day,

Provided is the ideal floor plan layout to prevent damages to on-sale products during your flash sale. After having run several simulations—both on the effectiveness of the floorplan and on the predicted movement of people—to check the number of damages that would occur for the layout based on the popularity of the products, we have determined that this design allows for the least damages. This is because our layout separates the cashier stations into multiple sections, which creates more overall visibility of the store for employees. The spacing of these cashiers allows for each department to be supervised as unobstructed as possible, which will deter customers from reckless behavior and also spaces departments in such a way that the dropping of products is less likely.

We would also like to suggest a few other points of interest to further ensure that your sale is both safe and a success. For one, it may be a worthwhile endeavor to install cameras throughout the store, which would allow your employees to keep an eye on products that would otherwise be out of sight for them. This way, they could prevent customers from opening or otherwise damaging products when they believe nobody is looking.

Another suggestion is to schedule some of your employees to patrol the store floor for the same reason. People are less likely to break rules when they think they'll be caught, after all, and the occasional employee walking around could deter them from knocking products around or acting in similar reckless manners.

Finally, it would likely be best to store some of your real products in a safe, employee-restricted area and instead provide singular product displays for products more susceptible to damage. This way, customers can decide whether they'd like to buy a product, but do not run the risk of damaging that same product in an attempt to retrieve it, because they will have to ask for the assistance of an employee in order to access whatever item they're looking for. This would be especially effective for more popular items, like gaming products or audio devices.

We hope you find your flash sale— and our floor plan— successful and satisfying.

Sincerely,
Your Renovation Team

References

[1] - WHAT CAUSES SHRINKAGE IN RETAIL (AND HOW TO FIGHT IT) [Internet].; 2018 [updated JUN 25,;]. Available from: https://www.shopkeep.com/blog/whaat-causes-retail-shrinkage#step-1.

[2] - Reduce Inventory Shrinkage [Internet].; 2019 [updated January 16,;]. Available from: https://blog.camerasecuritynow.com/2019/01/16/reduce-inventory-shrinkage/.

[3] - >4 Ways to Prevent Inventory Shrinkage [Internet]. []. Available from: https://smartwerksusa.com/articles/4-ways-to-prevent-inventory-shrinkage-in-retail/.

[4] - PyGame: Drag object on screen using mouse [Internet].; 2020 [updated February 24,;]. Available from: https://blog.furas.pl/python-pygame-drag-object-on-screen-using-mouse-gb.html.

[5] - How Long Does a Gaming PC Last [Internet].; 2019 [updated August 25,;]. Available from: https://gametechia.com/how-long-does-a-gaming-pc-last/.
(https://www.cnbc.com/2019/05/17/smartphone-users-are-waiting-longer-before-upgrading-heres-why.html)

[6] - How long can current gen last people? [Internet].; 2019 []. Available from: https://www.gamespot.com/forums/system-wars-314159282/how-long-can-current-gen-last-people-33437621/.

[7] - What Is the Life Span of a Computer Monitor? [Internet]. []. Available from: https://www.techwalla.com/articles/life-span-computer-monitor_.

[8] - How Long Should A Laptop Last? Laptop Lifespan & Average Battery Life [Internet].; 2018 [updated June 29,;]. Available from: https://techguided.com/how-long-should-a-laptop-last/.

[9] - The Average Lifespan of 7 Popular Tech Products [Internet].; 2015 [updated December 29,;]. Available from: https://www.business2community.com/tech-gadgets/average-lifespan-7-popular-tech-products-01413366.

[10] - HOW LONG SHOULD YOUR TV LAST? [Internet].; 2019 [updated February 13,;]. Available from: https://www.reviewed.com/televisions/features/how-long-should-a-tv-last.

[11] - What is the life span of a printer? [Internet].; 2014 [updated July 2,;]. Available from: https://www.answers.com/Q/What_is_the_life_span_of_a_printer.

[12] - How Long Do Laptops Last And How To Make Them Last Longer? [Internet].; 2019 [updated July 29,;]. Available from: https://durabilitymatters.com/how-long-do-laptops-last/.

[13] - Americans are now holding onto their iPhones for almost three years [Internet].; 2018 [updated October 30;]. Available from: Your Wireless Earbuds Are Trash (Eventually).

[14] - Your Wireless Earbuds Are Trash (Eventually) [Internet].; 2020 [updated February 26,;]. Available from: https://thewirecutter.com/blog/your-wireless-earbuds-are-trash-eventually/.

[15] - Here’s Why Your Headphones Keep Breaking (And What You Can Do) [Internet].; 2019 [updated November 28,;]. Available from: https://www.makeuseof.com/tag/this-is-why-your-headphones-keep-breaking/.

[16] - How Long Do Vacuum Cleaners Last? [Internet].; 2016 [updated Dec 31,;]. Available from: https://www.consumerreports.org/vacuum-cleaners/how-long-do-vacuum-cleaners-last/.

[17] - Vacuum Cleaner Lifespan - How Often to Replace [Internet]. []. Available from: https://themvacuums.com/vacuum-cleaner-lifespan-often-replace/.

[18] - How Often Should You Upgrade Your Primary Camera Body? [Internet].; 2015 []. Available from: https://www.slrlounge.com/often-upgrade-primary-camera-body/.

[19] - Abigail N. Smartphone users are waiting longer before upgrading – here's why. 2019 May 16,.

[20] - Canon EOS Digital Cameras Shutter Life Expectancy [Internet]. []. Available from: https://blog.usro.net/canon-eos-digital-cameras-shutter-life-expectancy/.

[21] - When Should You Replace Your Major Home Appliances? [Internet].; 2017 [updated APRIL 29,;]. Available from: https://www.howtogeek.com/304831/when-should-you-replace-your-major-home-appliances/.

[22] - This Is How Long These 6 Appliances Should Last [Internet].; 2015 [updated 14 October;]. Available from: https://www.wisebread.com/this-is-how-long-these-6-appliances-should-last.

[23] - What is the maximum occupancy of a typical Lowe’s store? [Internet].; 2020 [updated April 25,;]. Available from: https://www.reddit.com/r/Lowes/comments/fv2184/what_is_the_maximum_occupancy_of_a_typical_lowes/.

Appendix 1: Java Simulation

Note:

- **The full code and implementation of the java algorithm is in the github repository at: https://github.com/IMMC9539/IMMC_2020_java_sim**
- **A demo for the java simulation is at the link: https://drive.google.com/file/d/1tGG4QRfX1rXJSfN450bM63KSP9fweuYd/view?usp=sharing**

- **The full code and implementation of the python algorithm is in the github repository at: https://github.com/IMMC9539/IMMC_2020_python_sim**
- **A demo for the python simulation is at the link: https://drive.google.com/file/d/1GqcldZySt0D1Jpzh5RfqV7KnKvUW8_Uh/view?usp=sharing**

mainClass.java:

```
package immc;

import java.util.ArrayList;
import java.util.Arrays;

public class mainClass
{
	public static void main(String[] args)
	{
		ArrayList<person> people = new ArrayList<person>();
		map currentMap = new map();

		int maxPeople=500;

		currentMap.printMap();
		//each counter is approximately 1-2 minutes
		for (int i = 0; i<200; i++)
		{
//			System.out.println("");
//			System.out.println("Start of iteration "+i);
			for (person p:people)
			{
				//remove people from previous step
```

```
                        currentMap.removeItem(p.getRow(), p.getCol());
                        //perform action
                        p.randomAction(currentMap);
                        //put person back in
                        currentMap.addItem(p.getRow(), p.getCol());
                }

                for (int j = 0; j<people.size(); j++)
                {
                        //delete person if finished
                        if (currentMap.onCashRegistar(people.get(j).getRow(), people.get(j).getCol()))
                        {
                                people.remove(j);
                        }
                }
                maxPeople=newPeople(people, currentMap, maxPeople);
//                      System.out.println("");
//                      System.out.println("After iteration "+i);
//                      currentMap.printMap();
        }
        System.out.println("");
        System.out.println("Last iteration");
        currentMap.printMap();

        System.out.println("");
        System.out.println("Final Count");
        System.out.println("Initial items: "+currentMap.getNumInitial());
        System.out.println("Items bought: "+currentMap.getNumBought());
        System.out.println("Total items damaged:
"+(currentMap.getNumRoughHandled()+currentMap.getNumOpened()+currentMap.getNumDropped()));
        System.out.println("Remaining items:
"+(currentMap.getNumInitial()-currentMap.getNumBought()-(currentMap.getNumRoughHandled()+currentMa
p.getNumOpened()+currentMap.getNumDropped())));
        System.out.println("----------------------------");
        System.out.println("Items rough handled: "+currentMap.getNumRoughHandled());
        System.out.println("Items opened: "+currentMap.getNumOpened());
        System.out.println("Items dropped: "+currentMap.getNumDropped());
```

```
	}

	private static int newPeople(ArrayList<person> people, map currentMap, int maxPeople)
	{
		int number = (int)(Math.random()*5);

		if (maxPeople<5)
		{
			number = (int)(Math.random()*maxPeople);
		}

		for (int i = 0; i<number; i++)
		{
			people.add(new person(currentMap.getDoorY(), currentMap.getDoorX()));
			currentMap.addItem(currentMap.getDoorY(), currentMap.getDoorX());
		}

		return maxPeople-number;
	}

}
```

person.java:

```
package immc;

public class person
{
	private int r;
	private int c;
	//north, east, south, west. direction person is facing.
	private String direction;

	public person(int initialRow, int initialCol)
	{
		r=initialRow;
		c=initialCol;
		direction="north";
	}
```

```
	private void onItem(map currentMap, int newR, int newC)
	{
		double probability = (Math.random());
		double popularityScore = currentMap.getPopularityScore(newR, newC)/100;

		probabilities p = new probabilities(currentMap, newR, newC, r, c);

		double roughHandlingScore = 0.1*p.calculateRoughHandling();
		double openingPackagingScore = 0.1*p.calculateOpening();
		double droppingScore = 0.1*p.calculateDropping();

		//buy the item
		if (probability<popularityScore)
		{
			currentMap.removeItem(newR, newC, "bought");
		}
		//damage the item by rough handling
		else if (popularityScore<=probability&&probability<popularityScore+roughHandlingScore)
		{
			currentMap.removeItem(newR, newC, "roughHandled");
		}
		//damage the item by opening packaging
		else if
(popularityScore+roughHandlingScore<=probability&&probability<popularityScore+roughHandlingScore+ope
ningPackagingScore)
		{
			currentMap.removeItem(newR, newC, "opened");
		}
		//damage the item by dropping
		else if
(popularityScore+roughHandlingScore+openingPackagingScore<=probability&&probability<popularityScore+
roughHandlingScore+openingPackagingScore+droppingScore)
		{
			currentMap.removeItem(newR, newC, "dropped");
		}
		//ignore the item and move on
		else
		{
			//nothing
```

```
		}
	}

	public void moveForward(map currentMap)
	{
		int newR=r;
		int newC=c;
		if (direction.equals("north"))
		{
			newR--;
		}

		if (currentMap.pathAhead(newR, newC)||currentMap.onCashRegistar(newR, newC))
		{
			r=newR;
			c=newC;
		}
	}

	public void randomAction(map currentMap)
	{
		//right now the percentages are very arbitrary, this is simply to illustrate a possible concept for the paths of the people.
		//also, the reason why there are 4 possibilities is because someone is more likely to go forward.

		//rotate
		double probability = (int)(Math.random()*4);
		//stay
		if (probability==0||probability==1)
		{
			//no change in direction
		}
		//left
		else if (probability==2)
		{
			if (direction.equals("north"))
			{
				direction="west";
			}
```

```
		else if (direction.equals("east"))
		{
			direction="north";
		}
		else if (direction.equals("south"))
		{
			direction="east";
		}
		else
		{
			direction="south";
		}
	}
	//right
	else if (probability==3)
	{
		if (direction.equals("south"))
		{
			direction="west";
		}
		else if (direction.equals("west"))
		{
			direction="north";
		}
		else if (direction.equals("north"))
		{
			direction="east";
		}
		else
		{
			direction="south";
		}
	}


	//move
	probability = (int)(Math.random()*2);
	//forward
	if (probability==0)
```

```
{
	//determine what the new coordinates of the person would be
	int newR=r;
	int newC=c;
	if (direction.equals("north"))
	{
		newR--;
	}
	else if (direction.equals("east"))
	{
		newC++;
	}
	else if (direction.equals("south"))
	{
		newR++;
	}
	else
	{
		newC--;
	}

	//check whether or not you can actually move
	if (currentMap.pathAhead(newR, newC)||currentMap.onCashRegistar(newR, newC))
	{
		r=newR;
		c=newC;
	}
	else if (currentMap.isWall(newR, newC))
	{
		//do nothing; stay where you are
	}
	//would go to an item cell
	else
	{
		if (currentMap.getItemNum(newR, newC)>0)
		{
			onItem(currentMap, newR, newC);
		}
	}
```

```
                }
                //stay
                else
                {
                        //nothing
                }
        }

        public int getRow()
        {
                return r;
        }

        public int getCol()
        {
                return c;
        }
}
```

map.java:

```
package immc;
public class map
{
        private item[][] map;
        private int bought;
        private int roughHandling;
        private int openedPackaging;
        private int dropping;
        private int initialItems;
        private int doorX;
        private int doorY;
        private double cshRegX[];
        private double cshRegY[];

        public map()
        {
                map=newMap();
                bought=0;
                roughHandling=0;
```

```
            openedPackaging=0;
            dropping=0;
        }

        private item[][] newMap()
        {
            //set number of items to what you need. only in this class, because it might change based on the map.
            initialItems=257+383+444+181+50+220+55;

            item cshReg = new item("cash register");
            item door = new item("door");

            //Appliances
            item groupA = new item("A", 257, 10.87, 144);
            //TV and Home Theater
            item groupB = new item("B", 383, 11.13, 73+156);
            //Computers and Tablets
            item groupC = new item("C", 444, 15.32, 138);
            //Video Gaming
            item groupD = new item("D", 181, 15.97, 104);
            //Audio, Cell Phones, and Cameras
            item groupE = new item("E", 50+220+55, 18.29+13.04+15.38, 32);

            //26 by 26 array. perimeter is walls, so actually 24 by 24 array of items, with each item measuring 2m by 2m. Based on the given store layout.
//          item[][] array = new item[][] {
//              {wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall()},
//              {wall(), path(), path(), groupA, groupA, groupA, groupA, groupA, path(), path(), groupB, groupB, groupB, groupB, groupB, path(), path(), groupB, groupB, groupB, groupB, groupB, path(), path(), path(), wall()},
//              {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//              {wall(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//              {wall(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), path(), groupB, path(), path(), groupB, path(), path(), groupB, path(), path(), path(), path(), path(), wall()},
```

```
//                {wall(), groupA, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), groupB, path(), path(), groupB, path(), path(), groupB, path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA,
path(), path(), path(), groupB, path(), path(), groupB, path(), path(), groupB, path(), path(), groupB, path(),
wall()},
//                {wall(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA,
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupA, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), groupC, groupC, path(), groupC, groupC, path(), groupC, groupC, path(), path(), groupB, path(), wall()},
//                {wall(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA,
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA,
path(), path(), groupC, groupC, path(), groupC, groupC, path(), groupC, groupC, path(), path(), groupB, path(),
wall()},
//                {wall(), groupA, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupD, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupD, path(), groupD, path(), groupD, path(), groupD, path(), groupD, path(),
groupD, path(), groupC, path(), groupC, groupC, groupC, groupC, path(), groupC, path(), path(), path(), path(),
wall()},
//                {wall(), groupD, path(), groupD, path(), groupD, path(), groupD, path(), groupD, path(),
groupD, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupD, path(), groupD, path(), groupD, path(), groupD, path(), groupD, path(),
groupD, path(), groupC, path(), groupC, groupC, groupC, groupC, path(), groupC, path(), path(), groupB,
path(), wall()},
//                {wall(), groupD, path(), groupD, path(), groupD, path(), groupD, path(), groupD, path(),
groupD, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupD, path(), groupD, path(), groupD, path(), groupD, path(), groupD, path(),
groupD, path(), groupC, path(), groupC, groupC, groupC, groupC, path(), groupC, path(), path(), groupB,
path(), wall()},
//                {wall(), groupD, path(), groupD, path(), groupD, path(), groupD, path(), groupD, path(),
groupD, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), groupD, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), groupE, path(), groupE, groupE, groupE, groupE, path(), groupE, path(), path(), path(), path(), wall()},
//                {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
```

```
//                  {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                  {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupB, path(), wall()},
//                  {wall(), path(), cshReg, cshReg, cshReg, cshReg, cshReg, cshReg, cshReg, cshReg, cshReg,
cshReg, cshReg, cshReg, cshReg, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
wall()},
//                  {wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(),
wall(), wall(), wall(), wall(), wall(), wall(), wall(),  door , wall(), wall(), wall(), wall(), wall()},
//                  {wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(),
wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall()},
//                  };

                    //our new and improved layout
                    item[][] array = new item[][] {
                        {wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(),
wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall()},
                        {wall(), path(), groupB, groupB, path(), groupB, groupB, groupB, groupB, groupB, groupB,
groupB, groupB, path(), groupB, groupB, groupB, groupB, groupB, path(), groupB, groupB, groupB, groupB,
path(), wall()},
                        {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), wall()},
                        {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), groupB, groupB, groupB, groupB, groupB, path(), groupB, groupB, groupB, groupB, groupB, wall()},
                        {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), wall()},
                        {wall(), path(), groupE, path(), groupE, groupE, groupE, groupE, path(), groupE, path(),
cshReg, cshReg, path(), cshReg, cshReg, path(), path(), path(), path(), path(), path(), path(), groupB, path(),
wall()},
                        {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), groupB, path(), path(), groupB, path(), path(), groupB, path(), wall()},
                        {wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(),
path(), path(), path(), path(), groupB, path(), path(), groupB, path(), path(), groupB, path(), wall()},
                        {wall(), path(), path(), path(), path(), path(), path(), path(), groupA, path(), path(), groupA,
groupA, groupA, groupA, groupA, path(), path(), groupB, path(), path(), groupB, path(), path(), groupB,
wall()},
                        {wall(), path(), cshReg, cshReg, path(), cshReg, cshReg, path(), path(), groupA, path(), path(),
path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), wall()},
```

{wall(), path(), path(), path(), path(), path(), path(), path(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), cshReg, cshReg, path(), path(), path(), path(), wall()},
{wall(), groupC, path(), groupC, groupC, path(), groupC, path(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), path(), path(), path(), path(), path(), path(), wall()},
{wall(), path(), path(), path(), path(), path(), path(), path(), groupA, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), groupC, path(), groupC, groupC, path(), groupC, path(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), path(), path(), path(), path(), path(), path(), path(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), groupC, path(), groupC, groupC, path(), groupC, path(), groupA, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), path(), path(), path(), path(), path(), path(), path(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), groupC, path(), groupC, groupC, path(), groupC, path(), groupA, path(), groupA, groupA, path(), groupA, groupA, path(), groupA, groupA, path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), path(), path(), path(), path(), path(), path(), path(), groupA, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), groupC, path(), groupC, groupC, path(), groupC, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), cshReg, cshReg, path(), cshReg, cshReg, path(), path(), path(), path(), groupD, path(), groupD, path(), groupD, wall()},
{wall(), groupC, path(), groupC, groupC, path(), groupC, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), wall()},
{wall(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), cshReg, cshReg, cshReg, path(), path(), path(), path(), path(), path(), path(), path(), path(), path(), wall()},
{wall(), path(), wall()},
{wall(), path(), wall()},
{wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), wall(), door , wall(), wall(), wall(), wall(), wall()},
{wall(), wall()},

```
        };

        doorX=20;
        doorY=25;

        cshRegX=new double[] {2.5, 5.5, 10.5, 13.5, 10.5, 12.0, 13.5, 19.0};
        cshRegY=new double[] {9.0, 9.0, 5.0, 5.0, 20.0, 21.0, 20.0, 10.0};

        return array;
}

private item wall()
{
        return new item("wall");
}

private item path()
{
        return new item("");
}

public void addItem(int r, int c)
{
        map[r][c].addItem();
}

public void removeItem(int r, int c, String string)
{
        //this part of the code is irrelevant for now, but it's here if we need it.
        if (map[r][c].currentItems()!=0)
        {
                if (string.equals("roughHandled"))
                {
                        roughHandling++;
                }
                else if (string.equals("opened"))
                {
                        openedPackaging++;
                }
```

```
                else if (string.equals("dropped"))
                {
                        dropping++;
                }
                else
                {
                        bought++;
                }
        }

        //change number of items available. doesn't decrease if number of items is 0.
        map[r][c].removeItem();
}

public void removeItem(int r, int c)
{
        map[r][c].removeItem();
}

public int getItemNum(int r, int c)
{
        return map[r][c].currentItems();
}

public item getItem(int r, int c)
{
        return map[r][c];
}

public int getDoorX()
{
        return doorX;
}

public int getDoorY()
{
        return doorY;
}
```

```
public double[] getCashRegisterX()
{
	return cshRegX;
}

public double[] getCashRegisterY()
{
	return cshRegY;
}

public boolean onCashRegistar(int r, int c)
{
	if (map[r][c].getItemId()=="cash register")
	{
		return true;
	}
	return false;
}

public boolean pathAhead(int r, int c)
{
	if (map[r][c].getItemId().equals(""))
	{
		return true;
	}
	return false;
}

public boolean isWall(int r, int c)
{
	if (map[r][c].getItemId()=="wall")
	{
		return true;
	}
	return false;
}

public int getNumInitial()
{
```

```
		return initialItems;
	}

	public int getNumRoughHandled()
	{
		return roughHandling;
	}

	public int getNumOpened()
	{
		return openedPackaging;
	}

	public int getNumDropped()
	{
		return dropping;
	}

	public int getNumBought()
	{
		return bought;
	}

	public void printMap()
	{
		for (int r=0; r<map.length; r++)
		{
			for (int i = 0; i<10*26+1; i++)
			{
				System.out.print("-");
			}
			System.out.println();

			for (int c=0; c<map[0].length; c++)
			{
				System.out.print("|"+center(""));
			}
			System.out.println("|");
```

```
                for (int c=0; c<map[0].length; c++)
                {
                        if (map[r][c].equals(null))
                        {
                                System.out.print("|"+center("null"));
                        }
                        else
                        {
                                System.out.print("|"+center(map[r][c].toString()));
                        }                       }
                System.out.println("|");

                for (int c=0; c<map[0].length; c++)
                {
                        System.out.print("|"+center(""));
                }
                System.out.println("|");
        }
        for (int i = 0; i<10*26+1; i++)
        {
                System.out.print("-");
        }
        System.out.println();
}

public String center(String string)
{
        int totalSpace=9;
        String newString="";
        if (string.length()<=totalSpace)
        {
                int firstHalf=(totalSpace-string.length())/2;
                for (int i = 0; i<firstHalf;i++)
                {
                        newString+=" ";
                }
                newString+=string;
                for (int i = 0; i<totalSpace-string.length()-firstHalf;i++)
                {
```

```
                        newString+=" ";
                    }

            }
            else
            {
                    newString=string.substring(0,totalSpace);
            }
            return newString;
    }

    public double getPopularityScore(int newR, int newC)
    {
            return map[newR][newC].getPopularityScore();
    }

    public String getDepartment(int newR, int newC)
    {
            return map[newR][newC].getItemId();
    }
}
```

item.java:

```
package immc;

public class item
{
    private int numberOfItems;
    private String itemId;
    private double popularity;
    private int totalArea;

    public item(String givenItem)
    {
            itemId=givenItem;
            numberOfItems=0;
            popularity=0;
            totalArea=4;
    }
```

```
public item(String givenItem, int givenNumber, double  givenPopularity, int givenArea)
{
	numberOfItems=givenNumber;
	itemId=givenItem;
	popularity=givenPopularity;
	totalArea=givenArea;
}

public void addItem()
{
	numberOfItems++;
}

public void removeItem()
{
	if (numberOfItems>0)
	{
		numberOfItems--;
	}
}

public String getItemId()
{
	return itemId;
}

public String toString()
{
	if (itemId.contentEquals("cash register"))
	{
		return itemId;
	}
	else if (itemId.contentEquals("wall"))
	{
		return "///////";
	}
	return itemId+" "+numberOfItems;
}
```

```
	public int currentItems()
	{
		return numberOfItems;
	}

	public double getPopularityScore()
	{
		return popularity;
	}

	public int getTotalArea()
	{
		return totalArea;
	}
}
```

probabilities.java:

```
package immc;

public class probabilities
{
	private int people;
	private int product;
	private double pp;
	private double[] weight;
	//coordinates, (x, y) : (c, r)
	private double[] cashierX;
	private double[] cashierY;
	private int departmentX;
	private int departmentY;
	private map currentMap;
	private int departmentArea;
	private int personArea=4;

	private double max_dist=48*Math.sqrt(2);
	//maximum people to product ratio
	private double max_ppp=15;
	//dropping percentage
```

```
	public probabilities(map givenMap, int departmentGivenR, int departmentGivenC, int personGivenR, int personGivenC)
	{
		currentMap=givenMap;
		pp = currentMap.getItem(departmentGivenR, departmentGivenC).getPopularityScore()/100;
		weight = new double[] {0.2, 0.1, 0.7, 0.7, 0.3, 0.425, 0.15, 0.425};  //In order of opening package, rough handling, and total damage score

		people = currentMap.getItemNum(personGivenR, personGivenC);
		product = currentMap.getItemNum(departmentGivenR, departmentGivenC);
		departmentArea=currentMap.getItem(departmentGivenR, departmentGivenC).getTotalArea();

		cashierX = currentMap.getCashRegisterX();
		cashierY = currentMap.getCashRegisterY();
		departmentX = departmentGivenC;
		departmentY = departmentGivenR;
	}

	public double calculateDropping()
	{
		return weight[7]*(people_per_product_metric());
	}

	public double calculateOpening()
	{
		double ppp = people_per_product_metric();
		double vis = visibility();
		return weight[6]*(weight[0] * ppp + weight[1] * pp + weight[2] * vis);
	}

	public double calculateRoughHandling()
	{
		double ppp = people_per_product_metric();
		double vis = visibility();
		return weight[5]*(weight[3] * ppp + weight[4] * vis);
	}

	//People per product metric
	public double people_per_product_metric()
```

```
	{
	   return population_density()/(product_density() * max_ppp);
	}

	public double population_density()
	{
	   return (double)people+1/personArea;
	}

	public double product_density()
	{
	   return (double)product/departmentArea;
	}

	public double visibility()
	{
		double distance = (max_dist-getDistance()/max_dist);
		return Math.pow(Math.E, (-1 * (1 - pp) * (Math.pow(distance, 2))));
	}

	private double getDistance()
	{
		double minDistance=(double)Integer.MAX_VALUE;
		for (int i = 0; i<7; i++)
		{
			double distance=(Math.sqrt(Math.pow((double)cashierX[i]*2.0-departmentX*2.0,
2)+Math.pow((double)cashierY[i]*2.0-departmentY*2.0, 2)));
			if (distance<minDistance)
			{
				minDistance=distance;
			}
		}
		return minDistance;
	}
}
```

Appendix II: Python Simulation

Department.py

```
'''
IMMC 2020

This Program models a store Department
using the Excel spreadsheet and Pygame.
'''

###########
# IMPORTS #
###########
import pygame as pg
import information as i
import utils
from utils import Point

class Department():
    '''
    A Store Department that is modeled by a rectangle.
    '''

    def __init__(self, pos, dim, color, dept, quantity, dept_area,
disp_area, pop_per):
        '''
        Constructor for the Department class.

        Parameters:
        -----------
```

```
        self: The Department object.\n
        pos (tuple): The coordinates of the center of the Department.\n
        dim (tuple): The dimensions of the Department in pixels.\n
        color (tuple): The RGB color of the Department.\n
        dept (str): The name of the Department.\n
        quantity (int): The total quantity of products in the Department.\n
        dept_area (float): The total area of the Department (m^2)\n
        disp_area (float): The display area of the Department (m^2)\n
        pop_per (float): The popularity percentage of the Department.
        '''

        self._pos = pos
        self._dim = dim
        self._color = color
        self._rect = self.make_rect()

        self._dept = dept
        self._quantity = quantity
        self._dept_area = dept_area
        self._disp_area = disp_area
        self._pop_per = pop_per

        self._clicked = False

    def make_rect(self):
        '''
        Makes the rectangle.

        Parameters:
```

```
        -----------
        self: The Department object.
        '''

        # x coordinate of the left side of the rect; x - (l/2)
        left = self._pos[0] - (self._dim[0]/2)

        # y coordinate of the top side of the rect; y + (w/2)
        top = i.SCREEN_HEIGHT - ( self._pos[1] + (self._dim[1]/2) )

        rect = (left, top, self._dim[0], self._dim[1])

        return pg.Rect(rect)

    def get_pos(self):
        '''
        Returns the center position of the Department depect.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._pos

    def set_pos(self, new_pos):
        '''
        Sets the pos.

        Parameters:
```

```
        -----------
        self: The Department object.
        '''
        self._pos = new_pos

    def get_dim(self):
        '''
        Returns the dimensions of the Department depect.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._dim

    def get_color(self):
        '''
        Returns the color of the Department object.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._color

    def get_rect(self):
        '''
        Returns the rectangle of the Department object.
```

```
        Parameters:
        -----------
        self: The Department object.
        '''
        return self._rect

    def get_dept(self):
        '''
        Returns the name of the Department object.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._dept

    def get_quantity(self):
        '''
        Returns the quantity of the Department object.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._quantity

    def get_dept_area(self):
        '''
        Returns the total area of the Department object.
```

```
        Parameters:
        -----------
        self: The Department object.
        '''
        return self._dept_area

    def get_disp_area(self):
        '''
        Returns the display area of the Department object.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._disp_area

    def get_pop_per(self):
        '''
        Returns the popularity percentage of the Department object.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._pop_per

    def get_clicked(self):
        '''
```

```
        Returns if the Department object has been clicked.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._clicked

    def get_num_people(self):
        '''
        Returns the number of people at the Department.

        Parameters:
        -----------
        self: The Department object.
        '''
        return self._pop_per * i.NUM_PEOPLE

    def set_clicked(self, clicked):
        '''
        Sets whether the Department object has been clicked.

        Parameters:
        -----------
        self: The Department object.
        '''
        self._clicked = clicked

    def get_pop_density(self):
```

```
        '''
        Calculates the population density.

        Parameters:
        -----------
        self: The Department object.
        '''
        return utils.population_density(i.NUM_PEOPLE, self._pop_per,
self._dept_area - self._disp_area)

    def get_prod_density(self):
        '''
        Calculates the product density.

        Parameters:
        -----------
        self: The Department object.
        '''
        return utils.product_density(self._quantity, self._disp_area)

    def dropping(self):
        '''
        Calculates the dropping score from the people to product ratio.

        Parameters:
        -----------
        self: The Department object.
        '''
```

```
        return utils.people_per_product_metric(self.get_pop_density(),
self.get_prod_density(), i.max_people_per_product)

    def rough_handling(self):
        '''
        Calculates the rough handling score as a function of
        obstruction of the department and the people to product
        ratio in the department.

        Parameters:
        -----------
        self: The Department object.
        '''
        ppp = utils.people_per_product_metric(self.get_pop_density(),
self.get_prod_density(), i.max_people_per_product)

        vises = []
        for c in i.cashiers:
          vis = utils.obstruction(Point(c.get_pos()[0], c.get_pos()[1]),
Point(self._pos[0], self._pos[1]), self._pop_per)

          vises.append(vis)

        vis_ = utils.model_obstructions(vises)

        return i.w_1_RH * ppp + i.w_2_RH * vis_

    def open_packaging(self):
        '''
```

```
        Calculates the open packaging score as a function of obstruction of
the department,
        the popularity score, and the people to product ratio in the
department.

        Parameters:
        -----------
        self: The Department object.
        '''
        ppp = utils.people_per_product_metric(self.get_pop_density(),
self.get_prod_density(), i.max_people_per_product)

        vises = []
        for c in i.cashiers:
          vis = utils.obstruction(Point(c.get_pos()[0], c.get_pos()[1]),
Point(self._pos[0], self._pos[1]), self._pop_per)

          vises.append(vis)
        vis_ = utils.model_obstructions(vises)

        return i.w_1_OP * ppp + i.w_2_OP * self._pop_per + i.w_3_OP * vis_
```

Utils.py

```
'''
IMMC 2020

This program contains the basic formulas
used in calculating the damage score.
'''

###########
# IMPORTS #
###########
import numpy as np

def people_per_product_metric(pop_density, prod_density, max_ratio):
    return pop_density / (prod_density * max_ratio)

def population_density(num_people, pop_perc, free_dept_area):
    return (num_people * pop_perc) / free_dept_area

def product_density(num_products, display_area):
    return num_products / display_area

def obstruction(cashier_pos, dep_pos, pop_per, max_dist =
(2*(480**2))**0.5):
    '''
    A Gaussian function with average change in decrease as a function of
the popularity percentage.
    '''

    dist = ((cashier_pos.distance(dep_pos))/(max_dist))
```

```
    # return np.e ** (-1 * (pop_per) * (dist ** 2))
    return dist


def model_obstructions(obstructions, sub_constant = 0.1):
  '''
  Calculates the final obstruction score based on the
  individual obstruction scores for each cashier.
  '''
  if len(obstructions) == 1:
    return obstructions[0]
  else:
    avg = sum(obstructions)/len(obstructions)
    obstructions.remove(max(obstructions))

    sub = 0
    for o in obstructions:
      sub += sub_constant * o

    return avg - sub

#########
# POINT #
#########
class Point:

    def __init__(self, x, y):
        self.x = x
        self.y = y
```

```
def distance(self, point2):
    return ((self.y - point2.y) ** 2 + (self.x - point2.x) ** 2) ** 0.5
```

Main.py

```
'''
IMMC 2020

This program runs the simulation and calculates
the number of products damaged.
'''

###########
# IMPORTS #
###########
import pygame as pg
import information as i
from Department import Department
from calculate_score import get_score

def main():

    # initializes pygame
    pg.init()
    Screen = pg.display.set_mode(i.DIM)
    pg.display.set_caption("IMMC Simulator")
    clock = pg.time.Clock()
    running = True

    # add cashier to departments
    deps = i.departments.copy()

    for r in i.cashiers:
      deps.append(r)
```

```
get_score()

# while the program is running
while running:

    # Events
    for event in pg.event.get():

        # if the program is quit, stop loop
        if event.type == pg.QUIT:
            running = False

        # if the mouse is down
        elif event.type == pg.MOUSEBUTTONDOWN: # [1]
            if event.button == 1:
                for dep in deps:
                    if dep.get_rect().collidepoint(event.pos):
                        dep.set_clicked(True)
                        offset_x = dep.get_rect().x - event.pos[0]
                        offset_y = dep.get_rect().y - event.pos[1]

        # if the mouse is not down
        elif event.type == pg.MOUSEBUTTONUP:
            if event.button == 1:
                for dep in deps:
                    dep.set_clicked(False)

        # if the mouse is moving and clicked is true, move dept
        elif event.type == pg.MOUSEMOTION:
```

```
            for dep in deps:
                if dep.get_clicked():
                    dep.get_rect().x = event.pos[0] + offset_x
                    dep.get_rect().y = event.pos[1] + offset_y
                    dep.set_pos(dep.get_rect().center)

                    # makes sure depts do not go out of bounds
                    if dep.get_rect().right > i.SCREEN_WIDTH:
                        dep.get_rect().right = i.SCREEN_WIDTH
                    if  dep.get_rect().left < 0:
                            dep.get_rect().left = 0
                    if  dep.get_rect().bottom > i.SCREEN_HEIGHT:
                            dep.get_rect().bottom = i.SCREEN_HEIGHT
                    if  dep.get_rect().top < 0:
                            dep.get_rect().top = 0

                    # gets the damage score
                    get_score()
                    print(dep.get_rect().center)

        # update the screen
        Screen.fill(i.WHITE)
        for dep in deps:
            pg.draw.rect(Screen, dep.get_color(), dep.get_rect())
        pg.display.flip()
        clock.tick(i.FPS)

pg.quit()
```

```
if __name__ == "__main__":
    main()


##############
# REFERENCES #
##############
'''
1. How to drag an object with Pygame:
https://blog.furas.pl/python-pygame-drag-object-on-screen-using-mouse-gb.html
'''
```