```java
package hardwareStore;

public class AccessoryKitOption extends RentalOption
{
	private final int COST = 8;

	public AccessoryKitOption()	{}

	public String getDescription()
	{
		return "Accessory Kit";
	}
	public int getCost()
	{
		return COST;
	}
}
```

```java
package hardwareStore;

/**
 * A business customer always rents 3 items for 7 days.
 *
 * @author Lucas
 *
 */
public class BusinessCustomer extends Customer
{
	public BusinessCustomer(String name, Store store)
	{
		super(name,store);
	}

	public int getType()
	{
		return CustomerType.BUSINESS;
	}

	protected boolean canRent()
	{
		// Only rent if the store has at least 3 items in inventory and the
customer has less than 3 tools rented
		return (store.getInventory().size() >= 3) && (getNumToolsRented() <
MAX_RENTALS);
	}

	protected int howLong()
	{
		// Business customers always rent for 7 days
		return 7;
	}

	protected int howMany()
	{
```

```
			// Business customers always rent 3 tools
			return 3;
		}
}


package hardwareStore;

/**
 * A casual customer rents 1-2 tools for 1-2 days.
 *
 * @author Lucas
 *
 */
public class CasualCustomer extends Customer
{
	private final static int MAX_RENTALS_CASUAL = 2;

	public CasualCustomer(String name, Store store)
	{
		super(name, store);
	}

	public int getType() {
		return CustomerType.CASUAL;
	}

	public int howMany()
	{
		int maxTools = Math.min(store.getInventory().size(), MAX_RENTALS -
getNumToolsRented());
		maxTools = Math.min(MAX_RENTALS_CASUAL, maxTools);

		int numTools = (int)(Math.random() * maxTools) + 1;

		return numTools;
	}

	public int howLong()
	{
		// Rent for [1, 2] days
		int days = (int)(Math.random() * 2) + 1;
		return days;
	}

	protected boolean canRent()
	{
		// Rent if the store has at least 1 tool in inventory and the customer
has less than 3 tools rented
		return (store.getInventory().size() != 0) && (getNumToolsRented() <
MAX_RENTALS);
	}
}
```

```java
package hardwareStore;

/**
 * Tool category types
 *
 * @author Alex
 *
 */
public final class CategoryType
{
	public final static int PAINTING = 0;
	public final static int CONCRETE = 1;
	public final static int PLUMBING = 2;
	public final static int WOODWORK = 3;
	public final static int YARDWORK = 4;

	public final static int TYPE_COUNT = 5;
}
```

```java
package hardwareStore;

public class ConcreteTool extends Tool
{
	public ConcreteTool(String name)
	{
		super(name);
	}

	public int getDailyPrice()
	{
		return 15;
	}
	public int getCategory()
	{
		return CategoryType.CONCRETE;
	}
}
```

```
package hardwareStore;


import java.util.ArrayList;

import java.util.Observable;

import java.util.Observer;


/**

 * A Customer visits a Store and rents Tools via RentalRecords.
```

```
 *
 * Implements the Observer part of the Observer pattern
 * in conjunction with Store, which implements Observable/Subject
 *
 * Also implements the Template pattern in the generateRental function,
 * which makes calls to abstract functions that are implemented by types of Customers.
 *
 * Also implements the Command pattern in conjunction with
 * RentalRecords and Store. Customer creates a RentalRecord and passes it to Store,
 * which executes it.
 *
 * @author Lucas
 *
 */
@SuppressWarnings("deprecation")
public abstract class Customer implements Observer
{
	protected String name;
	protected Store store;
	protected ArrayList<RentalRecord> orderList;

	// The max tools a customer can have rented out
	protected final static int MAX_RENTALS = 3;

	// The % chance that a customer goes to the store if they can go
	protected final static double CHANCE_GOES_TO_STORE = 0.2;

	/**
	 * Constructor for Customer
```

```
	 * @param name The name of the customer.
	 * @param store The store that the customer will visit.
	 */
	public Customer(String name, Store store)
	{
		this.name = name;
		this.store = store;
		store.addObserver(this);
		this.orderList = new ArrayList<RentalRecord>();
	}

	abstract public int getType();

	/**
	 * @return The customer's name.
	 */
	public String getName()
	{
		return this.name;
	}

	/**
	 * Determines whether a customer can go in to the store to rent something.
	 * @return True if the store has enough inventory and the customer hasn't reached their max
rentals
	 */
	protected abstract boolean canRent();

	/**
```

```
 * Randomly select if a customer goes to the store. 20% chance of happening.
 * @return true if they go to the store and false otherwise.
 */
protected boolean willRent()
{
        return Math.random() <= CHANCE_GOES_TO_STORE;
}

/**
 * Generate the number of tools to rent for a given rental.
 * Varies depending on customer type.
 * @return The number of tools to rent.
 */
protected abstract int howMany();


/**
 * Generate the number of days that a rental will last.
 * Varies depending no customer type.
 * @return The number of days to rent for.
 */
protected abstract int howLong();


/**
 * Generates a random number of options to add from [0, 6].
 * @return The number of options.
 */
protected int numOptions()
```

```
{
        int num = (int)(Math.random() * 7);
        return num;
}



/**
 * TEMPLATE PATTERN - Each customer will add tools to an order, add options to the tools, and rent for a certain amount of time
 * how many tools and how long they will rent the tools changes depending on the type of customer.
 * Creates a Rental Record with the items and options that the customer rents.
 * @param currentDay The current day of the simulation
 * @return The generated Rental Record.
 */

protected final RentalRecord generateRental(int currentDay)
{

        // First, determine how many tools the customer will rent
        int numTools = this.howMany();

        // The tools and options to rent
        ArrayList<RentalOption> options = new ArrayList<RentalOption>();
        ArrayList<Tool> tools = new ArrayList<Tool>();

        // The customer's store's current inventory
        ArrayList<Tool> storeInventory = store.getInventory();
```

```
                // Add tools to the rental
                for(int t = 1; t <= numTools; t++)
                {
                        int numOptions = this.numOptions();
                        tools.add(storeInventory.get(storeInventory.size() - t));
                        for(int o = 0; o < numOptions; o++)
                        {
                                RentalOption opt = RentalOption.getRandomRentalOption();
                                options.add(opt);
                        }
                }

                // Create and return a rental record
                RentalRecord rental = new RentalRecord(tools, options, this.howLong(), currentDay,
this);
                return rental;



        }



        /**
         * @return The number of tools that the customer currently has rented out.
         */
   public int getNumToolsRented()
   {
        int total = 0;
        for(RentalRecord rental : orderList)
        {
```

```
                total += rental.getRentedTools().size();
        }
        return total;
}



/**
 * Runs a day for the customer. If they decide to rent tools that day,
 * generates the rental and passes it to the customer's store.
 * @param currentDay The current day of the simulation.
 */
public void runDay(int currentDay)
        {
        // First, determine if we CAN and WILL go to the store
        if(canRent() && willRent())
                {
                        // If we go to the store, generate a rental
                        RentalRecord rental = generateRental(currentDay);

                        // Pass the rental to the store
                        store.startRental(rental);

                        // Add the rental to the customer's order list
                        this.orderList.add(rental);
                }

        }
```

```
    /**
     * Returns the tools from a Rental Record to the store.
     * @param record The Rental Record being returned.
     */
    private void returnTools(RentalRecord record)
    {
                store.processReturn(record);
    }

    public void update(Observable obj, Object arg)
    {
        RentalRecord toReturn = (RentalRecord)arg;
        if(toReturn.getCustomer() == this && orderList.remove(toReturn))
        {
                returnTools(toReturn);
        }
    }
}
```

```
package hardwareStore;

public class CustomerType {

        public final static int CASUAL = 0;
        public final static int REGULAR = 1;
        public final static int BUSINESS = 2;

        public final static int TYPE_COUNT = 3;

}
```

```
package hardwareStore;

public class ExtensionCordOption extends RentalOption
{
        private final int COST = 4;

        public ExtensionCordOption(){}
```

```
	public String getDescription()
	{
		return "Extension Cord";
	}
	public int getCost()
	{
		return COST;
	}
}
```

```
package hardwareStore;


import java.io.BufferedReader;

import java.io.FileOutputStream;

import java.io.InputStreamReader;

import java.io.PrintStream;


public class Main

{

	public static void main(String[] args)

	{

		// Ask the user if they want to output to the console or to a file

	BufferedReader input = new BufferedReader(new InputStreamReader(System.in));

		System.out.println("Print to console (c) or create a file (f)?");

		String inputString = "";


		try

		{

			inputString = input.readLine();


			// Keep asking until the user gives a valid input

			while(!inputString.equalsIgnoreCase("c") && !inputString.equalsIgnoreCase("f"))

			{
```

```
                    System.out.println("Invalid input: Please input 'c' for console or 'f' for
file.");
                    inputString = input.readLine();
                }
            }
            catch (Exception ex)
            {
                System.err.println("An exception occurred while trying to read input:\n" +
ex.getStackTrace());
            }

            // Output to file if the user input an f
            Boolean outputToFile = inputString.equalsIgnoreCase("f");
            if(outputToFile)
            {
                try
                {
                    System.out.println("Printing to hardwarestore.out!");
                    PrintStream fileStream = new PrintStream(new
FileOutputStream("hardwarestore.out"));
                    System.setOut(fileStream);

                }
                catch (Exception ex)
                {
                    System.err.println("An exception occurred while trying to create an
output file:\n" + ex.getStackTrace());
                }
            }
```

```
// Get the World
World aWholeNewWorld = World.getTheWorld();

// Create a Store
Store theStore = new Store("Bartlebee's Hardware Shack");

// Create and add tools to the Store

// Concrete Tools
theStore.addToolToInventory(new ConcreteTool("Concrete Mixer"));
theStore.addToolToInventory(new ConcreteTool("Concrete Powder"));
theStore.addToolToInventory(new ConcreteTool("Jackhammer"));
theStore.addToolToInventory(new ConcreteTool("Concrete Saw"));
theStore.addToolToInventory(new ConcreteTool("Sledgehammer"));

// Painting Tools
theStore.addToolToInventory(new PaintingTool("Paint Roller"));
theStore.addToolToInventory(new PaintingTool("Mixing Stick"));
theStore.addToolToInventory(new PaintingTool("Paint Brush"));
theStore.addToolToInventory(new PaintingTool("Tarp"));
theStore.addToolToInventory(new PaintingTool("Easel"));

// Woodworking Tools
theStore.addToolToInventory(new WoodworkTool("Mitre Saw"));
theStore.addToolToInventory(new WoodworkTool("Sawhorse"));
theStore.addToolToInventory(new WoodworkTool("Hammer"));
theStore.addToolToInventory(new WoodworkTool("Table Saw"));
theStore.addToolToInventory(new WoodworkTool("Hand Saw"));
```

```
// Yardworking Tools
theStore.addToolToInventory(new YardworkTool("Rake"));
theStore.addToolToInventory(new YardworkTool("Wheelbarrow"));
theStore.addToolToInventory(new YardworkTool("Hoe"));
theStore.addToolToInventory(new YardworkTool("Lawnmower"));
theStore.addToolToInventory(new YardworkTool("Sheers"));

// Plumbing Tools
theStore.addToolToInventory(new PlumbingTool("Plumbing Wrench"));
theStore.addToolToInventory(new PlumbingTool("Plunger"));
theStore.addToolToInventory(new PlumbingTool("Industrial Toilet Brush"));
theStore.addToolToInventory(new PlumbingTool("Blowtorch"));

// Add the Store to the World
aWholeNewWorld.addStore(theStore);

// Create Customers and add them to the World

Customer timApple = new RegularCustomer("Timothy Apple", theStore);
aWholeNewWorld.addCustomer(timApple);

Customer billMicrosoft = new CasualCustomer("William Microsoft", theStore);
aWholeNewWorld.addCustomer(billMicrosoft);

Customer larryGoogle = new BusinessCustomer("Laurence Google", theStore);
aWholeNewWorld.addCustomer(larryGoogle);

Customer trumanBurbank = new RegularCustomer("Truman Burbank", theStore);
aWholeNewWorld.addCustomer(trumanBurbank);
```

```
        Customer marcusFacebook = new RegularCustomer("Marcus Facebook", theStore);
        aWholeNewWorld.addCustomer(marcusFacebook);

        Customer frankNStein = new RegularCustomer("Frank N Stein", theStore);
        aWholeNewWorld.addCustomer(frankNStein);

        Customer michaelRotch = new CasualCustomer("Michael Rotch", theStore);
        aWholeNewWorld.addCustomer(michaelRotch);

        Customer hughJassman = new CasualCustomer("Hugh Jassman", theStore);
        aWholeNewWorld.addCustomer(hughJassman);

        Customer williamStroker = new CasualCustomer("William Stroker", theStore);
        aWholeNewWorld.addCustomer(williamStroker);

        Customer benjaminDover = new BusinessCustomer("Benjamin Dover", theStore);
        aWholeNewWorld.addCustomer(benjaminDover);

        Customer jackSmith = new BusinessCustomer("Jack Smith", theStore);
        aWholeNewWorld.addCustomer(jackSmith);

        Customer neilDown = new BusinessCustomer("Neil Down", theStore);
        aWholeNewWorld.addCustomer(neilDown);

        // Run the simulation for 35 days
        aWholeNewWorld.runSimulation(35);
    }
}
```

```java
package hardwareStore;

/**
 * RentalOption types and a total count of types
 *
 * @author Alex
 *
 */
public class OptionType
{
	public final static int PROTECTIVE_GEAR = 0;
	public final static int ACCESSORY_KIT = 1;
	public final static int EXTENSION_CORD = 2;

	public final static int TYPE_COUNT = 3;
}
```

```java
package hardwareStore;

public class PaintingTool extends Tool
{
	public PaintingTool(String name)
	{
		super(name);
	}

	public int getDailyPrice()
	{
		return 10;
	}
	public int getCategory()
	{
		return CategoryType.PAINTING;
	}
}
```

```java
package hardwareStore;

public class PlumbingTool extends Tool
{
	public PlumbingTool(String name)
	{
		super(name);
	}

	public int getDailyPrice()
	{
		return 5;
	}
	public int getCategory()
	{
		return CategoryType.PLUMBING;
	}
```

```
}
```

```
package hardwareStore;

public class ProtectiveGearOption extends RentalOption
{
	private final int COST = 12;

	public ProtectiveGearOption(){}

	public String getDescription()
	{
		return "Protective Gear Package";
	}
	public int getCost()
	{
		return COST;
	}
}
```

```
package hardwareStore;

/**
 * A Regular customer rents 1-3 tools for 3-5 days.
 *
 * @author Lucas
 *
 */
public class RegularCustomer extends Customer
{
	private final static int MAX_RENTALS_REGULAR = 3;

	public RegularCustomer(String name, Store store)
	{
		super(name,store);
	}

	public int getType() {
		return CustomerType.REGULAR;
	}

	public int howMany()
	{
		int maxTools = Math.min(store.getInventory().size(), MAX_RENTALS -
getNumToolsRented());
		maxTools = Math.min(MAX_RENTALS_REGULAR, maxTools);

		int numTools = (int)(Math.random() * maxTools) + 1;

		return numTools;
	}

	public int howLong()
```

```
	{
		// [3, 5] days
		int days = 3 + (int)(Math.random() * 3);
		return days;
	}

	protected boolean canRent()
	{
		// Rent if the store has at least 1 tool in inventory and the customer
has less than 3 tools rented
		return (store.getInventory().size() != 0) && (getNumToolsRented() <
MAX_RENTALS);
	}
}


package hardwareStore;

/**
 * Rental Options are optional additional items added to a Rental Record.
 * The price of a rental option applies to an entire rental, not on a daily basis.
 *
 * @author Ayden
 *
 */
public abstract class RentalOption
{
	/**
	 * @return The description of the rental option.
	 */
	public abstract String getDescription();

	/**
	 * @return The cost to add the rental option to a rental.
	 */
	public abstract int getCost();

	/**
	 * Generate a random rental option.
	 * @return The randomly generated rental option
	 */
	public static RentalOption getRandomRentalOption()
	{
		RentalOption result;
		int optionType = (int)(Math.random() * OptionType.TYPE_COUNT);

		switch(optionType)
		{
			case OptionType.ACCESSORY_KIT:
				result = new AccessoryKitOption();
				break;
			case OptionType.EXTENSION_CORD:
				result = new ExtensionCordOption();
				break;
			case OptionType.PROTECTIVE_GEAR:
```

```
                default:
                        result = new ProtectiveGearOption();
                        break;
                }

                return result;
        }
}
```

```
package hardwareStore;

import java.util.UUID;

import java.util.ArrayList;


/**
 * Rental Records are created by Customers and executed by Stores.
 * Represents a rental, including the tools rented,
 * options added, rental length, and total cost
 *
 * Implements the Command Pattern - created by Customer and executed by Store.
 *
 * @author Ayden
 *
 */
public class RentalRecord
{
        private UUID rentalID;
        private int rentalLength;
        private int dayRented;
        private int orderCost;
        private ArrayList<Tool> rentedTools;
        private ArrayList<RentalOption> options;
        private Customer rentalCustomer;
```

```
	public RentalRecord(ArrayList<Tool> rentedTools, ArrayList<RentalOption> options, int rentalLength, int today, Customer rentalCustomer)
	{
		this.rentedTools = rentedTools;
		this.options = options;
		this.rentalLength = rentalLength;
		this.rentalID = UUID.randomUUID();
		this.dayRented = today;
		this.rentalCustomer = rentalCustomer;

		// Calculate the total cost
		calculateCost();
	}

	/**
	 * @return The unique ID for the rental record.
	 */
	public UUID getID()
	{
		return this.rentalID;
	}

	/**
	 * @return The day that the tools were rented on.
	 */
	public int getDayRented()
	{
		return this.dayRented;
	}
```

```
/**
 * @return The number of days that the tools were rented for.
 */
public int getRentalLength()
{
	return this.rentalLength;
}

/**
 * @return The list of tools in the rental record.
 */
public ArrayList<Tool> getRentedTools()
{
	return this.rentedTools;
}

/**
 * @return The options added to the rental.
 */
public ArrayList<RentalOption> getOptions()
{
	return this.options;
}

/**
 * @return The total cost of the rental record.
 */
public int getCost()
```

```
{
	return this.orderCost;
}

public Customer getCustomer()
{
	return this.rentalCustomer;
}

/**
 * Calculates the total cost of the rental based on the
 * options and tools added to the RentalRecord
 */
private void calculateCost()
{
	// Initialize orderCost to 0
	orderCost = 0;

	// Add the cost of the options
	for (RentalOption option: options)
	{
		orderCost += option.getCost();
	}

	// Add the total cost of the tools (cost per day * total days)
	for (Tool tool: rentedTools)
	{
		orderCost += (tool.getDailyPrice() * rentalLength);
	}
```

```
}

public void printRentalDescription() {
	// Customer Name
	String printString = this.getCustomer().getName() + " rented: ";

	// Tools
	for (Tool tool: this.getRentedTools() )
	{
		printString += tool.getName() + ", ";
	}
	printString = printString.substring(0, printString.length() - 2);

	// Options
	if(this.options.size() != 0) {
		printString += " with Options ";
		for (RentalOption option : this.options)
		{
			printString += option.getDescription() + ", ";
		}
	}

	// Total Cost
	printString += " which cost $" + this.getCost();

	// Duration
	printString += " for " + this.getRentalLength() + " days.";
	System.out.println(printString);
}
```

```
}
package hardwareStore;

import java.util.ArrayList;
import java.util.Observable;

/**
 * A Store is visited by Customers and rents Tools to those customers.
 *
 * Implements the Observable/Subject part of the Observer pattern
 * in conjunction with Customer, which implements Observer.
 *
 * @author Alex
 *
 */
@SuppressWarnings("deprecation")
public class Store extends Observable
{
	private ArrayList<Tool> inventory;
	private ArrayList<RentalRecord> activeRentals;
	private ArrayList<RentalRecord> archivedRentals;
	private String name;

	/**
	 * Constructor for Store
	 * @param name The name of the store
	 */
	public Store(String name)
	{
```

```
        this.inventory = new ArrayList<Tool>();
        this.activeRentals = new ArrayList<RentalRecord>();
        this.archivedRentals = new ArrayList<RentalRecord>();
        this.name = name;
}

/**
 * Constructor for Store with an inventory
 * @param name The name of the store
 * @param inventory The list of Tools that exist in the Store's inventory
 */
public Store(String name, ArrayList<Tool> inventory)
{
        this.inventory = new ArrayList<Tool>(inventory);
        this.activeRentals = new ArrayList<RentalRecord>();
        this.archivedRentals = new ArrayList<RentalRecord>();
        this.name = name;
}

/**
 * @return The name of the store.
 */
public String getName()
{
        return this.name;
}

/**
 * Add a tool to the store's inventory.
```

```
 * @param t The tool to be added.
 */
public void addToolToInventory(Tool t)
{
	this.inventory.add(t);
}

/**
 * Remove a tool from the store's inventory.
 * @param t The tool to be removed.
 */
public void removeToolFromInventory(Tool t)
{
	this.inventory.remove(t);
}

/**
 * @return The store's current inventory.
 */
public ArrayList<Tool> getInventory()
{
	return this.inventory;
}

/**
 * Look through the active rental records at the store and notify any customers
 * who have rentals that are due to be returned. Should be run before the beginning
 * of each day.
 * @param currentDay The current day in the simulation.
```

```
 */
public void checkRentalRecords(int currentDay)
{
	ArrayList<RentalRecord> recordsToReturn = new ArrayList<RentalRecord>();

	// Loop through the list of active rentals
	for (RentalRecord record: this.activeRentals)
	{
		// Number of days that have passed since the rental record was created.
		int daysPassed = currentDay - record.getDayRented();

		// If the days passed is equal to the rental length, the rental is due
		if (daysPassed == record.getRentalLength())
		{
			recordsToReturn.add(record);
		}
	}

	for (RentalRecord record : recordsToReturn)
	{
		// Notify the customers that the record needs to be returned
		setChanged();
		notifyObservers(record);
	}
}

/**
 * Receives a rental record from the Customer and starts the rental
 * by removing the tools from inventory and adding the rental record
```

```
 * to the list of active records.
 * @param toStart The rental to start.
 */
public void startRental(RentalRecord rental)
{
	// Remove the tools in the rental from inventory
	for(Tool rentedTool : rental.getRentedTools())
	{
		removeToolFromInventory(rentedTool);
	}

	// Add the rental to the list of active rentals
	activeRentals.add(rental);
}

/**
 * Receives a rental record from the Customer and ends the renal
 * by adding the tools back to inventory and moving the rental record
 * from the list of active rentls to the list of archived rentals.
 * @param record The rental to end.
 */
public void processReturn(RentalRecord rental)
{
	// Return the tools to the inventory
	for (Tool rentedTool : rental.getRentedTools())
	{
		addToolToInventory(rentedTool);
	}
```

```
		// Move the rental record from active to archive
		activeRentals.remove(rental);
		archivedRentals.add(rental);
	}

	/**
	 * @return The list of archived rentals.
	 */
	public ArrayList<RentalRecord> getArchive()
	{
		return this.archivedRentals;
	}

	/**
	 * @return The list of active rentals.
	 */
	public ArrayList<RentalRecord> getActiveRentals()
	{
		return this.activeRentals;
	}

	public void printInventory() {
		String printString = this.getName() + "'s inventory has " + this.getInventory().size() + "
items: ";
		for (Tool tool : this.inventory)
		{
			printString += tool.getName() + ", ";
		}
		printString = printString.substring(0, printString.length() - 2);
```

```
            System.out.println(printString);
        }

        //Display the past archived tools that were rented and by whom
        public void printArchivedRecords()
        {
            System.out.println("\n" + this.getArchive().size() + " Completed Rentals for " +
this.getName() + ":");
            for (RentalRecord archivedRental : archivedRentals)
            {
                archivedRental.printRentalDescription();
            }
        }

        public void printActiveRecords() {
            System.out.println("\n" + this.getActiveRentals().size() + " Active Rentals for " +
this.getName() + ":");
            for (RentalRecord archivedRental : activeRentals)
            {
                archivedRental.printRentalDescription();
            }
        }

        /**
         * Calculate the current total money made by the store.
         * @return The calculated total.
         */
        public int calculateTotalSales()
        {
```

```
        int total = 0;

        // Money made from active rentals
        for(RentalRecord rec : activeRentals)
        {
                total += rec.getCost();
        }

        // Money made from archived rentals
        for(RentalRecord rec : archivedRentals)
        {
                total += rec.getCost();
        }

        return total;
}

public void printSalesByCustomerType()
{
        int businessRentals = 0;
        int casualRentals = 0;
        int regularRentals = 0;
        int completedRentals = 0;
        for(RentalRecord rec : archivedRentals)
        {
                if(rec.getCustomer().getType() == CustomerType.BUSINESS)
                {
                        businessRentals++;
                }
```

```
        else if(rec.getCustomer().getType() == CustomerType.CASUAL)
        {
                casualRentals++;
        }
        else
        {
                regularRentals = regularRentals + 1;
        }
        completedRentals++;
}

for(RentalRecord rec : activeRentals)
{
        if(rec.getCustomer().getType() == CustomerType.BUSINESS)
        {
                businessRentals++;
        }
        else if(rec.getCustomer().getType() == CustomerType.CASUAL)
        {
                casualRentals++;
        }
        else
        {
                regularRentals++;
        }

        completedRentals++;
}
```

```
                System.out.println("Total number of completed rentals: " + completedRentals);
                System.out.println("The business customers had: " + businessRentals + " rentals");
                System.out.println("The regular customers had: " + regularRentals + " rentals");
                System.out.println("The casual customers had: " + casualRentals + " rentals");
        }
}
package hardwareStore;

import static org.junit.Assert.assertEquals;
import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;

import org.junit.jupiter.api.Test;

class TestRent
{
        Store emptyStore = new Store("Empty Store");
        Store okStore = new Store("Some stuff");
        Store niceStore = new Store("Mos Eisley Cantina");

        Tool tool1 = new ConcreteTool("Hammer");
        Tool tool2 = new ConcreteTool("Impact Driver");
        Tool tool3 = new ConcreteTool("Lever");
        Tool tool4 = new ConcreteTool("Screw");
        Tool tool5 = new ConcreteTool("Saw");
        Tool tool6 = new ConcreteTool("Drill");
        Tool tool7 = new ConcreteTool("Lightsaber");
```

```
RentalOption option1 = new ExtensionCordOption();
RentalOption option2 = new ProtectiveGearOption();
RentalOption option3 = new AccessoryKitOption();

ArrayList<RentalOption> options = new ArrayList<RentalOption>();

@Test
void testStoreEmptyBusiness()
{
	Customer business = new BusinessCustomer("Jordan Belfort", emptyStore);
	if(emptyStore.getInventory().size() != 0)
	{
		fail("Empty store is not empty");
	}
	business.runDay(1);
	assertEquals(business.getNumToolsRented(), 0);
}

@Test
void testStoreEmptyCasual()
{
	Customer casual = new CasualCustomer("Zed", emptyStore);
	if(emptyStore.getInventory().size() != 0)
	{
		fail("Empty store is not empty");
	}
	casual.runDay(1);
	assertEquals(casual.getNumToolsRented(), 0);
}
```

```
	@Test
	void testStoreEmptyRegular() {
		Customer regular = new RegularCustomer("The Dude", emptyStore);
		if(emptyStore.getInventory().size() != 0)
		{
			fail("Empty store is not empty");
		}
		regular.runDay(1);
		assertEquals(regular.getNumToolsRented(), 0);
	}

	@Test
	void testStoreThreeItems() {
		okStore.addToolToInventory(tool7);
		okStore.addToolToInventory(tool5);
		okStore.addToolToInventory(tool5);

		Customer business = new BusinessCustomer("Jordan Belfort", okStore);

		if(okStore.getInventory().size() != 3)
		{
			fail("Store does not have exactly three items");
		}

		RentalRecord rental = business.generateRental(1);
		assertEquals(3, rental.getRentedTools().size());
	}
}
```

```
package hardwareStore;

import static org.junit.jupiter.api.Assertions.*;

import java.util.ArrayList;

import org.junit.jupiter.api.Test;

class TestStore
{
	Store s = new Store("Test");
	Customer c = new RegularCustomer("Tim", s);
	Tool concTool = new ConcreteTool("Test Concrete Tool");
	RentalOption extCordOpt = new ExtensionCordOption();

	final int RENTAL_LENGTH = 3;

	/**
	 * Check that a tool is in inventory after it has been added
	 * and no longer in inventory after it has been removed.
	 */
	@Test
	void testAddAndRemoveToolInventory()
	{
		s.addToolToInventory(concTool);

		// Inventory size should now be 1
		assertEquals(s.getInventory().size(), 1);
```

```
        // Tool in inventory should match our test tool
        assertEquals(s.getInventory().get(0), concTool);

        // Now, remove the tool from inventory
        s.removeToolFromInventory(concTool);

        // Inventory size should now be 0
        assertEquals(s.getInventory().size(), 0);
}

/**
 * Check that a tool is properly removed from inventory when starting a rental
 * and added back to inventory when a rental is completed.
 */
@Test
void testStartAndEndRental()
{
        // Add a test tool to the store's inventory
        s.addToolToInventory(concTool);

        // Tools list for the rental record, only containing the test tool
        ArrayList<Tool> tList = new ArrayList<Tool>();
        tList.add(concTool);

        // Options list for the rental record
        ArrayList<RentalOption> oList = new ArrayList<RentalOption>();
        oList.add(extCordOpt);

        // Create and start the rental record
```

```
                RentalRecord r = new RentalRecord(tList, oList, RENTAL_LENGTH, 0, c);
                s.startRental(r);

                // Inventory should now be empty
                assertEquals(0, s.getInventory().size());

                // There should now be 1 active rental, which is the rental option we created
                assertEquals(1, s.getActiveRentals().size());
                assertEquals(s.getActiveRentals().get(0), r);

                // Now, process the return
                s.processReturn(r);

                // Inventory should equal the tools list from the rental record since we checked every
tool out
                assertEquals(s.getInventory(), tList);

                // There should now be 0 active rentals
                assertEquals(0, s.getActiveRentals().size());

                // There should now be 1 archived rental, which is the rental option we created
                assertEquals(0, s.getActiveRentals().size());
                assertEquals(s.getArchive().get(0), r);
        }

        /**
         * Test that the total calculated sales is correct
         */
        @Test
```

```
	void testCalculateTotal()
	{
		// Run the test rental again to set up
		testStartAndEndRental();

		// Correct total is the daily price of concTool * the length of the rental + the cost of the extension cord option
		int correctTotal = concTool.getDailyPrice() * RENTAL_LENGTH;
		correctTotal += extCordOpt.getCost();

		// Make sure that the calculated total is equal to the correct total
		assertEquals(s.calculateTotalSales(), correctTotal);
	}
}
package hardwareStore;

import static org.junit.jupiter.api.Assertions.*;

import org.junit.jupiter.api.Test;

class TestWorld
{
	World hitchhikersWorld = World.getTheWorld();
	Store hardwareStore = new Store("The Hard Store");
	Customer newCustomer = new CasualCustomer("Steve", hardwareStore);

	@Test
	void testAddCustomer()
	{
```

```
        //Test adding one customer
        hitchhikersWorld.addCustomer(newCustomer);

        assertEquals(hitchhikersWorld.getCustomers().size(), 1);

        ////Test adding two customers at once
        Customer new2Customer = new CasualCustomer("Steve2", hardwareStore);
        Customer new3Customer = new CasualCustomer("Steve3", hardwareStore);

        hitchhikersWorld.addCustomer(new2Customer);
        hitchhikersWorld.addCustomer(new3Customer);

        assertEquals(hitchhikersWorld.getCustomers().size(), 3);
}

//Ensure the world can add a store properly
@Test
void testAddStore()
{
        hitchhikersWorld.addStore(hardwareStore);

        assertEquals(hitchhikersWorld.getStores().size(), 1);
}

//Ensure the world is able to increment the days properly
@Test
void testCurrentDays()
{
```

```
		hitchhikersWorld.startNewDay();


		assertEquals(hitchhikersWorld.getStagedDay(), 2);
	}


	//Ensure the world is able to increment the days properly
	@Test
	void testRunSimulation()
	{
		hitchhikersWorld.runSimulation(4);


		// At the start of the 6th day, but have yet to perform the actions of that day.
		assertEquals(hitchhikersWorld.getStagedDay(), 6);
	}



}
```

```
package hardwareStore;

/**
 * Tools are rented to Customers from Stores via Rental Records.
 *
 * @author Alex
 *
 */
public abstract class Tool
{
	protected String name;

	/**
	 * Constructor for Tools
	 * @param name The name of the tool
	 */
	public Tool(String name)
	{
		this.name = name;
	}

	/**
```

```
	 * @return The name of the tool.
	 */
	public String getName()
	{
		return name;
	}

	/**
	 * @return The cost per day for renting the tool based on the category that
the tool is a part of.
	 */
	public abstract int getDailyPrice();

	/**
	 * @return The category that the tool is in. Category types are defined in the
CategoryType class.
	 */
	public abstract int getCategory();
}
```

```
package hardwareStore;

public class WoodworkTool extends Tool
{
	public WoodworkTool(String name)
	{
		super(name);
	}

	public int getDailyPrice()
	{
		return 20;
	}
	public int getCategory()
	{
		return CategoryType.WOODWORK;
	}
}
```

```
package hardwareStore;

import java.util.ArrayList;

/**
 * A World contains Customers and Stores and
 * handles running a simulation for a given number of days.
 *
 * Implements the Singleton pattern - there can only be one World in a program
 * which is created/accessed using the getTheWorld function.
 *
 * @author Alex
 *
 */
```

```java
public class World
{
	private static World theWorld;
	private int stagedDay;
	private ArrayList<Store> stores;
	private ArrayList<Customer> customers;

	private World()
	{
		stores = new ArrayList<Store>();
		customers = new ArrayList<Customer>();
		stagedDay = 1;
	}

	/**
	 * Increment the current day then run through the logic for stores and
customers for each day
	 */
	public void startNewDay()
	{
		if(stagedDay != 1)
		{
			System.out.println("\n---------------\n");
		}
		else
		{
			System.out.println("\n");
		}


		System.out.println("Cue the sun. Day " + stagedDay + " is starting
now.\n");

		// Print current inventory, active records, and completed records
		for (Store store: stores) {
			store.printInventory();
			store.printActiveRecords();
			store.printArchivedRecords();
		}

		// Loop through each store
		for (Store store : stores)
		{
			// Check if they have any due rentals and notify customers
			store.checkRentalRecords(stagedDay);
		}

		// Now, loop through each customer and have them run through their day
		for(Customer customer: customers)
		{
			customer.runDay(stagedDay);
		}

		// Stage the next day by incrementing the day counter
		stagedDay++;
```

```
	}

	/**
	 * Run the simulation for a given number of days. This calls the
	 * startNewDay function days number of times. If run multiple times,
	 * will continue
	 * @param days The number of days to run the simulation for.
	 */
	public void runSimulation(int days)
	{
		int startDay = stagedDay;
		while(stagedDay - startDay < days)
		{
			startNewDay();
		}

		System.out.println("\n---------------\n");

		for(Store s : stores)
		{
			int total = s.calculateTotalSales();

			s.printSalesByCustomerType();

			System.out.println("Cha-ching! " + s.getName() + " made a total
of: $" + total);

		}
	}

	public void addStore(Store s)
	{
		stores.add(s);
	}

	public void addCustomer(Customer c)
	{
		customers.add(c);
	}

	public int getStagedDay() {
		return this.stagedDay;
	}

	public ArrayList<Customer> getCustomers() {
		return this.customers;
	}

	public ArrayList<Store> getStores() {
		return this.stores;
	}

	public static World getTheWorld()
	{
		if(theWorld == null)
```

```
                {
                        theWorld = new World();
                }

                return theWorld;
        }

}
```

```
package hardwareStore;

public class YardworkTool extends Tool
{
        public YardworkTool(String name)
        {
                super(name);
        }

        public int getDailyPrice()
        {
                return 28;
        }
        public int getCategory()
        {
                return CategoryType.YARDWORK;
        }
}
```