**UML**

The full UML class diagram that we came up with in the previous lecture looks like this.

A diagram of a diagram

Description automatically generated

**Source Code**

Note: A zip file containing the full source code in a Maven project is attached to this lecture and available for download.

In this section, I am going to discuss the source code that I wrote for this example and how it relates back to the UML diagram that I created.  Our project structure is as follows:

1. examples
2. - business
3. - DeliveryCalculator.java
4. - DeliveryCalculatorImpl.java
5. - OrderController.java
6. - domain
7. - Address.java
8. - Category.java
9. - Customer.java
10. - DeliveryPriority.java
11. - Item.java
12. - OrderHeader.java
13. - OrderLine.java
14. - integration
15. - CreditCardPaymentGateway.java
16. - CryptoPaymentGateway.java
17. - PaymentGateway.java
18. - reports
19. - SalesReport.java
20. - Main.java //Only for testing - see attached zip file.
21. - TestData.java //Only for testing - see attached zip file.

**Domain**

We'll start with our domain classes - these classes represent the objects that our system deals with.

Firstly, we have items.  Items belong to a category, but categories can exist without any items in them - hence, we modeled an association relationship on our diagram.  We also need to know if items are restricted so that we don't sell them to under-age customers.  Instead of exposing the category on each item and potentially breaking encapsulation (because we reveal underlying implementation details to users of the class), we rather expose a method on the item that tells us whether or not the item is in a restricted category.  Under the covers, this method just delegates straight to the Category object.

Item.java

1. package examples.domain;
3. public class Item {
5. private String itemId;
6. private String name;
7. private String description;
8. private double price;
10. private Category category;
12. public Item() {
13. }
15. public String getItemId() {
16. return itemId;
17. }
19. public void setItemId(String itemId) {
20. this.itemId = itemId;
21. }
23. public String getName() {
24. return name;
25. }
27. public void setName(String name) {
28. this.name = name;
29. }
31. public String getDescription() {
32. return description;
33. }
35. public void setDescription(String description) {
36. this.description = description;
37. }
39. public double getPrice() {
40. return price;
41. }
43. public void setPrice(double price) {
44. this.price = price;
45. }
47. public Category getCategory() {
48. return category;
49. }
51. public void setCategory(Category category) {
52. this.category = category;
53. }
55. public boolean isRestricted() {
56. return category.isRestricted();
57. }
58. }

Category.java

1. package examples.domain;
3. public class Category {
5. private String name;
6. private boolean restricted;
8. public Category(String name, boolean restricted) {
9. this.name = name;
10. this.restricted = restricted;
11. }
13. public boolean isRestricted() {
14. return restricted;
15. }
17. public void setRestricted(boolean restricted) {
18. this.restricted = restricted;
19. }
21. public String getName() {
22. return name;
23. }
25. public void setName(String name) {
26. this.name = name;
27. }
28. }

Customers, as you know from our requirements and our diagram, have a single address and they cannot exist without an address - hence, we have a composition relationship.  However, that is a logical restriction in our system - it's not really enforced in code.  If we don't configure an address for a customer, our system will fail somewhere down the line though - likely while attempting to process a delivery.

In addition, we care about the age of our customers; we can't sell restricted items to customers under 18 years old!  In order to implement this requirement, I've added an 'age' variable to the customer.  This variable is only calculated when it's first requested (in other words, we don't spend processing effort before we need to).  In a real-world system, we would have to keep in mind that this would have to be updated if a customer's date of birth happens to be changed.

Customer.java

1. package examples.domain;
3. import java.util.Calendar;
4. import java.util.Date;
6. public class Customer {
8. private String customerId;
9. private String name;
10. private String emailAddress;
11. private Date dateOfBirth;
12. private Address address;
14. private int age;
16. public Customer() {
17. }
19. public String getCustomerId() {
20. return customerId;
21. }
23. public void setCustomerId(String customerId) {
24. this.customerId = customerId;
25. }
27. public String getName() {
28. return name;
29. }
31. public void setName(String name) {
32. this.name = name;
33. }
35. public String getEmailAddress() {
36. return emailAddress;
37. }
39. public void setEmailAddress(String emailAddress) {
40. this.emailAddress = emailAddress;
41. }
43. public Date getDateOfBirth() {
44. return dateOfBirth;
45. }
47. public void setDateOfBirth(Date dateOfBirth) {
48. this.dateOfBirth = dateOfBirth;
49. }
51. public Address getAddress() {
52. return address;
53. }
55. public void setAddress(Address address) {
56. this.address = address;
57. }
59. public int getAge() {
60. //Lazy-load age when it's needed.
61. if (age == 0) {
62. Calendar birthCalendar = Calendar.getInstance();
63. birthCalendar.setTime(dateOfBirth);
65. Calendar now = Calendar.getInstance();
67. age = now.get(Calendar.YEAR) - birthCalendar.get(Calendar.YEAR);
68. }
70. return age;
71. }
72. }

Address.java

1. package examples.domain;
3. public class Address {
5. private String addressLine1;
6. private String addressLine2;
7. private String city;
8. private String postalCode;
10. public Address() {
11. }
13. public String getAddressLine1() {
14. return addressLine1;
15. }
17. public void setAddressLine1(String addressLine1) {
18. this.addressLine1 = addressLine1;
19. }
21. public String getAddressLine2() {
22. return addressLine2;
23. }
25. public void setAddressLine2(String addressLine2) {
26. this.addressLine2 = addressLine2;
27. }
29. public String getCity() {
30. return city;
31. }
33. public void setCity(String city) {
34. this.city = city;
35. }
37. public String getPostalCode() {
38. return postalCode;
39. }
41. public void setPostalCode(String postalCode) {
42. this.postalCode = postalCode;
43. }
44. }

Now that we have our items and customers, we can bring it all together in the order-related classes.  Orders consist of an OrderHeader with multiple OrderLine objects (a composition relationship).  An OrderHeader cannot be created without a customer, although customers can exist without having any orders.  We model this as an association, and enforce the constraint by creating an OrderHeader constuctor that accepts a Customer object as parameter.  Logically, an order cannot exist without at least one OrderLine.  We use the OrderHeader class to create order lines, since those lines can then explicitly belong only to the OrderHeader instance that created them.  We also expose a 'hasRestrictedItems()' method on the OrderHeader class so that any classes that process orders can check for restricted items without having to delve into the underlying implementation details of the OrderHeader class (i.e. by checking each individual line item manually).  Our OrderLine class can calculate the total cost of each line on the order.

Our orders also have a delivery priority (represented by the DeliveryPriority enum) that will be used to calculate a delivery fee.  However, that calculation doesn't happen on the OrderHeader object- it happens in a calculator class elsewhere.  As such, it's not modeled as a derived attribute.  The order total, however, is just the sum of the totals on each order line, so that is modeled as a derived attribute on the OrderHeader class.

If you look back at the diagram, you might notice that we didn't include a dependency relationship between the OrderHeader and DeliveryPriority classes.  Instead, I modeled the DeliveryPriority as an attribute on the OrderHeader.  Both options are acceptable and it could have been a dependency, but that would have resulted in a messier diagram that would have been harder to read.  Remember, try to keep your diagrams simple and easy-to-read, lest they lose their value due to too much complexity.

DeliveryPriority.java

1. package examples.domain;
3. public enum DeliveryPriority {
5. NORMAL, OVERNIGHT;
6. }

OrderHeader.java

1. package examples.domain;
3. import java.util.ArrayList;
4. import java.util.Collections;
5. import java.util.Date;
6. import java.util.List;
8. public class OrderHeader {
10. private String orderId;
11. private Date orderDate;
12. private double deliveryFee;
13. private DeliveryPriority priority;
14. private Customer customer;
16. private List<OrderLine> orderLines = new ArrayList<OrderLine>();
18. private double orderTotal;
20. public OrderHeader(Customer customer) {
21. this.customer = customer;
23. this.orderId = "ORD" + System.currentTimeMillis();
24. this.orderDate = new Date(); //Now.
25. }
27. public void addLine(Item item, int quantity) {
28. OrderLine newLine = new OrderLine();
29. newLine.setItem(item);
30. newLine.setQuantity(quantity);
32. this.orderLines.add(newLine);
33. }
35. public String getOrderId() {
36. return orderId;
37. }
39. public void setOrderId(String orderId) {
40. this.orderId = orderId;
41. }
43. public Date getOrderDate() {
44. return orderDate;
45. }
47. public void setOrderDate(Date orderDate) {
48. this.orderDate = orderDate;
49. }
51. public double getDeliveryFee() {
52. return deliveryFee;
53. }
55. public void setDeliveryFee(double deliveryFee) {
56. this.deliveryFee = deliveryFee;
57. }
59. public DeliveryPriority getPriority() {
60. return priority;
61. }
63. public void setPriority(DeliveryPriority priority) {
64. this.priority = priority;
65. }
67. public Customer getCustomer() {
68. return customer;
69. }
71. public void setCustomer(Customer customer) {
72. this.customer = customer;
73. }
75. public List<OrderLine> getOrderLines() {
76. return Collections.unmodifiableList(orderLines);
77. }
79. public double getOrderTotal() {
80. //Lazy-load order total when it's needed.
81. if (orderTotal == 0) {
82. for (OrderLine line : orderLines) {
83. orderTotal += line.getLineTotal();
84. }
85. }
87. return orderTotal;
88. }
90. public boolean hasRestrictedItems() {
91. for (OrderLine line : orderLines) {
92. if (line.getItem().isRestricted()) {
93. return true;
94. }
95. }
97. return false;
98. }
99. }

OrderLine.java

1. package examples.domain;
3. public class OrderLine {
5. private Item item;
6. private int quantity;
8. public OrderLine() {
9. }
11. public Item getItem() {
12. return item;
13. }
15. public void setItem(Item item) {
16. this.item = item;
17. }
19. public int getQuantity() {
20. return quantity;
21. }
23. public void setQuantity(int quantity) {
24. this.quantity = quantity;
25. }
27. public double getLineTotal() {
28. return item.getPrice() \* quantity;
29. }
30. }

**Business Logic & Integration**

The classes in our business logic and integration layer are responsible for processing orders and payments.

On our diagram, we modeled an DeliveryCalculator interface - this interfaces defines the contract for calculating delivery fees.  We also modeled a DeliveryCalculatorImpl class that implements (or in UML terms, realizes) this interface.  Because we code to an interface, we can provide alternative implementations to anything that requires a DeliveryCalculator - like a stub for testing.

Our DeliveryCalculatorImpl class also has a dependency on the DeliveryPriority enum, since fees are determined by the delivery priority.  We also include two constant values on the class - the fixed fee for overnight delivery, and the minimum order amount to waive delivery fees.

Lastly, we take distance into account when calculating delivery, but since the distance calculation is only ever relevant to the delivery fee calculation, there's no need to ever expose it to anything outside of DeliveryCalculatorImpl class - hence, it's a protected method.  I've simplified the logic for the example, but in the real-world, a method like this would probably make a call to a service somewhere.

DeliveryCalculator.java

1. package examples.business;
3. import examples.domain.OrderHeader;
5. public interface DeliveryCalculator {
7. double calculateDelivery(OrderHeader order);
8. }

DeliveryCalculatorImpl.java

1. package examples.business;
3. import examples.domain.Address;
4. import examples.domain.DeliveryPriority;
5. import examples.domain.OrderHeader;
7. public class DeliveryCalculatorImpl implements DeliveryCalculator {
9. private static final double OVERNIGHT\_DELIVERY\_FEE = 75;
10. private static final double DELIVERY\_FEE\_WAIVE\_AMOUNT = 50;
12. @Override
13. public double calculateDelivery(OrderHeader order) {
14. DeliveryPriority priority = order.getPriority();
16. if (priority == DeliveryPriority.OVERNIGHT) {
17. return OVERNIGHT\_DELIVERY\_FEE;
18. } else {
19. if (order.getOrderTotal() > DELIVERY\_FEE\_WAIVE\_AMOUNT) {
20. return 0; //Free delivery
21. }
23. int distance = calculateDistance(order.getCustomer().getAddress());
24. return distance \* 0.5; //Simple demo calculation - probably not a good idea in real life!
25. }
26. }
28. protected int calculateDistance(Address customerAddress) {
29. //Invoke a service somewhere to calculate distance from our warehouse to the customer's address.
30. return 42;
31. }
32. }

When we looked at our requirements, we said that we're going to cater for two different payment gateways.  Again, this is something that we can model as an interface (PaymentGateway) that defines the contract for integrating with a payment gateway.  We provide two implementations for this interface - one for credit card payments, and another one for cryptocurrency payments.

When we make a payment, the code that initiates that payment shouldn't have to care about *how* the payment is done.  Instead, it will delegate that responsibility to a particular PaymentGateway implementation (which can be configured via dependency-injection) that knows *how*to process a payment.

Our two payment gateways implement the specifics of how payments should be processes (via an authenticated request to a specific IP address and port for credit card payments, or via an API key for cryptocurrency payments).

PaymentGateway.java

1. package examples.integration;
3. import examples.domain.Customer;
5. public interface PaymentGateway {
7. void processPayment(Customer customer, double amount);
8. }

CreditCardPaymentGateway.java

1. package examples.integration;
3. import examples.domain.Customer;
5. public class CreditCardPaymentGateway implements PaymentGateway {
7. private String targetServerIp;
8. private int port;
10. @Override
11. public void processPayment(Customer customer, double amount) {
12. System.out.printf("Invoking credit card payment gateway on %s:%d%n", targetServerIp, port);
14. //Unimplemented; high-level flow is below.
15. //Establish connection to target IP on specified port.
16. authenticate();
17. //Create request to credit card payment gateway.
18. //Send request.
20. System.out.println("Payment successful!");
21. }
23. protected void authenticate() {
24. //Unimplemented; high-level flow is below.
25. //Send authentication request to credit card payment gateway.
26. }
28. public String getTargetServerIp() {
29. return targetServerIp;
30. }
32. public void setTargetServerIp(String targetServerIp) {
33. this.targetServerIp = targetServerIp;
34. }
36. public int getPort() {
37. return port;
38. }
40. public void setPort(int port) {
41. this.port = port;
42. }
43. }

CryptoPaymentGateway.java

1. package examples.integration;
3. import examples.domain.Customer;
5. public class CryptoPaymentGateway implements PaymentGateway {
7. private String url;
8. private String apiKey;
10. public CryptoPaymentGateway(String apiKey) {
11. this.apiKey = apiKey;
12. }
14. @Override
15. public void processPayment(Customer customer, double amount) {
16. System.out.printf("Invoking cryptocurrency payment gateway on %s with API key: %s%n", url, apiKey);
18. //Unimplemented; high-level flow is below.
19. //Create request to cryptocurrency payment gateway.
20. //Send request to URL with API key.
21. System.out.println("Payment successful!");
22. }
24. public String getUrl() {
25. return url;
26. }
28. public void setUrl(String url) {
29. this.url = url;
30. }
31. }

Our OrderController class is responsible for bringing it all together and actually processing orders.  It has association relationships with both the PaymentGateway and DeliveryCalculator interfaces.  It also has a dependency on the OrderHeader class, since it's responsible for creating orders.

The OrderController class is also responsible for validating orders, but since that logic isn't relevant anywhere outside of this class, the 'validateOrder()' method can be implemented as a protected method.

1. package examples.business;
3. import examples.domain.Customer;
4. import examples.domain.OrderHeader;
5. import examples.integration.PaymentGateway;
7. public class OrderController {
9. private DeliveryCalculator deliveryCalculator;
10. private PaymentGateway paymentGateway;
12. public OrderController() {
13. }
15. public OrderHeader createOrder(Customer customer) {
16. return new OrderHeader(customer);
17. }
19. public void processOrder(OrderHeader order) {
20. if (!validateOrder(order)) {
21. throw new IllegalStateException("Order validation failed - customers under 18 cannot buy restricted items.");
22. }
24. double deliveryFee = deliveryCalculator.calculateDelivery(order);
25. order.setDeliveryFee(deliveryFee);
27. paymentGateway.processPayment(order.getCustomer(), order.getOrderTotal() + deliveryFee);
28. }
30. protected boolean validateOrder(OrderHeader order) {
31. int customerAge = order.getCustomer().getAge();
33. if (customerAge <= 18 && order.hasRestrictedItems()) {
34. return false;
35. }
37. return true;
38. }
40. //Here, our setters are a rudimentary form of dependency injection.
41. public void setDeliveryCalculator(DeliveryCalculator deliveryCalculator) {
42. this.deliveryCalculator = deliveryCalculator;
43. }
45. public void setPaymentGateway(PaymentGateway paymentGateway) {
46. this.paymentGateway = paymentGateway;
47. }
48. }

**Reporting**

The last remaining bit of functionality that we need to implement is reporting for the CFO.

A SalesReport object aggregates a number of OrderHeader objects, and then writes out a simple report to the command line.  In a real application, this report might take the form of a spreadsheet or PDF document.

I wanted sales reports to be created in a very specific manner.  To accomplish this, I modeled the class with a static method that acts as a factory method for creating sales reports - and the class has a private constructor, so no one can create instances of it without using my static factory method.

I also included a constructor that accepts a list of OrderHeaders, so that we can't try to generate a report without specifying the orders that should be included on the report.

1. package examples.reports;
3. import examples.domain.OrderHeader;
4. import examples.domain.OrderLine;
6. import java.util.List;
8. public class SalesReport {
10. protected List<OrderHeader> orders;
12. private SalesReport(List<OrderHeader> orders) {
13. this.orders = orders;
14. }
16. public static SalesReport generateReport(List<OrderHeader> orders) {
17. return new SalesReport(orders);
18. }
20. public void display() {
21. //Write simple output to console.
22. System.out.println("===SALES REPORT===");
24. for (OrderHeader order : orders) {
25. System.out.println("Order: " + order.getOrderId() + ", Date: " + order.getOrderDate() + ", Total: " + order.getOrderTotal());
26. }
28. System.out.println("====END REPORT====");
29. }
31. }

**Summary**

In this article, we looked at the source code that implements the UML diagram for a simple order processing system that we created in the previous lecture.  Note how the code clearly follows the structure represented on the diagram, although the diagram doesn't show us what the code is actually doing in terms of how we write our methods.  Again, remember that class diagrams are structural diagrams - they show us*what*something looks like, not necessarily how it works.