California State University Fullerton

CPSC 462



Object Oriented Software Design

SW Architecture Document (SAD)

for the



Tuffy Flights

System

|  |  |  |
| --- | --- | --- |
| **Nathan Marcos** | **Jared Castaneda** | **Jawad Swed** |
| Design Modeling, Implementation | Project Management, Implementation | Business Modeling, Design Modeling |
| [nathanmarcos@csu.fullerton.edu](mailto:nathanmarcos@csu.fullerton.edu) | [jaredcast@csu.fullerton.edu](mailto:jaredcast@csu.fullerton.edu) | [jawadswed@csu.fullerton.edu](mailto:jawadswed@csu.fullerton.edu) |

Revision History:

| Version | Date | Summary of Changes | Author |
| --- | --- | --- | --- |
| 1.0 | 9/28/2020 | * Initial Release | Nathan Marcos  Jared Castaneda  Jawad Swed |
| 1.1 | 10/15/2020 | * Added a Creator GRASP decision * Added an Information Expert GRASP decision * Added a Controller GRASP decision * Added Package Diagram * Added Domain Layer Interface * Added Technical Services Interface * Wrote about components | Nathan Marcos  Jared Castaneda  Jawad Swed |
| 1.2 | 12/5/2020 | * Removed the old design model from the GRASP decisions and added the new design model into them * Edited decisions, making them more geared towards what was in textbook * Edited roles * Completed logical view | Nathan Marcos  Jared Castaneda  Jawad Swed |

Table of Contents

[1 Architectural Representation 1](#_Toc58187199)

[2 Architectural Decisions 2](#_Toc58187200)

[2.1 Low Coupling / High Cohesion GRASP Decision 2](#_Toc58187201)

[2.1.1 Decision to be made 2](#_Toc58187202)

[2.1.2 Options Considered 2](#_Toc58187203)

[2.1.3 Selection and Rationale 3](#_Toc58187204)

[2.2 Creator GRASP Decision 3](#_Toc58187205)

[2.2.1 Decision to be made 3](#_Toc58187206)

[2.2.2 Options Considered 4](#_Toc58187207)

[2.2.3 Selection and Rationale 5](#_Toc58187208)

[2.3 Information Expert GRASP Decision 5](#_Toc58187209)

[2.3.1 Decision to be made 5](#_Toc58187210)

[2.3.2 Options Considered 6](#_Toc58187211)

[2.3.3 Selection and Rationale 7](#_Toc58187212)

[2.4 Controller GRASP Decision 7](#_Toc58187213)

[2.4.1 Decision to be made 7](#_Toc58187214)

[2.4.2 Options Considered 7](#_Toc58187215)

[2.4.3 Selection and Rationale 8](#_Toc58187216)

[3 Logical View 9](#_Toc58187217)

[3.1 Package Diagrams 9](#_Toc58187218)

[3.1.1 Presentation (UI) Layer Components 9](#_Toc58187219)

[3.1.2 Domain (Application) Layer Components 9](#_Toc58187224)

[3.1.2.1 Customer 9](#_Toc58187225)

[3.1.2.2 FlightApp 9](#_Toc58187226)

[3.1.2.3 Session 9](#_Toc58187227)

[3.1.3 Technical Services Layer Components 10](#_Toc58187228)

[3.1.3.1 Logging 10](#_Toc58187229)

[Persistence 10](#_Toc58187230)

[3.1.3.2 10](#_Toc58187231)

[3.2 Interface Diagrams 11](#_Toc58187232)

[3.2.1 Presentation (UI) Layer Interface Diagrams 11](#_Toc58187233)

[3.2.2 Domain Layer Interface Diagrams 11](#_Toc58187238)

[3.2.3 Technical Services Interface Diagrams 12](#_Toc58187240)

[12](#_Toc58187241)

[3.3 Design Patterns 12](#_Toc58187242)

[3.3.1 Polymorphism GRASP Pattern 12](#_Toc58187243)

[3.3.1.1 Generalization / Specialization Diagrams 12](#_Toc58187244)

[3.3.1.2 Factory Pattern Diagrams 13](#_Toc58187245)

[3.3.1.3 Source Code References 13](#_Toc58187246)

[3.3.2 Protected Variations GRASP Pattern 13](#_Toc58187247)

[3.3.2.1 Generalization / Specialization Diagrams 13](#_Toc58187248)

[3.3.2.2 Abstract Factory Pattern Diagrams 13](#_Toc58187249)

[3.3.2.3 Source Code References 13](#_Toc58187250)

**Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.Error! Hyperlink reference not valid.**

# Architectural Representation

This document presents the architecture using logical views. The architecture is described using rational models, which explains why we chose specific decisions over other, and UML to show which class diagrams were feasible.

# Architectural Decisions

## Low Coupling / High Cohesion GRASP Decision

To reduce the impact of change throughout the system, we assigned classes with specific responsibilities following the General Responsibility Assignment Software Patterns/Principles (GRASP). The Customer class will be able to access Tickets, Meals, and Payment classes. These classes are all independent of each other but when accessed by Customer, creates high cohesion because of how they are used. Customers are able to select specific attributes from those classes to create a final payment.

### Decision to be made

The system is focused on allowing customers to buy plane tickets with varying options. The properties of good architecture decisions involve being able to control which classes interact with each other and being able to quickly generate and remove classes using polymorphism. We want to decide on a design that can apply low coupling and high cohesion. A bad decision would be to implement code and design that lets classes rely too heavily on one another. Here we can change flights and the sales will be able to handle any changes.

### Options Considered

| Low Coupling / High Cohesion | Static View | Dynamic View |
| --- | --- | --- |
| Option 1  (Rejected) |  |  |
| Option 2  (Selected) |  |  |
| Design Model Reference | Design Model Artifact Page 1 Section 1.1 Book Flight | Design Model Artifact Page 4 Section 2.1 Add Payment |

### Selection and Rationale

Option 1 has been discarded because a sale can only be captured by selecting a flight. A flight does not always have to be sold so it should not have its own sales attributes. For example, searching a flight does not mean booking and paying for a flight.

Option 2 has been selected because it allows each class accessed by Customer to be independent and allows them to work together when the Customer class requests to pay for their flight. Classes like Meal and Ticket are designed to not rely heavily on one single flight.

## Creator GRASP Decision

### Decision to be made

We want to make a decision on how various classes can be created. A good decision is a Customer being able to create a class with proper parameters. Support of polymorphism is also a benefit.. A poor decision is having the Customer create a class that does not take parameters inside and does not return the class back.

This system allows customers to create a payment. Therefore, there must be a way for the customer to open a payment with different credit cards. We decided that the Customer class can request the Payment class to open a payment and close a payment.

### Options Considered

| Creator | Static View | Dynamic View |
| --- | --- | --- |
| Option 1  (Rejected) |  |  |
| Option 2  (Selected) |  |  |
| Design Model Reference | Design Model Artifact Page 1 Section 1.1 Book Flight | Design Model Artifact Page 4 Section 2.1 Add Payment |

### Selection and Rationale

Option 1 has been discarded because a flight does not create a ticket. A ticket is created when a customer books a flight and receives a seat number and then pays for it. The Sale class should be creating tickets. We decided that tickets are based off seat class such as first class, which does not create anything, so we focused on having Payment being a creator class. createTicket() is also void, which doesn’t support something being created. There is also a lack of proper parameters.

Option 2 has been selected because when customers request to pay for a flight, the Payment class properly creates payments using factories and parameters. Payments can be opened and deleted.

## Information Expert GRASP Decision

### Decision to be made

We need to make the decision of what class can obtain what information from things that are related to it. A bad decision is retrieving too much information at a time. A good decision is instead utilizing classes and different functions to obtain that specific information needed instead of relying on one function to return everything. When designing, we decided to think of objects returning their own information. Reducing the chain of information being passed is ideal.

### Options Considered

| Information Expert | Static View | Dynamic View |
| --- | --- | --- |
| Option 1  (Rejected) |  |  |
| Option 2  (Selected) |  |  |
| Design Model Reference | Design Model Artifact Page 1 Section 1.1 Book Flight | Design Model Artifact Page 4 Section 2.3 Get Ticket |

### Selection and Rationale

Option 1 has been discarded because tickets hold information based off something that is purchased. The Ticket class does not have enough information in it for other classes to pull from and therefore does not assign responsibility well. There is a chain where Flight’s information would have to get returned to Ticket, which would get returned to Customer. It has to call two different search functions. We decided to make Ticket return different information and make flight information available in the Customer class already.

Option 2 has been selected because the Customer can easily pull specific information from the Ticket and Meal classes. Nothing is too restricted or chained together.

## Controller GRASP Decision

### Decision to be made

The system needs an assigned class with the responsibility to control which use case scenarios will be used. We need to make a decision on what first object should receive requests from the client. A good decision will consider responsibility and representation of either an entire system, object, or use case. The Session Controller coordinates the system’s operations by starting a session based on which user logs in. The Session Controller allows certain classes to work on other objects and does not do much work itself.

### Options Considered

| Controller | Static View | Dynamic View |
| --- | --- | --- |
| Option 1  (Rejected) |  |  |
| Option 2  (Selected) |  |  |
| Design Model Reference | Design Model Artifact Page 1 Section 1.2 Authenticate Customer | Design Model Artifact Page 4 Section 2.1 Add Payment |

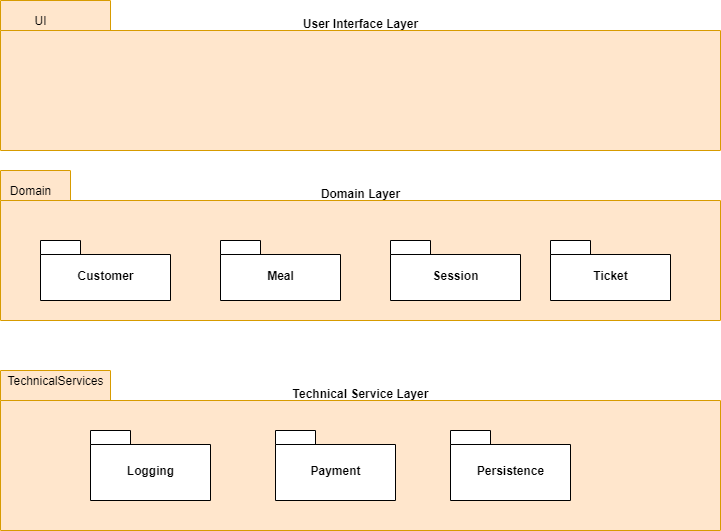
### Selection and Rationale

Option 1 has been discarded because the Session Controller would have three separate use case controllers and therefore would require unnecessary time to create a new use case if needed. With a single controller the system can have new user types using one controller.

Option 2 has been selected because a Customer will be the root object in the multiple use cases involving a Customer. Use cases such as paying with a credit card or booking a flight revolve around the Customer.

# Logical View

## Package Diagrams



### Presentation (UI) Layer Components

### Domain (Application) Layer Components

#### Customer

The Customer component consists of Customer.cpp, Customer.hpp, CustomerHandler.cpp, and CustomerHandler.hpp. This outlines the Customer class with various functions that will be used in a Customer session. The most important functions are bookFlight, payCreditCard, and showTickets. These functions are used in Customer requests to obtain responses from the System. They were all used in our technical demo. CustomerHandler contains the virtual functions. Customer will implement the CustomerHandler interface.

#### Session

Included in the Session component is Session.cpp, Session.hpp, SessionHandler.cpp, and SessionHandler.hpp. Session.cpp includes much of the setup for all sessions, such as the virtual functions getCommands() and executeCommands(). These are used by all User roles and sessions to request a list of available commands from the System and to make a request to execute one of them. These commands differ depending on the currently authenticated user’s role (as shown at the end of Session.cpp), so getCommands() is considered polymorphic.

SessionHandler.cpp confirms user’s authentication and then creates a role-specific session for that user.

#### Meal

Meal contains a MealHandler which will display the type of meal requested back to the user. It also has Meal.hpp, which utilizes different meal classes that can be purchased when booking a flight. A factory pattern and polymorphism was implemented here.

#### Ticket

Ticket has a TicketHandler.hpp which can return the type of ticket for a flight to the User. There is also Ticket.hpp, showing the different classes of tickets that Customers can purchase. More ticket types are able to be added without ripple effects, much like Meal.

### Technical Services Layer Components

#### Logging

The Logging component consists of the SimpleLogger and LoggerHandler. Its main purpose is to properly have a timer for messages detailing requests and responses between the User the system. The date and time of these logged messages will be printed first on the line. There are also messages correlating to the constructor and deconstructor that will inform the User when the Simple Logger is either initialized or shut down.

#### Persistence

Inside the Persistence component is PersistenceHandler.cpp, PersistenceHandler.hpp, SimpleDB.cpp, and SimpleDB.hpp. The Simple DB needs to be initialized alongside the Simple Logger when the program starts. Inside of SimpleDB.cpp is a vector called storedUsers which holds a user’s email login, password, and at least one authorized role. It will return the matching user if the email matches the name passed in. This is essential to authenticating the system and will not work properly if a user is not inside of this vector. The very first request when launching the program is to obtain a vector of available roles by using the findRoles() function.

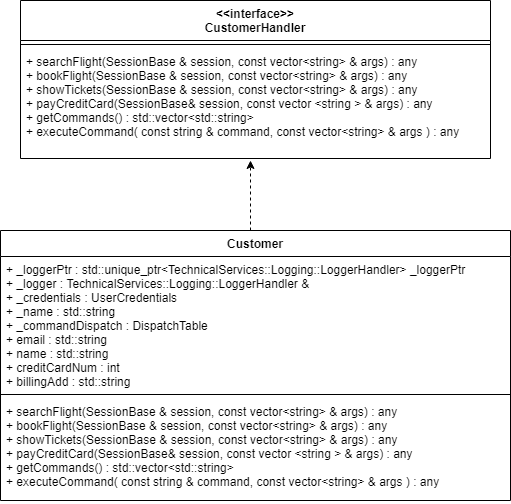
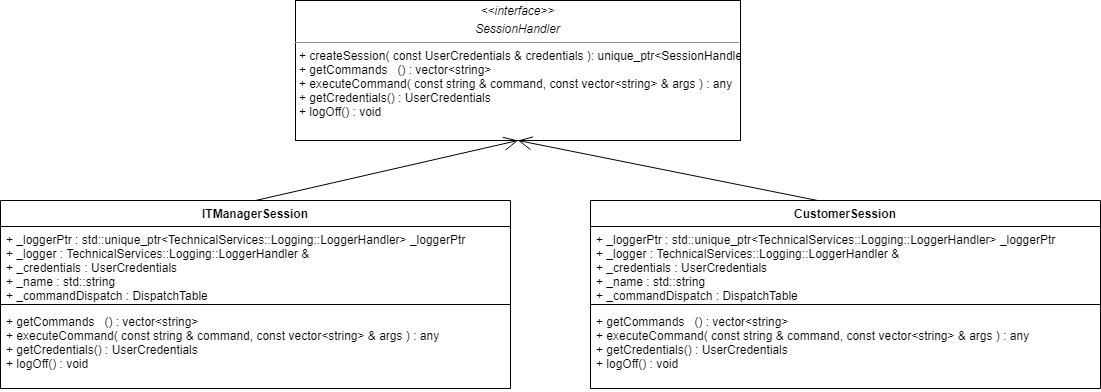
#### Payment

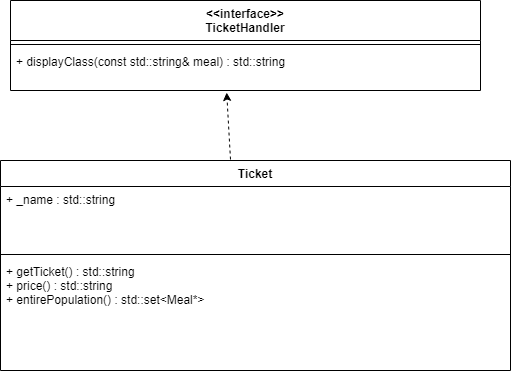
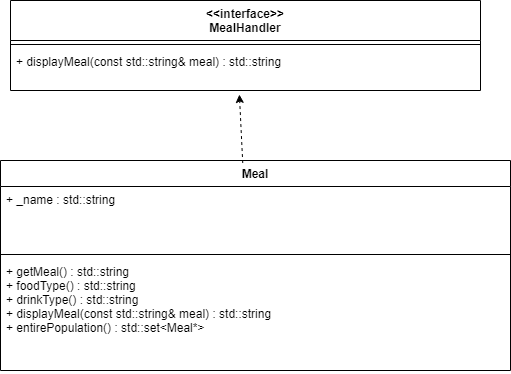
Inside the Payment component is PaymentHandler.hpp. While not complete for this iteration, we have plans to have the other interfaces interact with it to create additional payment for flights. Payment.hpp defines a Payment abstract product interface as well as additional derived classes that Customers can request. This was designed to easily accept new credit card payment methods. There are also abstract factory interfaces used for creating payment interfaces with the proper parameters.

## Interface Diagrams

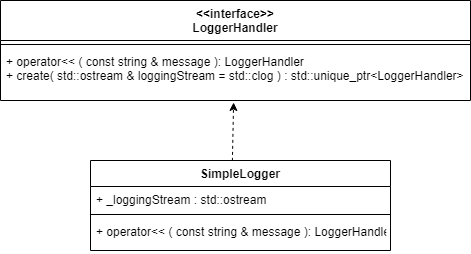
### Presentation (UI) Layer Interface Diagrams

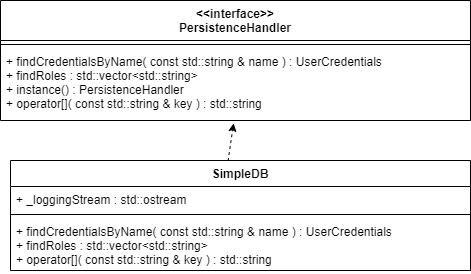
### Domain Layer Interface Diagrams

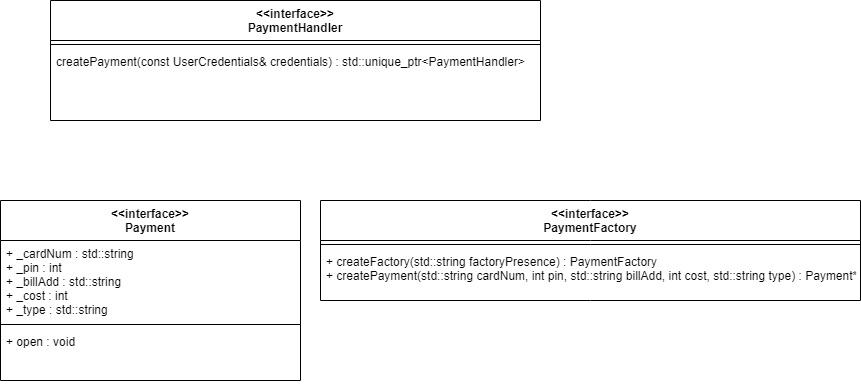




### Technical Services Interface Diagrams







## Design Patterns

### Polymorphism GRASP Pattern

#### Generalization / Specialization Diagrams

| Static View |
| --- |
|  |
| The Generalization base class Meal has two specializations, Vegetarian and Kids. Meal belongs to the Customer class who can request for the specializations. |

#### Factory Pattern Diagrams

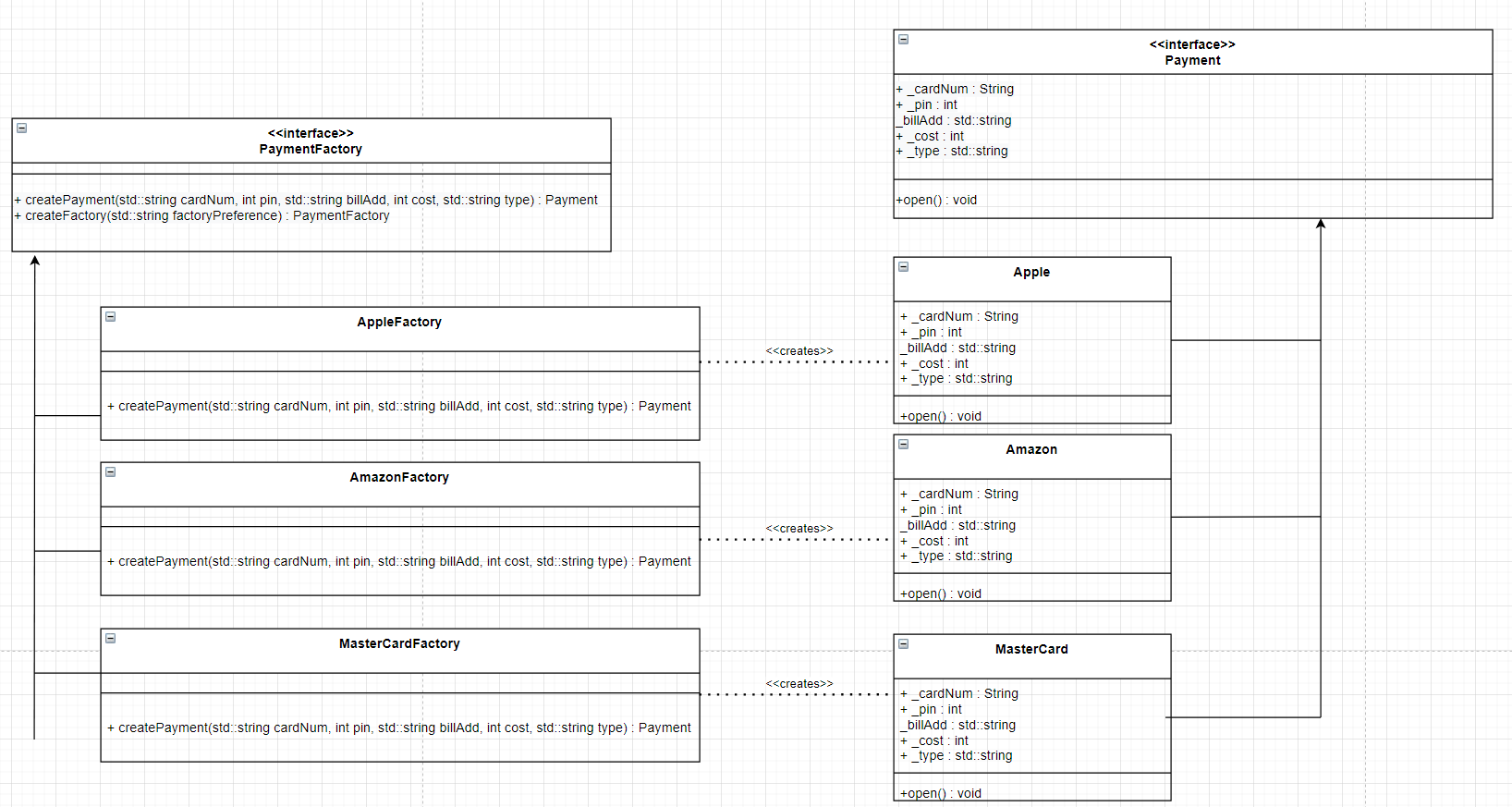
| Static View | Dynamic View |
| --- | --- |
|  |  |
| Customer is able to open a Amazon or Apple credit payment when using payWithCreditCard function | Customer opens an Amazon payment |

#### Source Code References

| Source code file name | Line number(s) |
| --- | --- |
| Payment.hpp | 1-83 |
|  |  |

### Protected Variations GRASP Pattern

#### Generalization / Specialization Diagrams



#### Abstract Factory Pattern Diagrams

| Static View | Dynamic View |
| --- | --- |
|  |  |
| PaymentFactory creates different factories depending on the type of credit card. These factories then create their own respective payment product. | Payment is calling the createPayment function to create a new Amazon factory |

#### Source Code References

| Source code file name | Line number(s) |
| --- | --- |
| Payment.hpp | 116-160 |
|  |  |