

California State University Fullerton

CPSC 462



Object Oriented Software Design SW Architecture Document (SAD) for the



High Velocity Sales Technology System

Ryan McDonald
Senior Architect
rtmcdonald96@csu.fullerton.edu

Alexander Frederick
Senior Technical Director
afrederick2@csu.fullerton.edu

Benjamin Baesu
Senior Localization SW Designer
BBaesu@csu.fullerton.edu

Revision History:

Version	Date	Summary of Changes	Author
1.0	9 November 2020	<ul style="list-style-type: none">Initial Release	Alexander Frederick Ryan McDonald
1.1	7 December 2020	<ul style="list-style-type: none">Added CreatePayment classAdded PaymentHandler classAdded VisaPayment classAdded GenericPayment classAdded section 3.3 to display the polymorphism pattern and the protected variations pattern	Alexander Frederick

Table of Contents

1	Architectural Representation	1
2	Architectural Decisions.....	2
2.1	Low Coupling / High Cohesion GRASP Decision	2
2.1.1	Decision to be made	2
2.1.2	Options Considered	3
2.1.3	Selection and Rationale	4
2.2	Creator GRASP Decision	5
2.2.1	Decision to be made	5
2.2.2	Options Considered	6
2.2.3	Selection and Rationale	8
2.3	Information Expert GRASP Decision	8
2.3.1	Decision to be made	8
2.3.2	Options Considered	9
2.3.3	Selection and Rationale	10
2.4	Controller GRASP Decision	11
2.4.1	Decision to be made	11
2.4.2	Options Considered	11
2.4.3	Selection and Rationale	13
3	Logical View.....	14
3.1	Package Diagrams.....	14
3.1.1	Presentation (UI) Layer Components	14
3.1.2	Domain (Application) Layer Components.....	14
3.1.2.1	DataSets.....	14
3.1.2.2	Session	14
3.1.2.3	Sales.....	14
3.1.3	Technical Services Layer Components.....	15
3.1.3.1	Persistence	15
3.1.3.2	Logging.....	15
3.1.3.3	Payment.....	15
3.2	Interface Diagrams	15
3.2.1	Presentation (UI) Layer Interface Diagrams	15
3.2.2	Domain Layer Interface Diagrams	15
3.2.3	Technical Services Interface Diagrams	16

3.3	Design Patterns	17
3.3.1	Polymorphism GRASP Pattern	17
3.3.1.1	Generalization / Specialization Diagrams.....	17
3.3.1.2	Factory Pattern Diagrams.....	17
3.3.1.3	Source Code References.....	18
3.3.2	Protected Variations GRASP Pattern	19
3.3.2.1	Generalization / Specialization Diagrams.....	19
3.3.2.2	Abstract Factory Pattern Diagrams	20
3.3.2.3	Source Code References.....	21

1 Architectural Representation

Architecturally we want the application to be built within specific layers and compartmentalized functionality within a proper control hierarchy. Firstly we want to properly define a Session controller that creates and controls the specific functionality of the application. It will interface with the UI layer and the objects it creates should interface with the Technical Services layer. Then we want to make sure that the classes maintain a low coupling / high cohesion format, and the compartmentalized objects are information experts within their data set domain.

The logical packages should be in 3 distinct layers: UI, Domain, and Technical Services. Within these packages there should be further compartmentalized components that pertain to specific tasks we want completed. This should reduce the amount of crossover of information and allow for more specific events to be directed to the outer layers. Only the Session controller should be interfacing with the UI, and the Technical services layer should be accessed by dependency on the Domain objects.

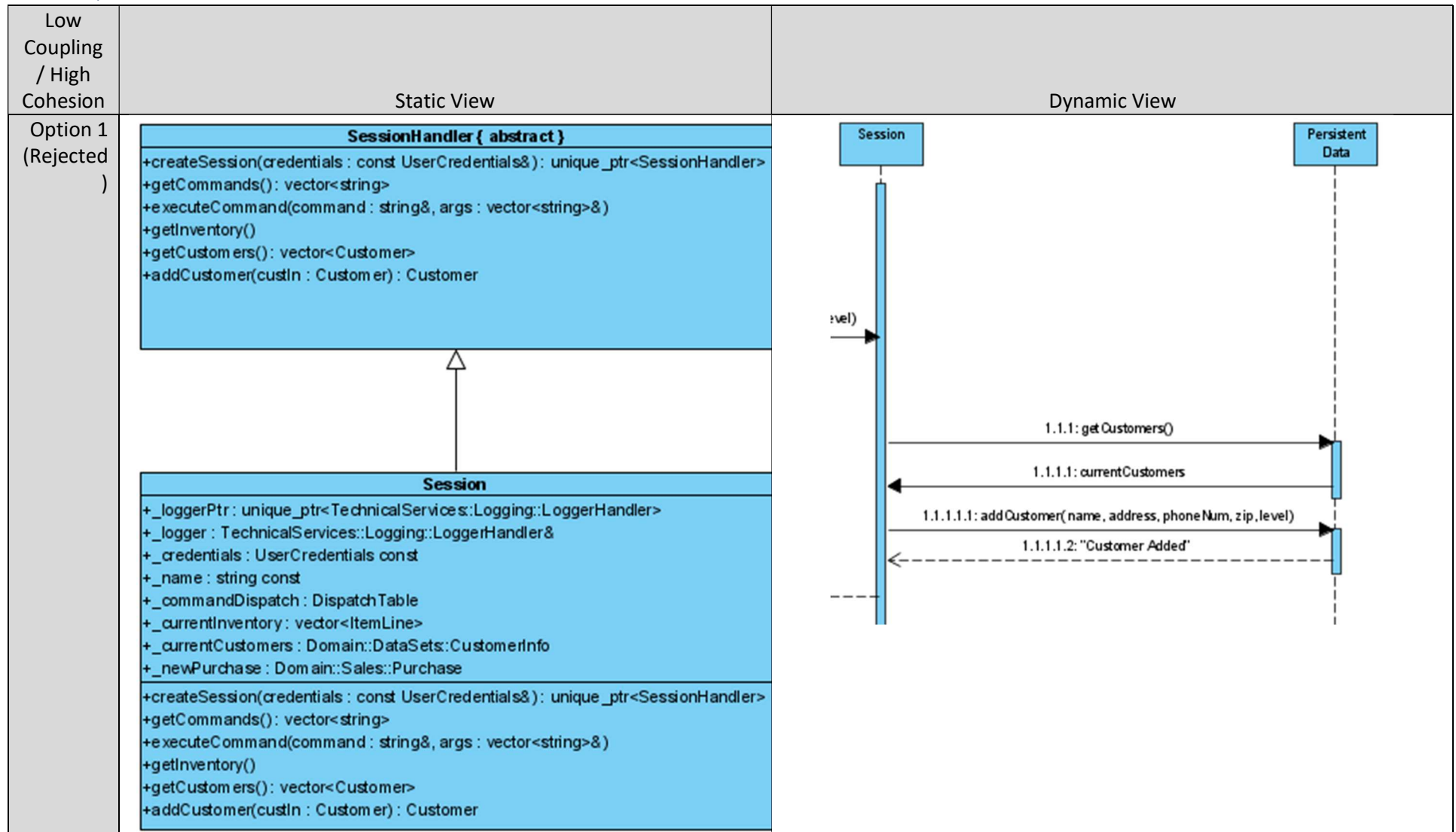
2 Architectural Decisions

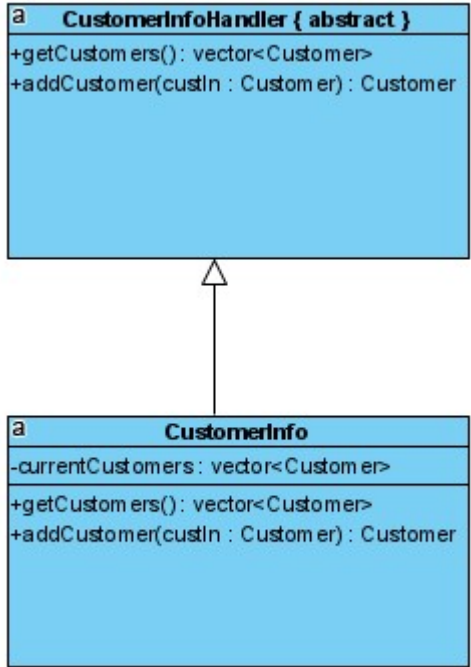
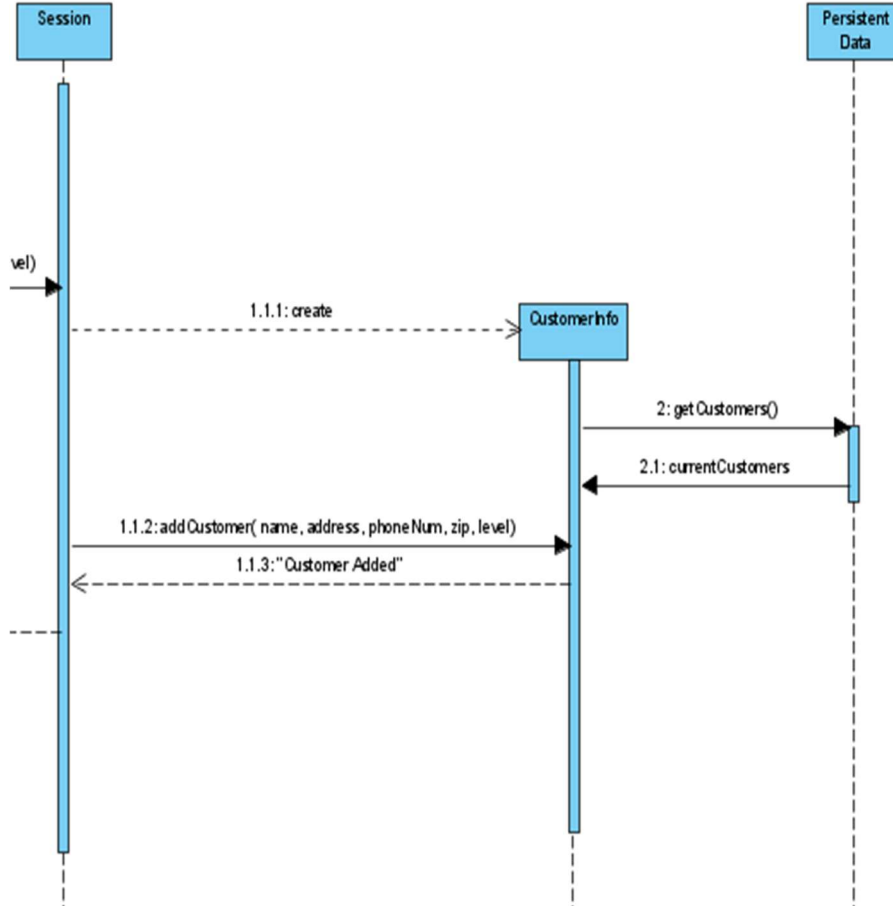
2.1 Low Coupling / High Cohesion GRASP Decision

2.1.1 Decision to be made

We want the main class that is coupled to the UI layer to have high cohesion with the classes coupled to the Technical Services layer. In order to do this we need to have our domain classes couple together to connect the layers properly and reduce the ability to connect to other layers when we don't want to allow that.

2.1.2 Options Considered



Low Coupling / High Cohesion	Static View	Dynamic View
Option 2 (Selected)	 <pre> classDiagram class CustomerInfoHandler { <<abstract>> +getCustomers() vector<Customer> +addCustomer(custIn: Customer) Customer } class CustomerInfo { -currentCustomers vector<Customer> +getCustomers() vector<Customer> +addCustomer(custIn: Customer) Customer } CustomerInfoHandler < -- CustomerInfo </pre>	 <pre> sequenceDiagram participant Session participant CustomerInfo participant PersistentData Session->>Session: vel() Session-->>CustomerInfo: 1.1.1: create activate CustomerInfo Session->>CustomerInfo: 1.1.2: addCustomer(name, address, phoneNum, zip, level) CustomerInfo-->>Session: 1.1.3: "Customer Added" deactivate CustomerInfo CustomerInfo->>PersistentData: 2: getCustomers() PersistentData-->>CustomerInfo: 2.1: currentCustomers deactivate PersistentData </pre>
Design Model Reference	Page 1, Paragraph 1.1	Page 9, Paragraph 2.3

2.1.3 Selection and Rationale

Option 1 has been discarded because we don't want the session controller to be directly interfacing to the persistence layer. In this view we could end up with an overly generalized controller that can do everything and see everything.

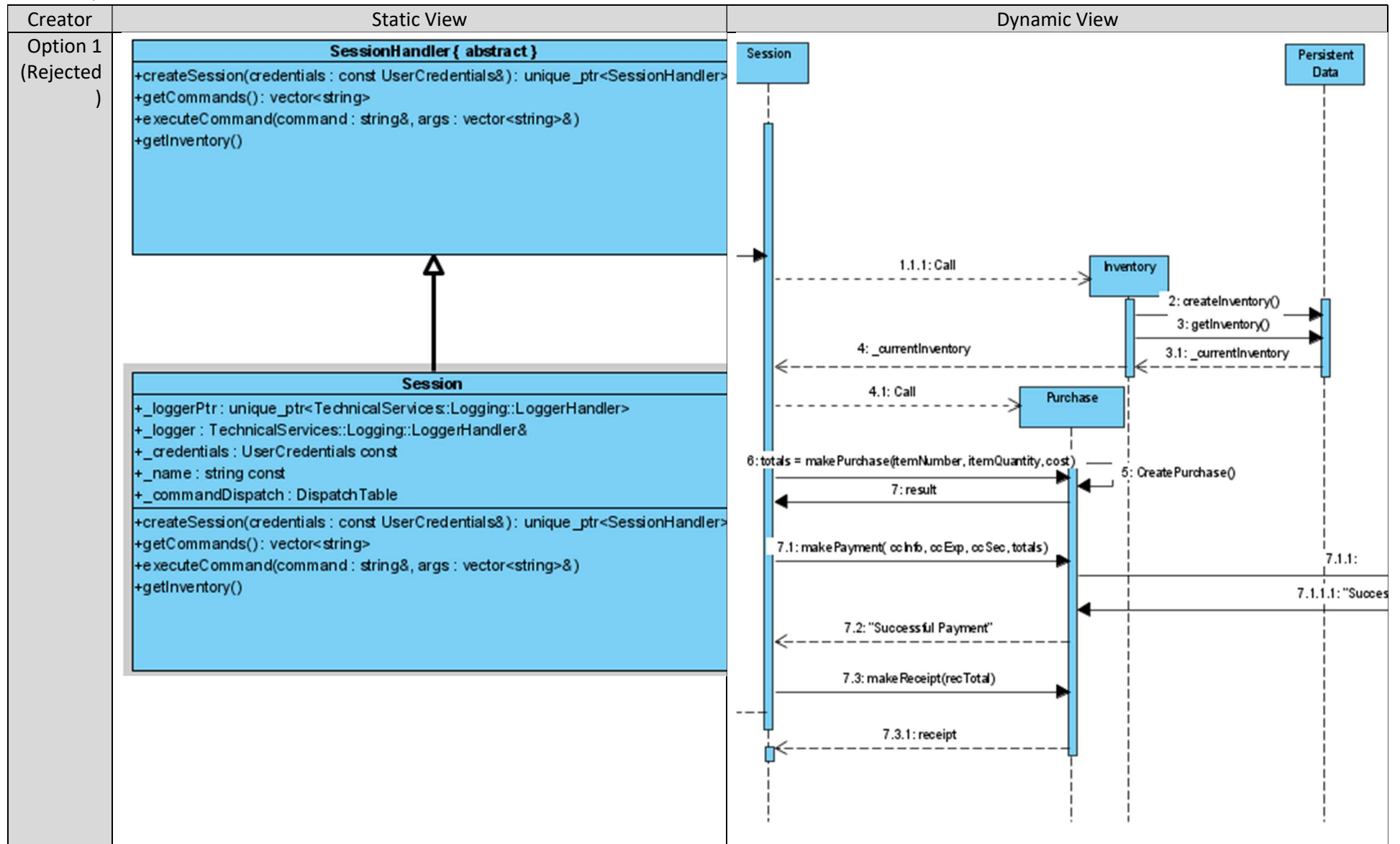
Option 2 has been selected because we want to compartmentalize information and pathing. In this effect we know that the Customer objects are specific to customer events.

2.2 Creator GRASP Decision

2.2.1 Decision to be made

We want a specific class to control the creation of other objects within the domain, because of this we need to allow one class to generate objects of the other types within our domain. We also do not want these objects to be created unless the proper role is requesting to create it. To do this we want to allow the objects to exist within the creator, but only be instantiated when the proper functionality is being called.

2.2.2 Options Considered



Creator	Static View	Dynamic View
Option 2 (Selected)	<p>SessionHandler { abstract }</p> <pre> +createSession(credentials : const UserCredentials&): unique_ptr<SessionHandler> +getCommands(): vector<string> +executeCommand(command : string&, args : vector<string>&) +getInventory() </pre> <p>Session</p> <pre> + _loggerPtr: unique_ptr<TechnicalServices::Logging::LoggerHandler> + _logger : TechnicalServices::Logging::LoggerHandler& + _credentials : UserCredentials const + _name : string const + _commandDispatch : DispatchTable + _currentInventory : vector<ItemLine> + _currentCustomers : Domain::DataSets::CustomerInfo + _newPurchase : Domain::Sales::Purchase +createSession(credentials : const UserCredentials&): unique_ptr<SessionHandler> +getCommands(): vector<string> +executeCommand(command : string&, args : vector<string>&) +getInventory() </pre> <p>SessionHandler { abstract } is the base class for Session.</p>	<pre> sequenceDiagram participant Session participant CustomerInfo Session->>Session: level() Session-->CustomerInfo: 1.1.1: create activate CustomerInfo CustomerInfo->>CustomerInfo: 2.1 deactivate CustomerInfo Session->>CustomerInfo: 1.1.2: add Customer(name, address, phone Num, zip, level) activate CustomerInfo CustomerInfo-->>Session: 1.1.3: "Customer Added" deactivate CustomerInfo </pre>
Design Model Reference	Page 4, Paragraph 1.4	Page 9, Paragraph 2.3

2.2.3 Selection and Rationale

Option 1 has been discarded because we do not want to have each class creating their overall data set object. We want the session to create an object, then use its functionality due to the specific role and event request.

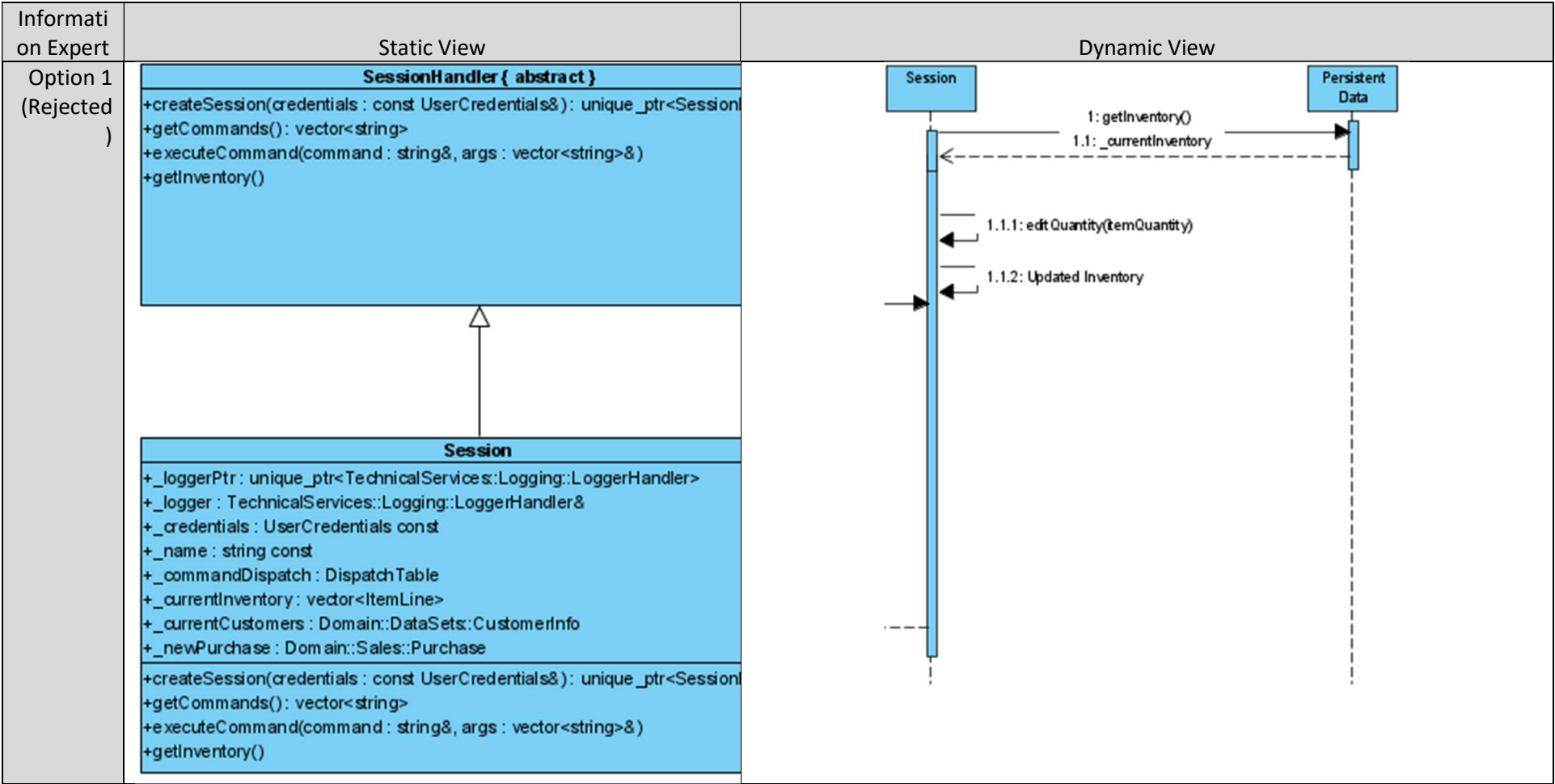
Option 2 has been selected because the session will be the creator of the objects, explicitly when we need them, and they specifically do their own functionality, rather than the class creating the instance of data for itself and allowing the session to access the data through them, instead of the data living in the session object.

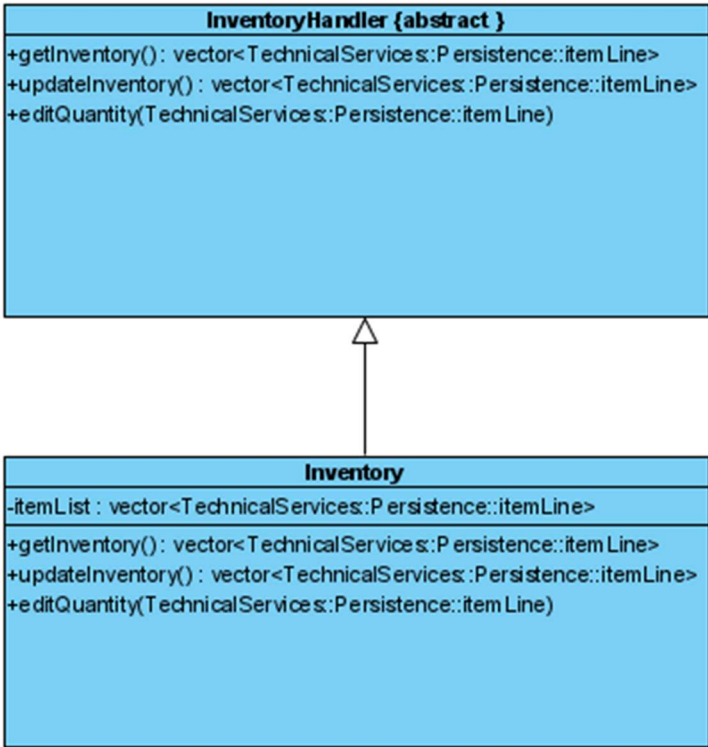
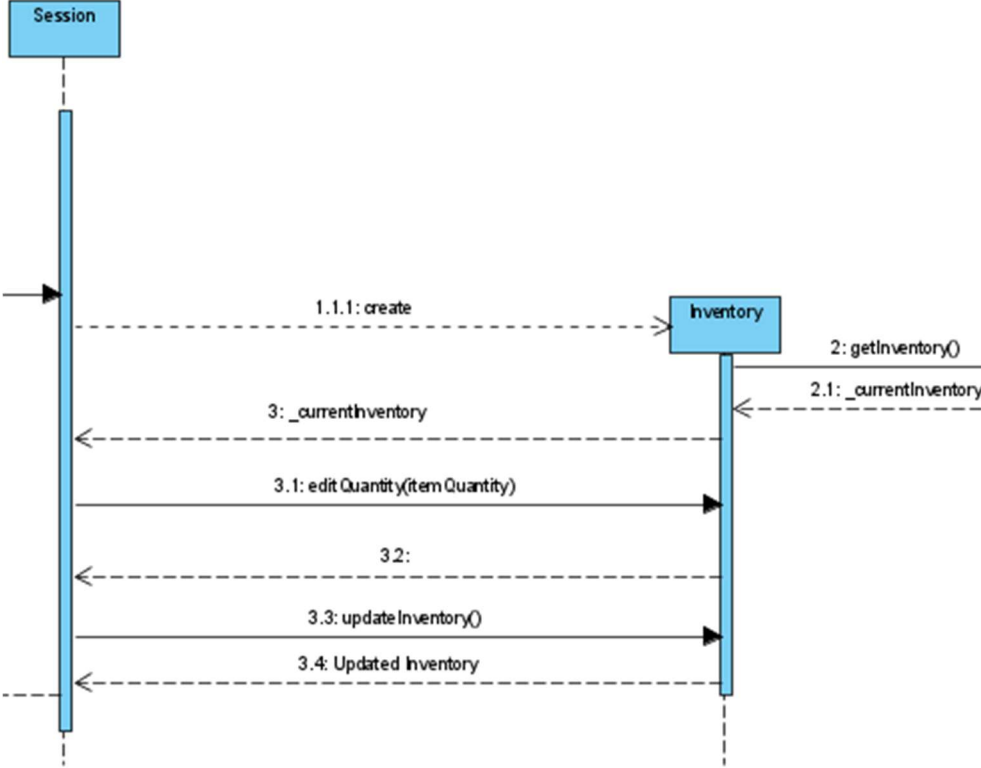
2.3 Information Expert GRASP Decision

2.3.1 Decision to be made

We want specific classes to access the information from technical services and maintain said information for use by the Domain and returned to the UI layer. While we have more than one type of information that needs to be held, we want to make sure to minimize the opportunity to allow un-needed information to be gathered when the role should not have access to it.

2.3.2 Options Considered



Information Expert	Static View	Dynamic View
Option 2 (Selected)	 <pre> classDiagram class InventoryHandler { <<abstract>> +getInventory() vector<TechnicalServices::Persistence::itemLine> +updateInventory() vector<TechnicalServices::Persistence::itemLine> +editQuantity(TechnicalServices::Persistence::itemLine) } class Inventory { -itemList vector<TechnicalServices::Persistence::itemLine> +getInventory() vector<TechnicalServices::Persistence::itemLine> +updateInventory() vector<TechnicalServices::Persistence::itemLine> +editQuantity(TechnicalServices::Persistence::itemLine) } InventoryHandler < -- Inventory </pre>	 <pre> sequenceDiagram participant Session participant Inventory Note over Session: 1.1.1: create Session-->>Inventory: activate Inventory Note over Inventory: 2: getInventory() Inventory-->>Session: 2.1: _currentInventory deactivate Inventory activate Session Note over Session: 3.1: editQuantity(itemQuantity) Session->>Inventory: activate Inventory Note over Inventory: 3.2: Inventory-->>Session: deactivate Inventory activate Session Note over Session: 3.3: updateInventory() Session->>Inventory: activate Inventory Note over Inventory: 3.4: Updated Inventory Inventory-->>Session: deactivate Inventory deactivate Session </pre>
Design Model Reference	Page 2, Paragraph 1.2	Page 10, Paragraph 2.4

2.3.3 Selection and Rationale

Option 1 has been discarded because we don't want session to directly interface the technical services layer for every single data object. This also would result in session being able to acquire data that we don't need, or want, at times when the access level should not permit this information from being gathered.

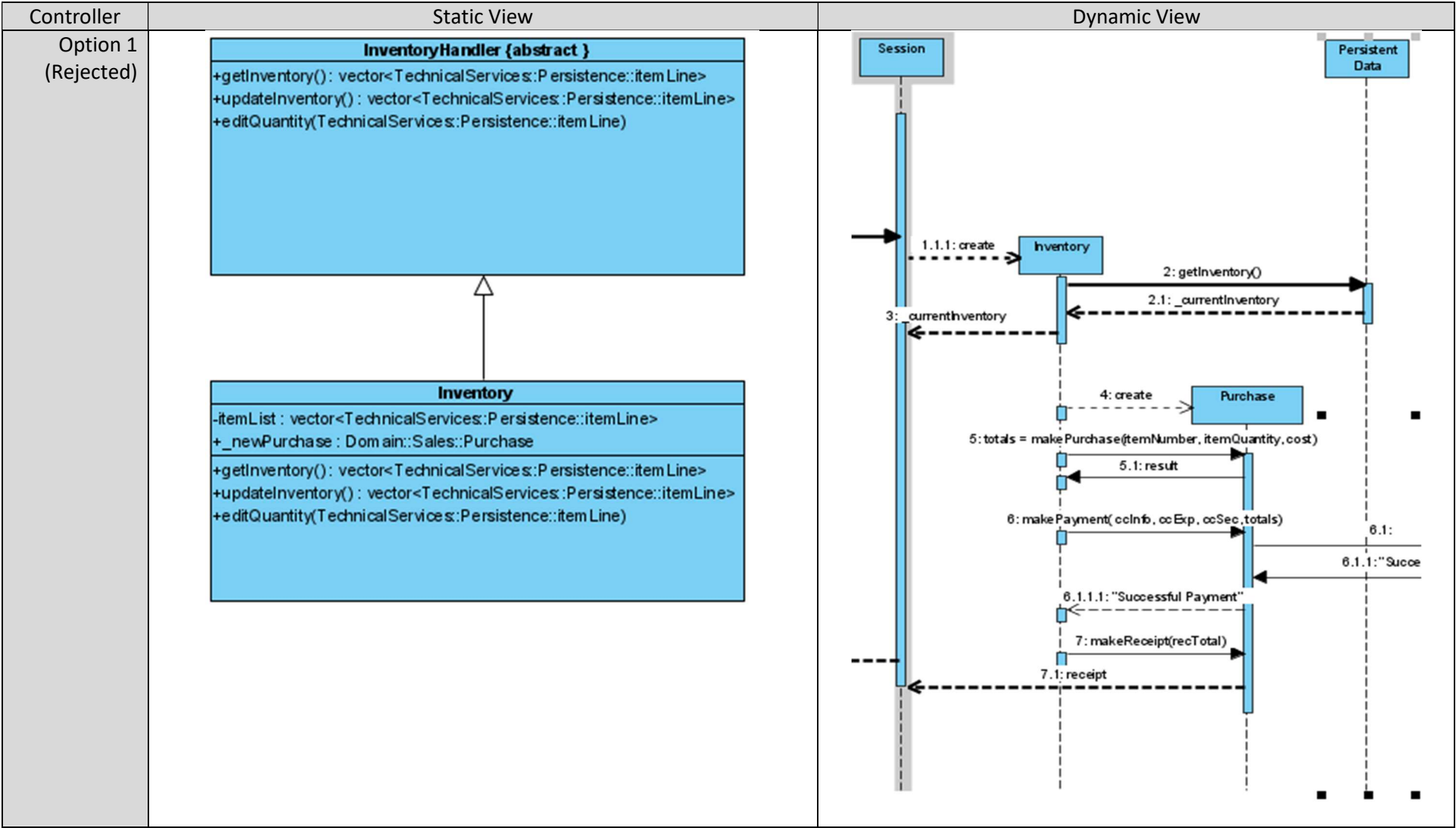
Option 2 has been selected because we allow the more compartmentalized class to be an expert on the information it requires for its events. In this case the inventory gets and adjusts the inventory objects that are gathered from the technical services interface.

2.4 Controller GRASP Decision

2.4.1 Decision to be made

We want an overall controller class to create a session based on the role of the user. This session should be flexible enough to maintain an object of each type of class, but limited enough to not allow users to access information from other roles or areas we do not need them to get access to. The session will be the first object interfaced by the UI layer and control all aspects of the domain's objects and allow them to access the underlying layers.

2.4.2 Options Considered



Controller	Static View	Dynamic View
<p>Option 2 (Selected)</p>	<pre> classDiagram class SessionHandler { <<abstract>> +createSession(credentials: const UserCredentials&): unique_ptr<SessionHandler> +getCommands(): vector<string> +executeCommand(command: string&, args: vector<string>&) +getInventory() } class Session { +_loggerPtr: unique_ptr<TechnicalServices::Logging::LoggerHandler> +_logger: TechnicalServices::Logging::LoggerHandler& +_credentials: UserCredentials const +_name: string const +_commandDispatch: DispatchTable +_currentInventory: vector<ItemLine> +_currentCustomers: Domain::DataSets::CustomerInfo +_newPurchase: Domain::Sales::Purchase +createSession(credentials: const UserCredentials&): unique_ptr<SessionHandler> +getCommands(): vector<string> +executeCommand(command: string&, args: vector<string>&) +getInventory() } SessionHandler < -- Session </pre>	<pre> sequenceDiagram participant Session participant Inventory participant Purchase Session->>Inventory: 1.1.1: create activate Inventory Inventory-->>Session: 3: _currentInventory deactivate Inventory Session->>Purchase: 3.1: create activate Purchase Purchase-->>Session: 3.2: totals = makePurchase(itemNumber, itemQuantity, cost) deactivate Purchase Session->>Purchase: 3.2.1: result activate Purchase Purchase->>Session: 3.2.1.1: makePayment(ccInfo, ccExp, ccSec, totals) deactivate Purchase Session-->>Purchase: 3.2.1.2: "Successful Payment" activate Purchase Purchase->>Session: 3.2.1.3: makeReceipt(recTotal) deactivate Purchase Session-->>Purchase: 3.2.1.4: receipt deactivate Session </pre>
Design Model Reference	Page 4, Paragraph 1.4	Page 10, Paragraph 2.4

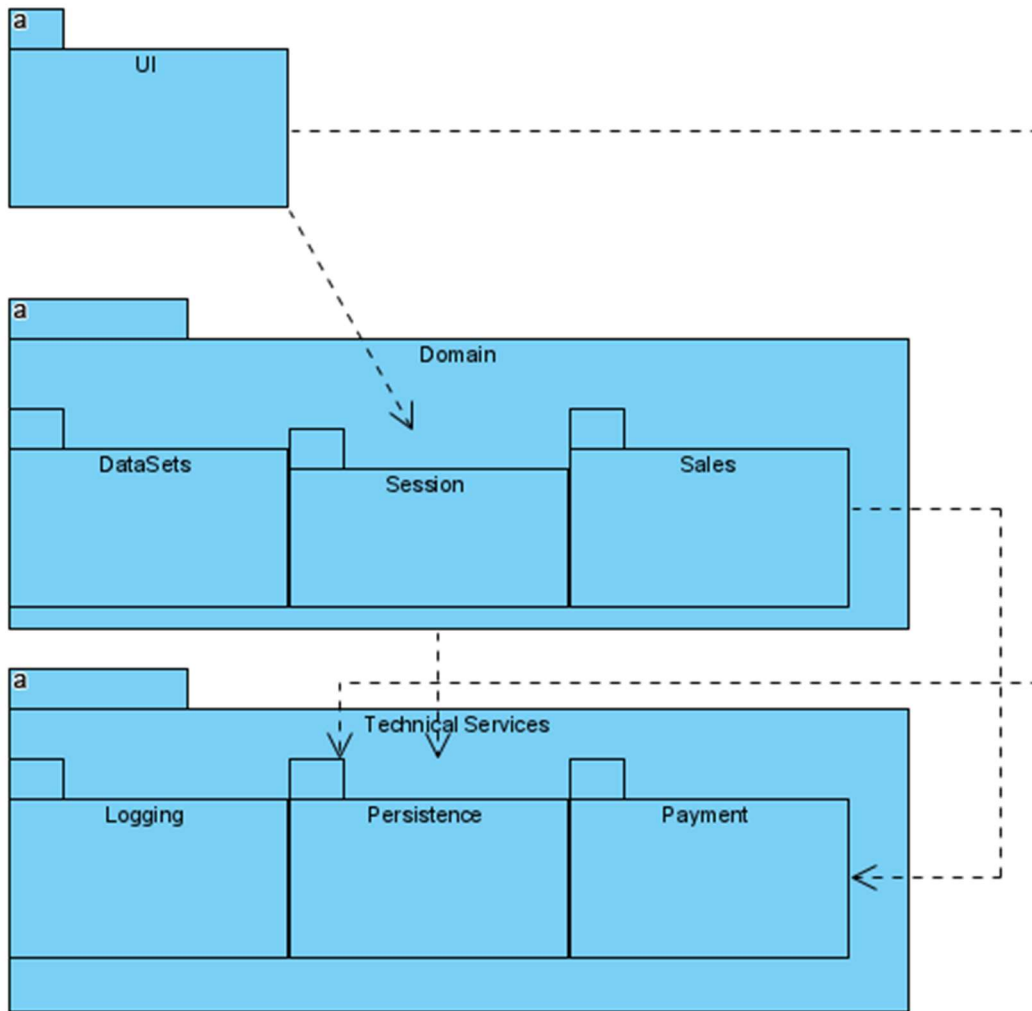
2.4.3 Selection and Rationale

Option 1 has been discarded because it allows the compartmentalized classes to control other compartmentalized groups. We want the overall control of the application to be maintained by a higher level of the hierarchy. In some cases it may make sense to allow subgroups to control another subgroup component, in this case we do not want to allow the control of the session to be given down to inventory.

Option 2 has been selected because it allows us to use previous constructs to generate new constructs, and fits within the session controller sub pattern we would prefer. We already have the inventory group in order to view and edit it in another case, so in this case we allow that event to be re-used, and then perform our new event of a purchase. This gives us the reusability of certain functions across the application when we may need them.

3 Logical View

3.1 Package Diagrams



3.1.1 Presentation (UI) Layer Components

N/A

3.1.2 Domain (Application) Layer Components

3.1.2.1 *DataSets*

DataSets is a component that is meant to control the information requested from the technical services layer and manipulate or adjust that information, as required by the session package.

3.1.2.2 *Session*

Session is the overall controller and creator component that is tasked with interfacing with the front end and connecting to the other domain instruments that interface with the technical services layer. It is meant to be our most forward facing domain object.

3.1.2.3 *Sales*

Sales component is meant to control the sales based events that are requested by the session controller package. There will be many different types of sales performed as well as a data set that control the sales information at a future iteration.

3.1.3 Technical Services Layer Components

3.1.3.1 Persistence

The persistence component is meant to be a façade of a back-end database system. Future iterations should be able to adjust where the data goes, but still request or send the same information back.

3.1.3.2 Logging

The Logging component is used to troubleshoot data paths and events that happen in the system at the domain layer. It should maintain a log of events and data transfers between components so as to be able to view events as they happened in the order they happened.

3.1.3.3 Payment

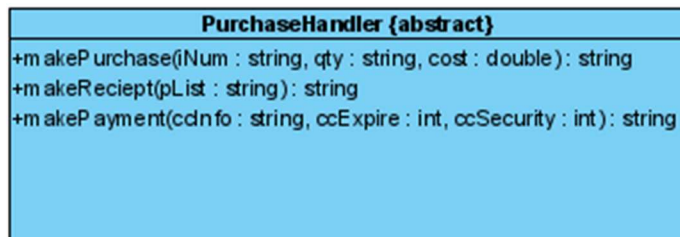
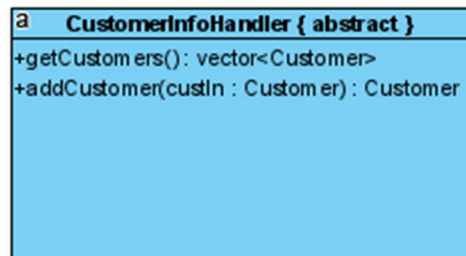
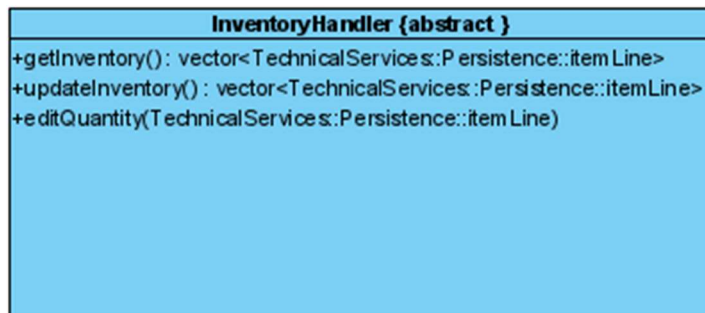
The Payment component is used as an abstract factory interface in order to better facilitate connection to external payment systems. This factory will generate specialized messages to different external payment systems. This will result in increased ability to change payment providers while the application is already deployed.

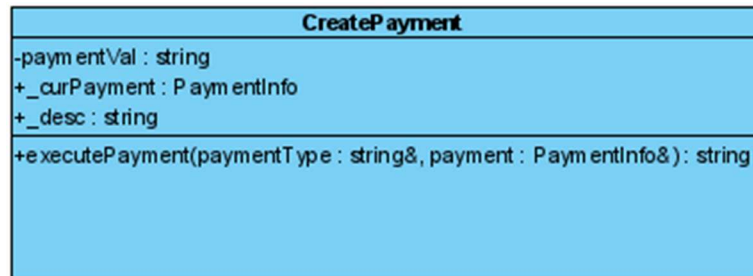
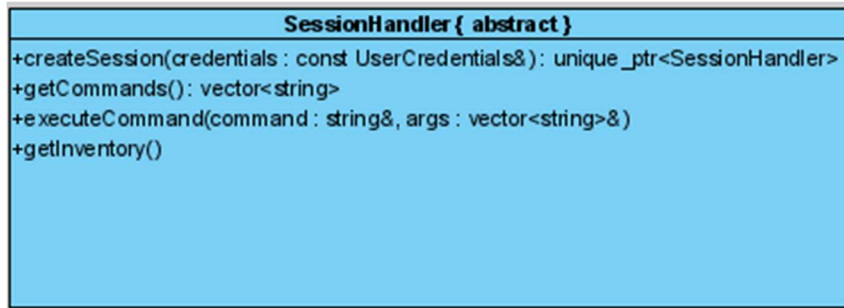
3.2 Interface Diagrams

3.2.1 Presentation (UI) Layer Interface Diagrams

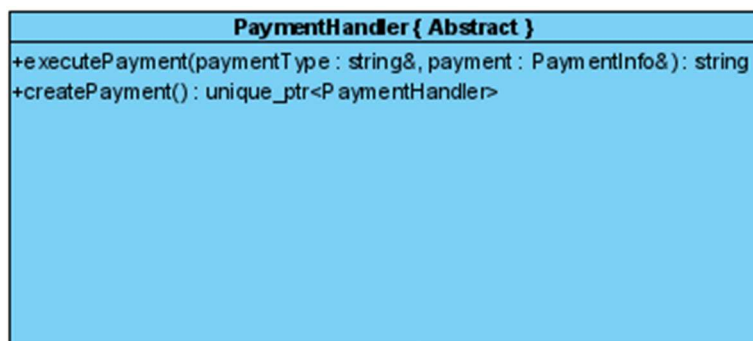
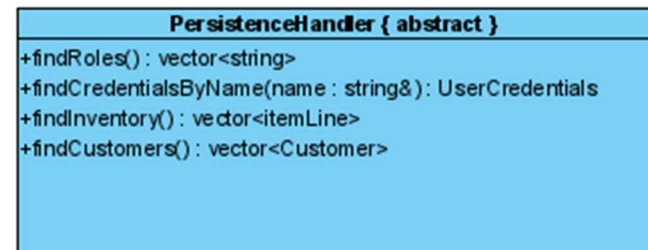
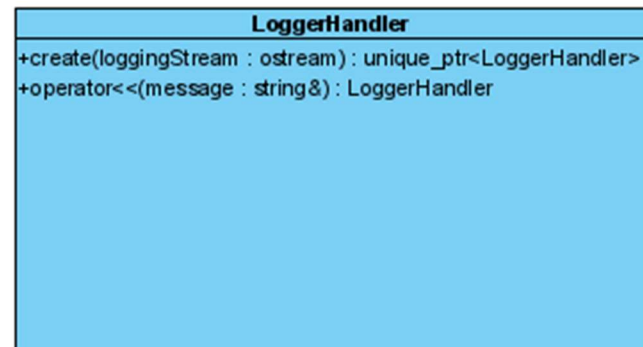
N/A

3.2.2 Domain Layer Interface Diagrams





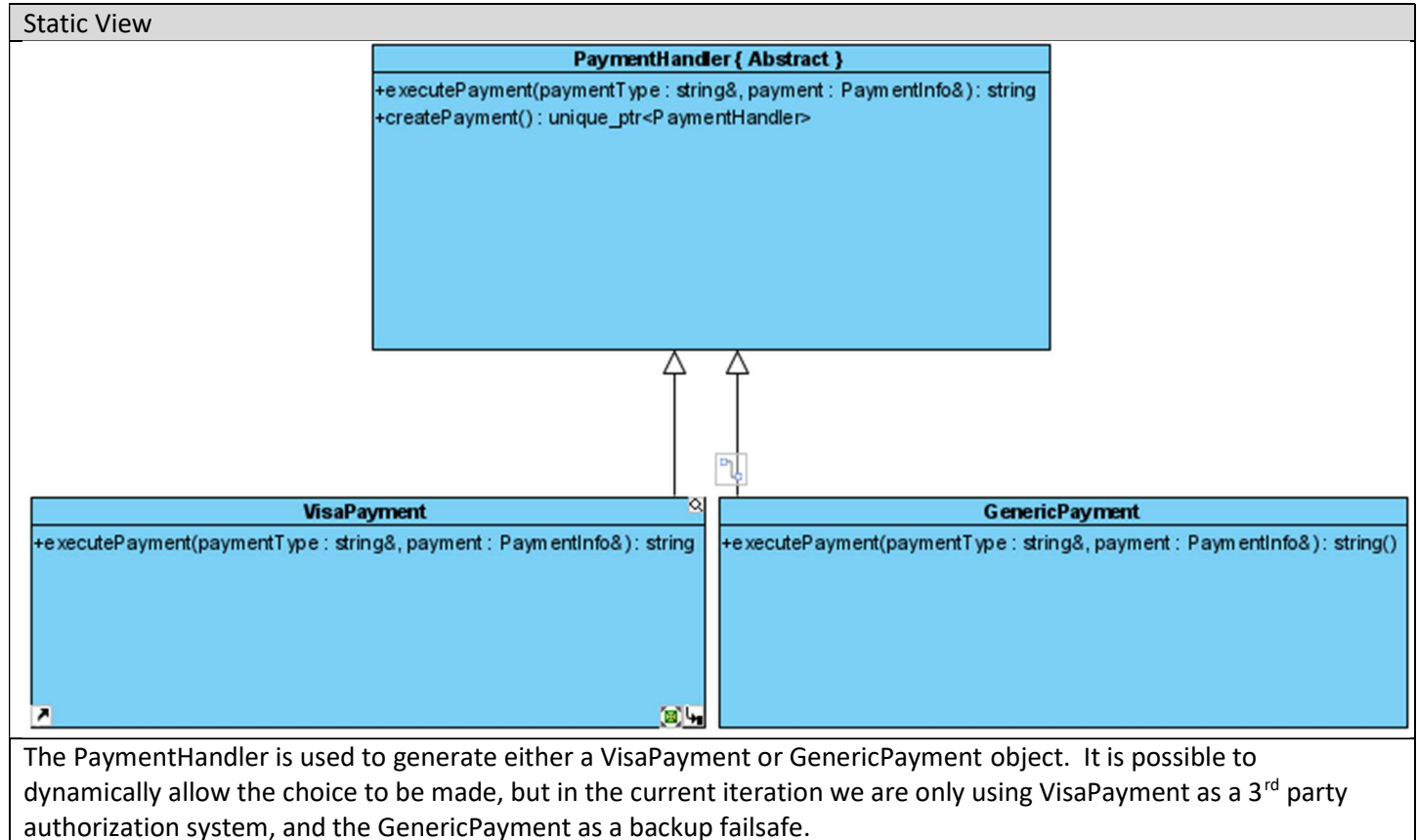
3.2.3 Technical Services Interface Diagrams



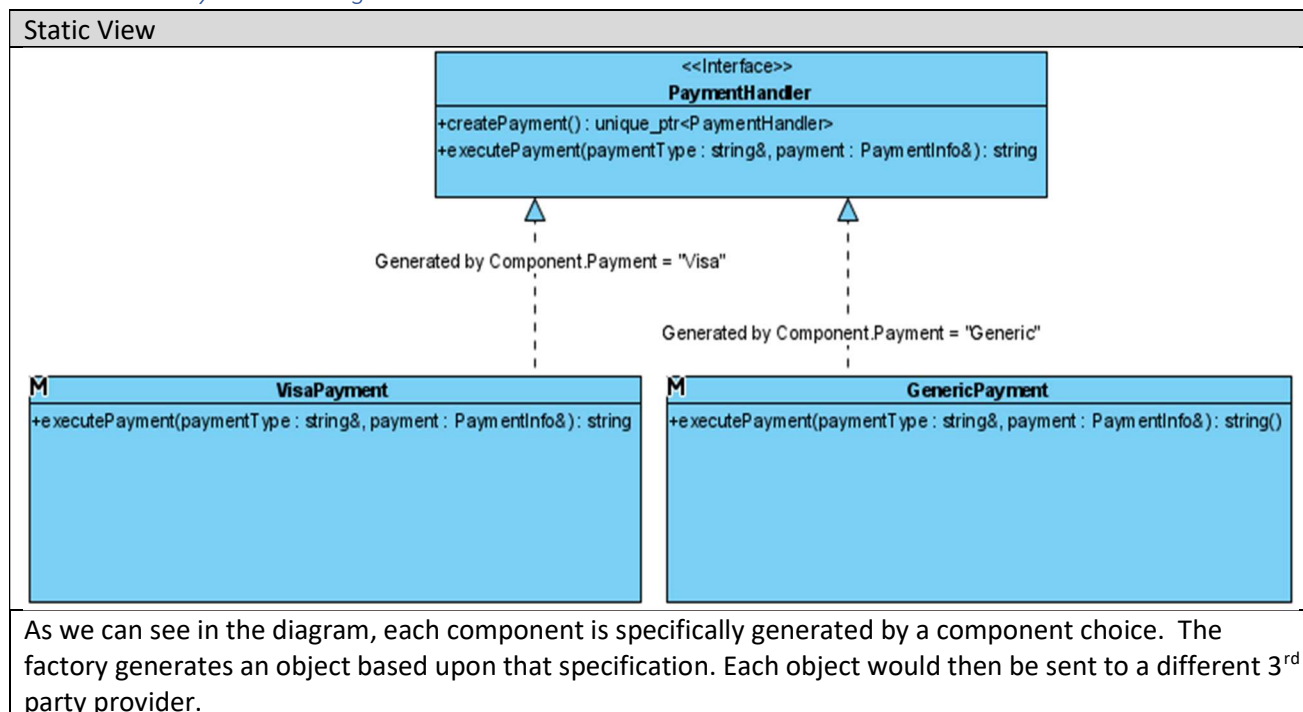
3.3 Design Patterns

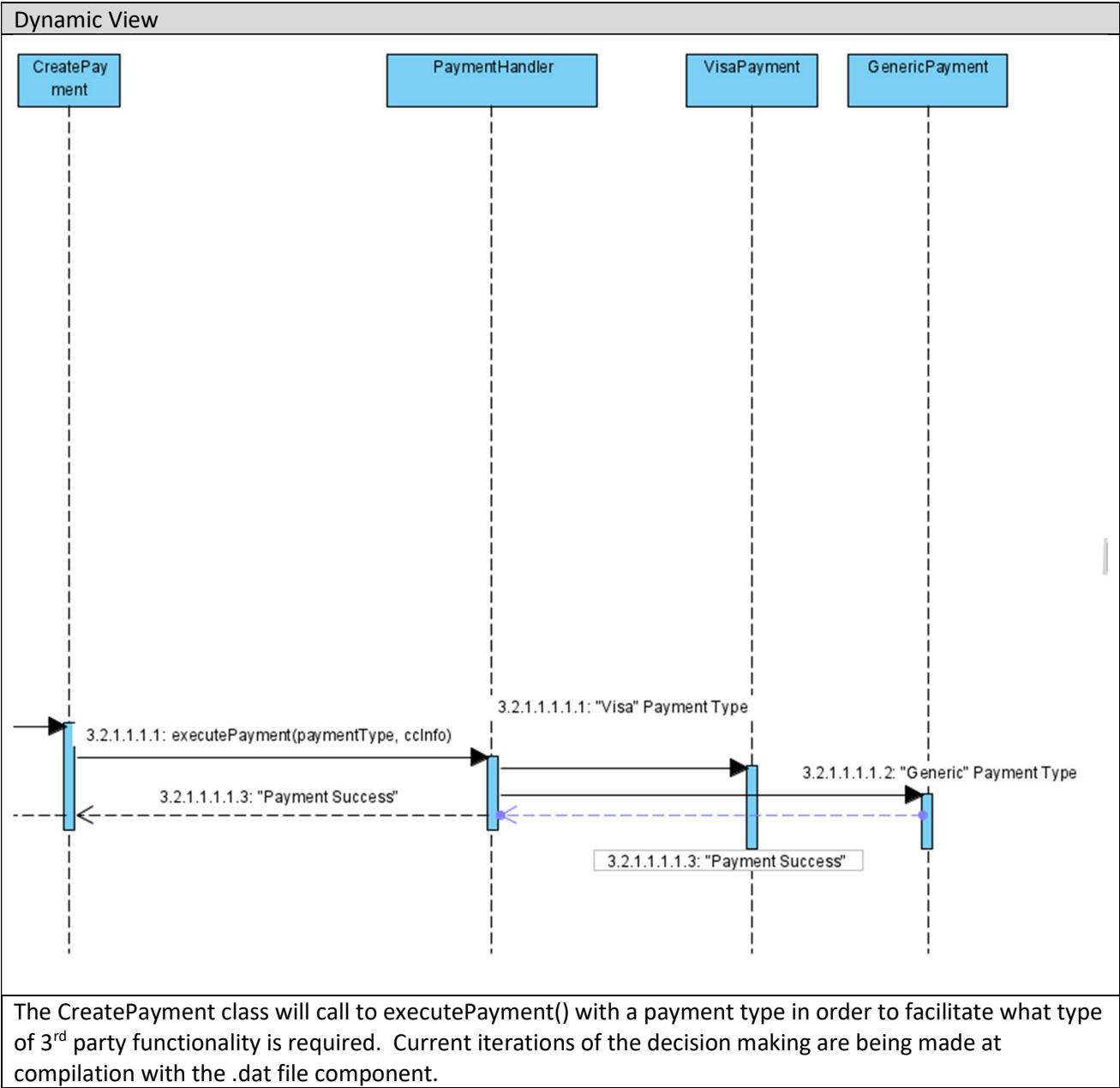
3.3.1 Polymorphism GRASP Pattern

3.3.1.1 Generalization / Specialization Diagrams



3.3.1.2 Factory Pattern Diagrams



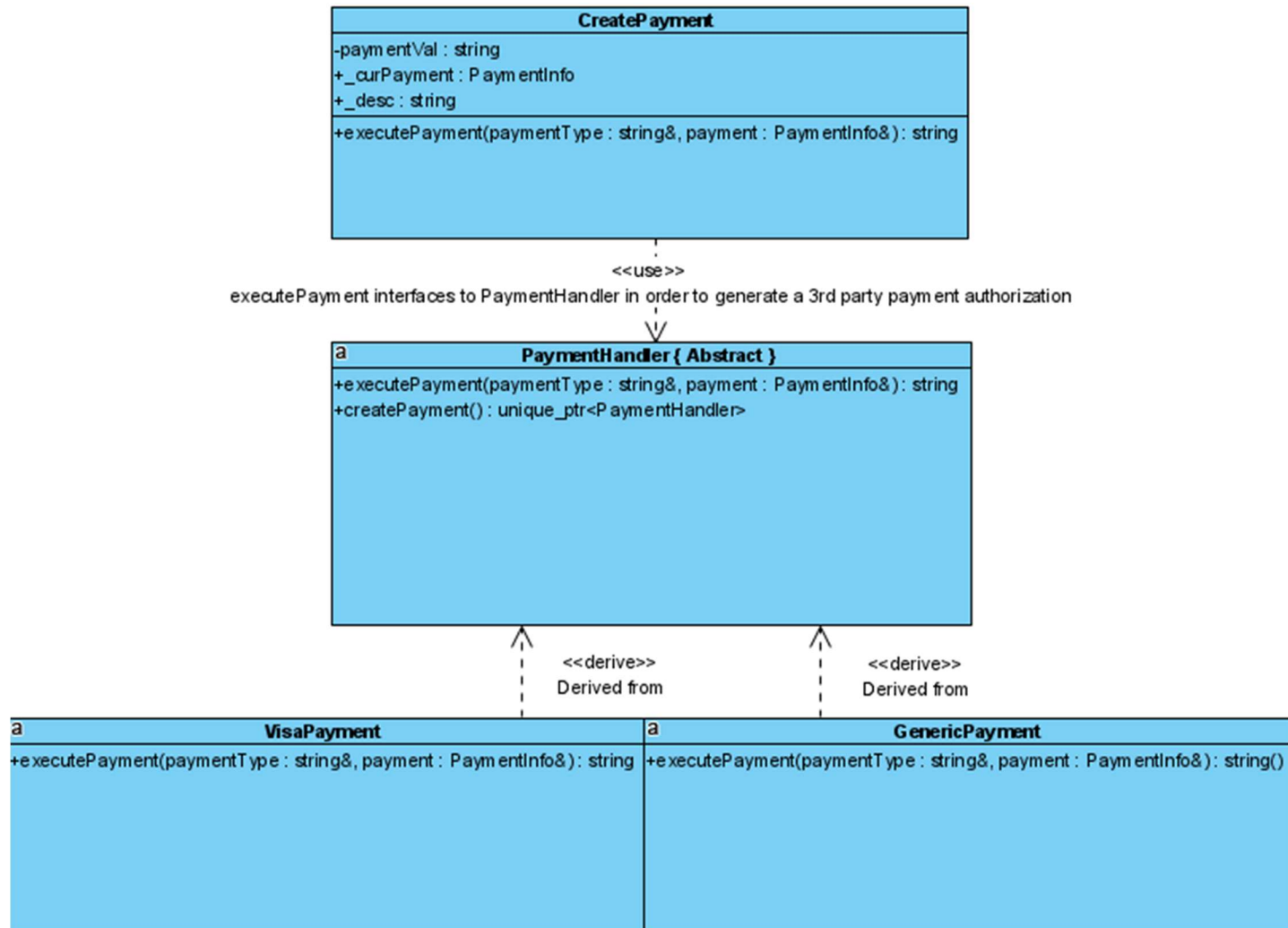


3.3.1.3 Source Code References

Source code file name	Line number(s)
PaymentHandler.cpp	11 - 23
VisaPayment.cpp	8 - 21
GenericPayment.cpp	8 - 21

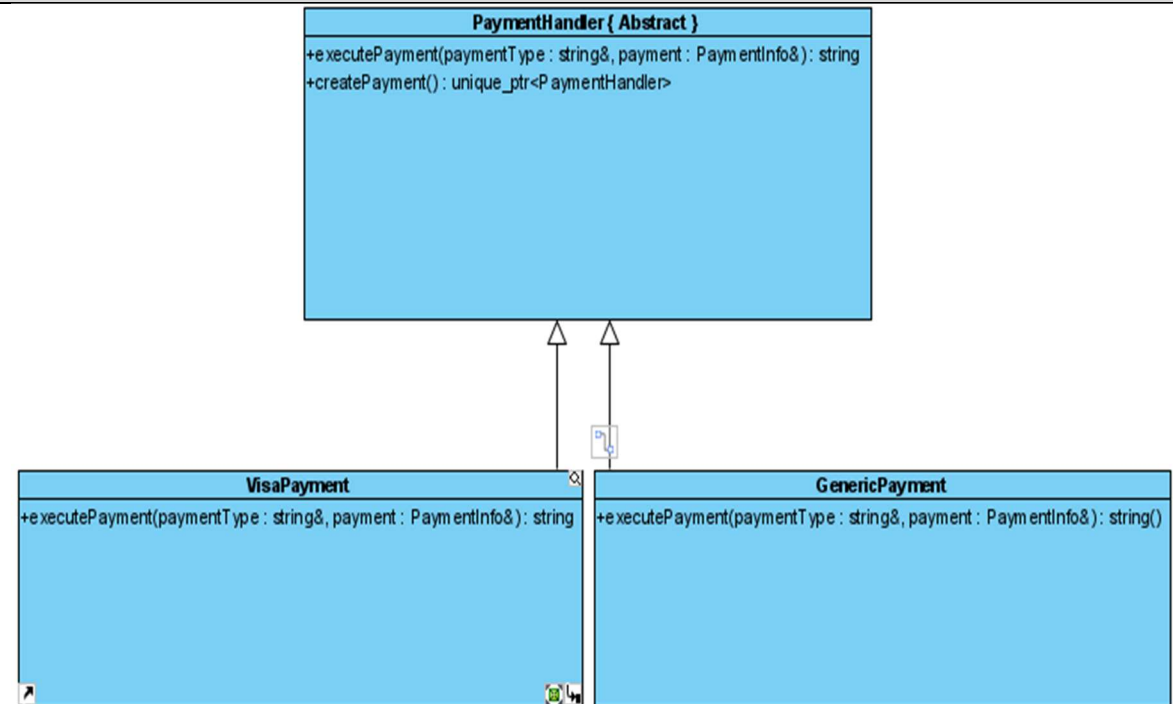
3.3.2 Protected Variations GRASP Pattern

3.3.2.1 Generalization / Specialization Diagrams

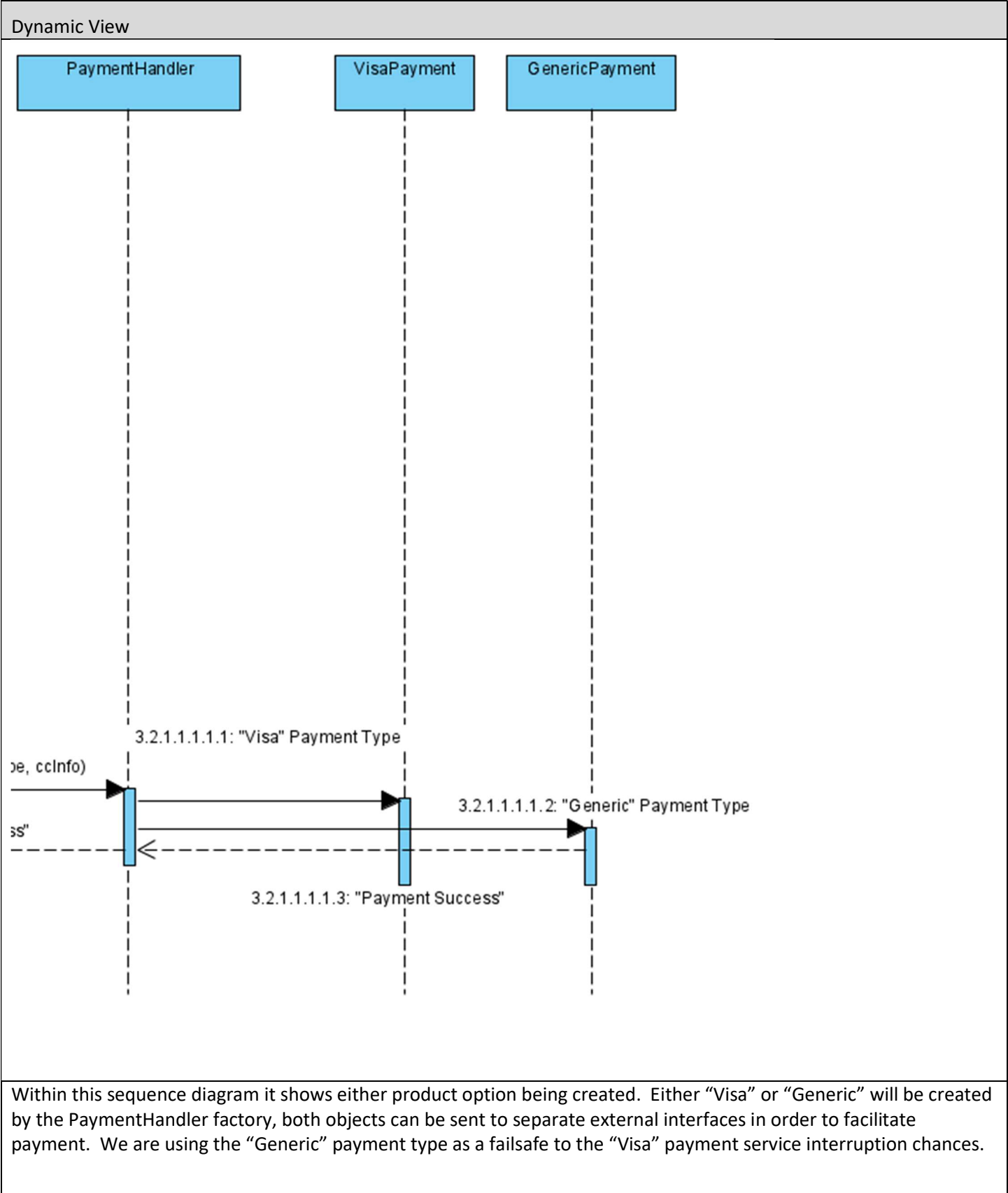


3.3.2.2 Abstract Factory Pattern Diagrams

Static View



The PaymentHandler abstraction creates either a VisaPayment or GenericPayment based on the availability of the current external payment system. It is defined internally in order to control adjustments within the scope of the domain of the application. The .dat file will define which variation is used based on the Component.Payment value



3.3.2.3 Source Code References

Source code file name	Line number(s)
PaymentHandler.cpp	11 - 23

Source code file name	Line number(s)
VisaPayment.cpp	8 - 21
GenericPayment.cpp	8 - 21