Flowershop e-commerce microservice

Olivier Dufort, Amir Ghadimmi & Lhamidi Mohamed

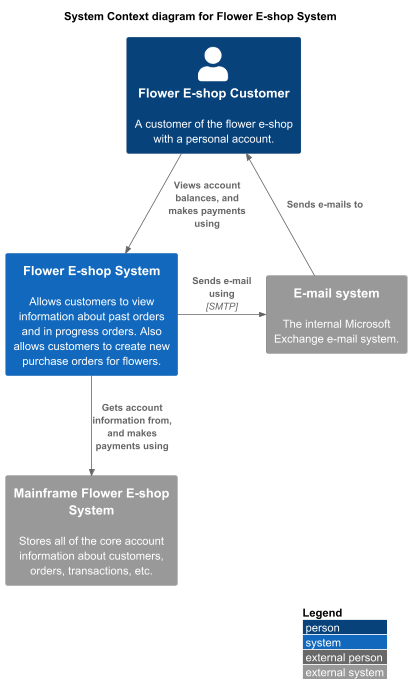
Web services and distributed computing

420-N45-LA sect. 00002

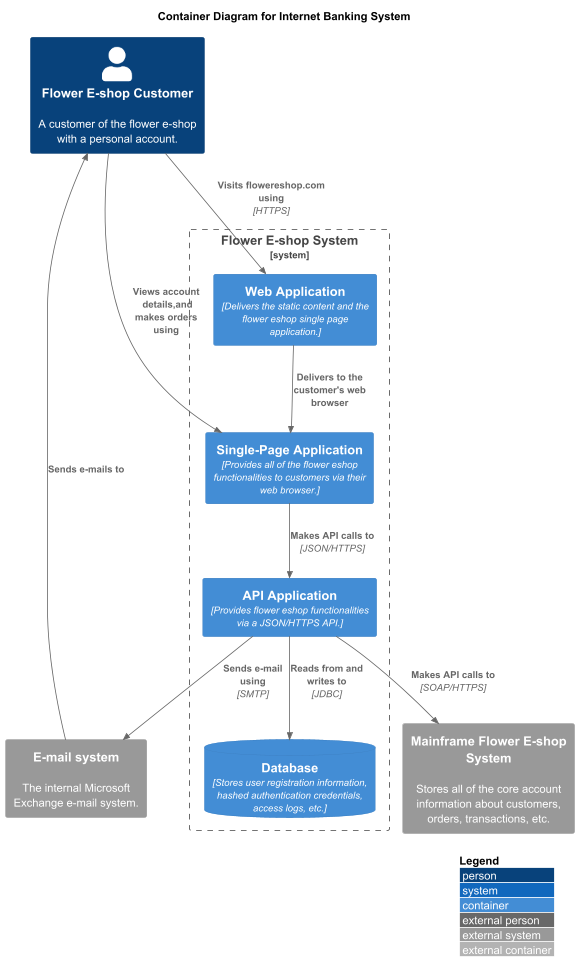
Presented to Haikel Hichri

2025-04-22

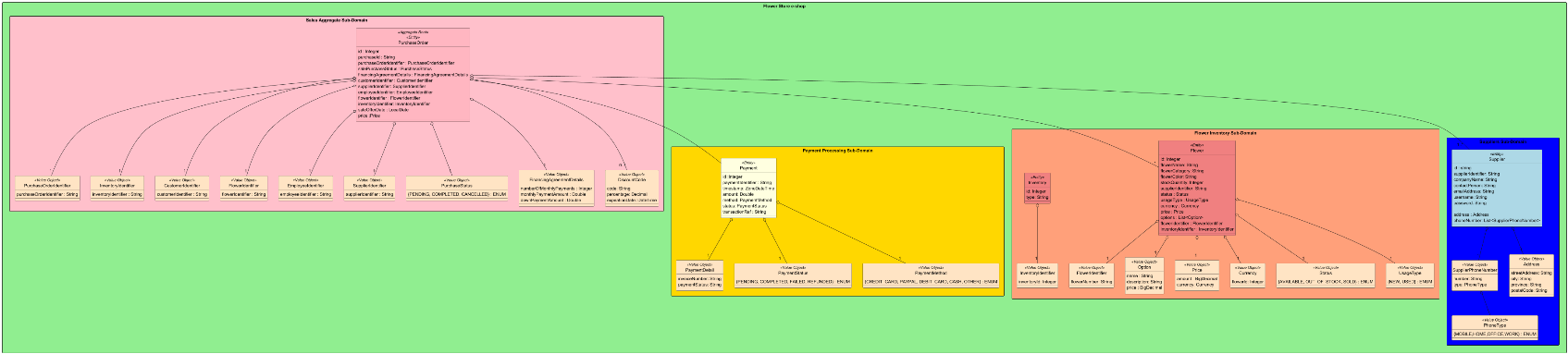
L1 diagram:



In our L1 diagram our Flower e-shop customer can interact with our Flower E-shop system, to be able to view his account, past orders, and place a new order purchase and complete the sale. The Flower e-shop system will then interact with our Mainframe (External database) Flower E-shop external system to get the desired data. Once everything is completed, our Flower E-shop system will send an email via Microsoft email providing service, which is an external service, to our client to confirm the transaction.

L2 diagram:

In our L2 diagram, the Flower E-shop customer now interacts with the Flower E-shop system, which is broken down into components. He will first access the web application using HTTP routing through our online domain flowershop.com. He will then be able to use the Single page application to complete payments, create sale orders, see supplier info, etc. The system will then communicate to the internal database to validate the user and to the external database to fetch the requested information via HTTP calls. Once the user completes a SaleOrder, the purchase confirmation receipt, and the sale transaction details are sent via the external Email system to the user.

DDD diagram:

Although it is difficult to see here, please view the image provided in our submission file to clearly see the DDD. The changes we made to the DDD diagram are that we removed classes that were not part of our Microservice application, since it seemed to have confused the teacher more than anything. Instead we focused on detailing our classes and subdoamins, adding the Payment subdomain instead of Employee, and adding all entities accordingly proper to our code architecture. We also made it “clearer” what the aggregating subdomain was to avoid confusion.

ChangeLogs:

--This is the change log, but based on option 2 given in the document. --

--Checkpoint 1 --

The prompts used in Lab1 were unfortunately deleted after the 30 days had passed, so we unfortunately do not have them to show what was done in Lab1, hence why we are using the second approach proposed in the milestone 1 document, which is to explain the changes made when integrating the classes and interfaces from Lab1 into the different microservices that needed them.

We integrated the classes and interfaces from Lab1 into our final project microservices separately. Starting With the Inventory subdomain, then the Employee subdomain, which we re-used from the cardealership monolith app in Lab1 and converted into a microservice this time, and finally the Suppliers subdomain which we also integrated. However, we realised when talking to the teacher that our aggregate subdomain was also from cardealership, so we deleted the Employee subdomain to avoid violating a constraint and instead we instead added the Payment subdomain from our teammates Lab1. The integration of all subdomains was simple, we simply changed the import names to fit our current packages, and copy-pasted our previous classes and code into the new micro-service modules. Afterwards, we added dockerfiles and docker-compose.yaml to all the microservices that did not have it, as not the entire team had completed Lab1 with Docker, so we added those individually in the microservices to be able to test each one individually to see if errors in building occurred or if database connection errors existed, as well as testing our endpoints for each precise microservice. We used the Dockerfiles and docker-compose.yaml provided in class when coding the cardealership microservice as a guide and changed them accordingly to our needs. We renamed the databases to correspond to our databases, same for each docker container name and service name, which we changed to fit our current project. We did keep the ports as shown in class, for example, 8081 : 8080, which properly routed to port 8080 externally, but on different ports internally. For now these files all use MySQL and phpMyAdmin. So we had to change one of them to use postgres sql as shown in class and in the class notes pdf.

Another change we made to our project, was the addition of our Sales aggregating subdomain, which was not requested in milestone, and is not complete. We made this choice and change to properly build everything and save some time in milestone 2 and avoid more refactoring and integration of our Lab1 into our final project. Although it is not yet fully working as requested by the teacher, it is still a good test to see if everything works well, for now, when building. This avoid going back and forth between our monolith Lab1 and this assignment in milestone 2, whereas we already needed to do it for this assignment, hence why we figured we might as well add it for now and keep it as is until milestone 2. Since not much refactoring and such was needed for milestone, we proceeded with steps 1,2 and 3 of the document. This involved creating the L1, L2 and DDD for our final proect which is a flower eshop. The prompts to get the general layout of the l1 were the following:

//An image of the Personal Banking System in the classnotes was also given to the AI

"Can you re-generate the puml code to get the following?"

//This gave an okay result but with some weird esthetic, here is the second prompt to //fix it.

"okay I mean it sort of works, but can you drop the squibly lines esthetic and keep everything in boxes similar to the image?

Try to use the information below as an example, it might not be complete, but it is what I am expected to produce, or similar.

C4 model PUML code samples

You can get C4 model templates from this Github repository: https://github.com/plantuml-stdlib/C4-PlantUML

For example, here is the template for L1-model

@startuml

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4\_Context.puml

' uncomment the following line and comment the first to use locally

' !include C4\_Context.puml

LAYOUT\_WITH\_LEGEND()

title System Context diagram for Internet Banking System

Person(customer, "Personal Banking Customer", "A customer of the bank, with personal bank accounts.")

System(banking\_system, "Internet Banking System", "Allows customers to view information about their bank

accounts, and make payments.")

System\_Ext(mail\_system, "E-mail system", "The internal Microsoft Exchange e-mail system.")

System\_Ext(mainframe, "Mainframe Banking System", "Stores all of the core banking information about customers,

accounts, transactions, etc.")

Rel(customer, banking\_system, "Uses")

Rel\_Back(customer, mail\_system, "Sends e-mails to")

Rel\_Neighbor(banking\_system, mail\_system, "Sends e-mails", "SMTP")

Rel(banking\_system, mainframe, "Uses")

@enduml

This example use the items defined in the template provided in this URL:

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4\_Context.puml

We can start with the sample provide here:

https://github.com/plantuml-stdlib/C4-PlantUML/blob/master/samples/C4\_Context%20Diagram%20Sample%20-

%20bigbankplc.puml

and modify it to describe our system, for example a car dealership system. Here is our modified code: We did not

include the customer person, because we are assuming the customers will not access the system.

@startuml

!include https://raw.githubusercontent.com/plantuml-stdlib/C4-PlantUML/master/C4\_Context.puml

' uncomment the following line and comment the first to use locally

'!include C4\_Context.puml

LAYOUT\_WITH\_LEGEND()

title System Context diagram for Car Dealership System

Person(employee, "employee","An employee of the car dealership business.")

System(cardealership\_system, "Car Dealership System","Allows customers to browse vehicle inventory. Allows

employees to manage vehicle inventory and sales.")

Rel(employee, cardealership\_system, "Tracks inentory and customer purchases using")

@enduml

Figure 6. L1 Context Diagram of the car dealership system.

Similarly, for the L2 container diagram, we can modify the example given here:

https://github.com/plantuml-stdlib/C4-PlantUML/blob/master/samples/C4\_Container%20Diagram%20Sample%20-

%20bigbankplc.puml

@startuml

!includeurl https://raw.githubusercontent.com/RicardoNiepel/C4-PlantUML/master/C4\_Container.puml

!define DEVICONS https://raw.githubusercontent.com/tupadr3/plantuml-icon-font-sprites/master/devicons

!define FONTAWESOME https://raw.githubusercontent.com/tupadr3/plantuml-icon-font-sprites/master/fontawesome-5

!include FONTAWESOME/users.puml

AddElementTag("microService", $shape=EightSidedShape(), $bgColor="CornflowerBlue", $fontColor="white",

$legendText="micro service\neight sided")

AddElementTag("storage", $shape=RoundedBoxShape(), $bgColor="lightSkyBlue", $fontColor="white")

AddRelTag(aggregate\_root, $lineColor="red")

LAYOUT\_TOP\_DOWN()

'LAYOUT\_WITH\_LEGEND()

SHOW\_PERSON\_OUTLINE()

'HIDE\_STEREOTYPE()

title Container Diagram for Car Dealership System

Person(employee, "Employee", "An employee of the car dealership.")

System\_Boundary(c1, "Car Dealership System"){

Container(web\_app, "Web Application", "JavaScript, React", "Delivers the static content and the car dealership

single page application.")

Container(singlepage\_app, "Single-Page Application", "JavaScript, React", "Provides all the Car Dealership

functionality to employeees via their web browser.")

Container(api\_gateway, "API-Gateway", "Java, Spring Boot", "Provides car dealership functionality via a

JSON/REST HTTPS API.", $tags = "microService")

Container(purchases\_service, "purchases-service", "Java, Spring Boot", "Provides purchases functionality using

the aggregator pattern via a JSON/REST HTTPS API.", $tags = "microService")

Container(inventory\_service, "inventory-service", "Java, Spring Boot", "Provides inventory functionality via a

JSON/REST HTTPS API.", $tags = "microService")

Container(employees\_service, "employees-service", "Java, Spring Boot", "Provides employee functionality via a

JSON/REST HTTPS API.", $tags = "microService")

Container(customers\_service, "customers-service", "Java, Spring Boot", "Provides customer functionality via a

JSON/REST HTTPS API.", $tags = "microService")

ContainerDb(customers\_db, "Customers DB", "MySQL 5.5.x", "Stores customer information.", $tags =

"storage")

ContainerDb(employees\_db, "Employees DB", "MySQL 5.5.x", "Stores employee information.", $tags =

"storage")

ContainerDb(inventory\_db, "Inventory DB", "MySQL 5.5.x", "Stores inventory information.", $tags = "storage")

ContainerDb(purchases\_db, "Purchases DB", "Mongo DB", "Stores purchase information.", $tags = "storage")

}

Rel(employee, web\_app, "Visits cardealership.com using", "HTTPS")

Rel\_R(web\_app, singlepage\_app, "Delivers to the customer's web browser")

Rel(employee, singlepage\_app, "views inventory and tracks purchases")

Rel(singlepage\_app, api\_gateway, "Makes API calls to", "JSON/REST/HTTPS")

Rel(api\_gateway, purchases\_service, "Makes API calls to", "JSON/REST/HTTPS")

Rel(purchases\_service, inventory\_service, "Makes API calls to", "JSON/REST/HTTPS")

Rel(purchases\_service, customers\_service, "Makes API calls to", "JSON/REST/HTTPS")

Rel(purchases\_service, employees\_service, "Makes API calls to", "JSON/REST/HTTPS")

Rel\_R(purchases\_service, purchases\_db, "Reads from and writes to", "Mongo/MQL/TCP")

Rel\_D(inventory\_service, inventory\_db, "Reads from and writes to", "JDBC/SQL/TCP")

Rel\_D(customers\_service, customers\_db, "Reads from and writes to", "JDBC/SQL/TCP")

Rel\_D(employees\_service, employees\_db, "Reads from and writes to", "JDCB/SQL/TCP")

Rel(api\_gateway, inventory\_service, "Makes API calls to", "JSON/REST/HTTPS")

Rel(api\_gateway, customers\_service, "Makes API calls to", "JSON/REST/HTTPS")

Rel(api\_gateway, employees\_service, "Makes API calls to", "JSON/REST/HTTPS")

@enduml"

With this prompt we got almost an exact match to the diagram in the classnotes, which we had difficulties with the code provided for some reason. Armed with the code, we modified all the elements to fit our system, such as changing "Personal Banking Customer" to "Flower eshop Customer", and so fourth, whilst also changing the actual variable names in the code for better code structure and readability. We also chose to add an outside system, the payment system, as we determined that we are using a payment system in our application, but it will be provided by an external client to wich we will pay a fee to use the payment system instead of having to code it ourselves and implement the security for it. This is our reasoning, similar to the usage of an exernal email system instead of creating an entirely new one from scratch. Once everything was tweaked and the diagram corresponded to our system, we sent the following prompt:

//We attached and image of the L2 diagram in the classnotes

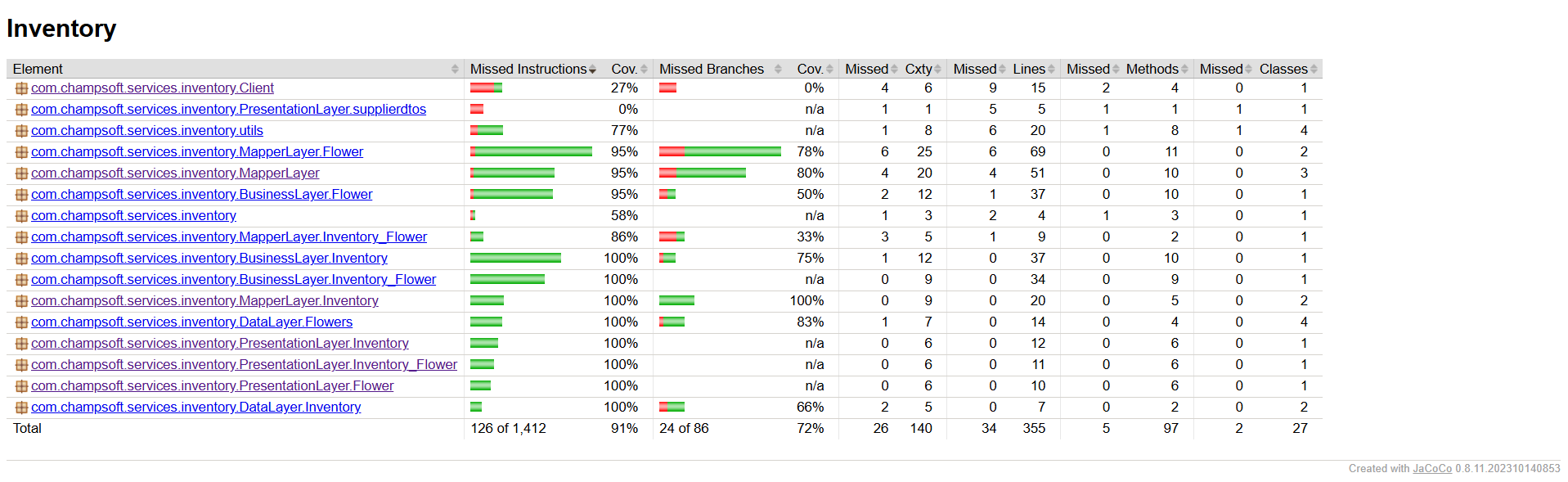
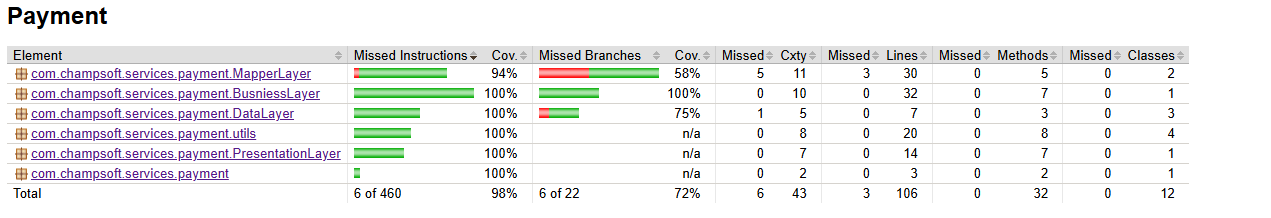
"Okay now re-generate the puml code for this L2 diagram."

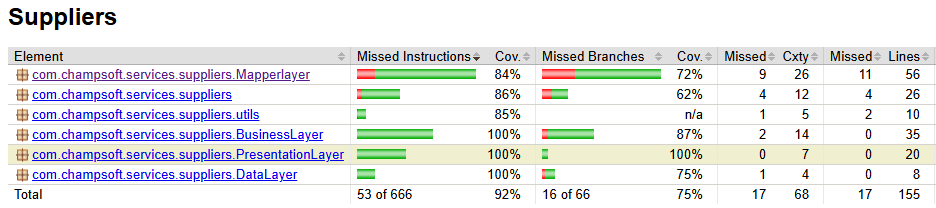
With the previous prompts already in the conversation, the AI was able to replicate the Banking System L2 diagram properly in one prompt, and it matched the classnotes almost perfectly. With this new diagram, changes similar to the L1 were made to convert the Diagram for the banking system into our flower eshop diagram. We once again added the Payment system externally, as well as removing the mobile application system, since our system is an online based system that will be served over web browser, but will not have a dedicated mobile application.

Then, we revised our DDD diagram, to better explain the bounded contexts, relationships and define the few things the teacher said needed some work, such as adding an aggregating class that is clear and visible, and tagged with "aggregator" and better defining some bounded contexts. Finally, we also changed some of the language to have a constant ubiquitous language across all our code and diagrams. Changes such as payment being changed to Sales, as we were not using a payment class in our system, and instead use an outside system that communicates with ours to process payments.However, we do have a Sales class, which is our aggregator, but it was not present in our initial DDD model and named something different, yet served the purpose of creating a sale order. To complete our DDD we removed the classes we no longer had in our project, and focused on detailing the ones we did have. Hence the changes.

Other changes we made to the project were the databases. As the milestone 1 document requested, we needed a MySQL database, a Postgres database and a no SQL database (we used mongoDB). To implement these changes, we played around in the configuration files (application.yml, Dockerfile, docker-compose.yaml), as well as our ressources to have the proper database per microservice. Finally, we created the api-gateway-service, similar to what was in the class notes and what we did in class.

Jacoco test reports:



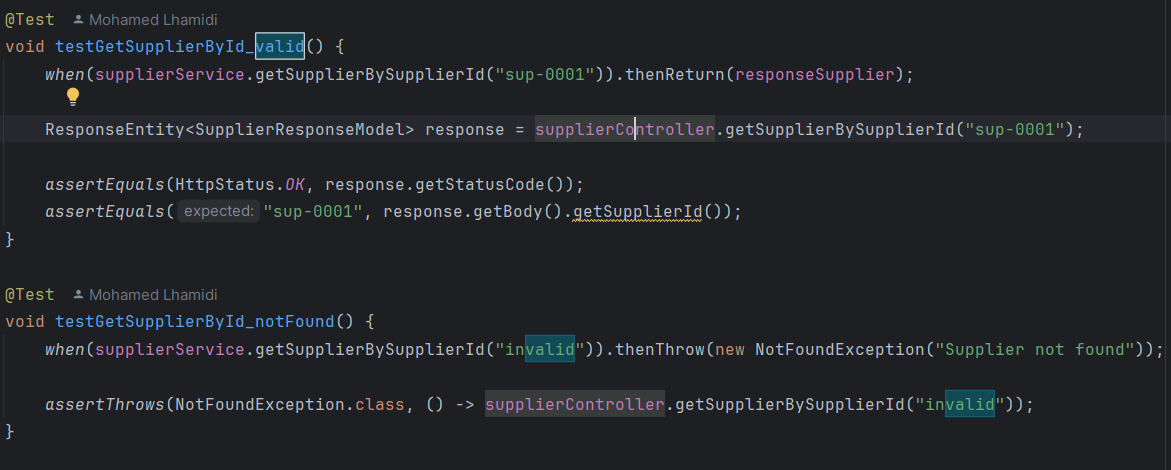


Positive and negative path tests for controller-based integration:

Inventory positive and negative path tests:



Supplier positive and negative path tests:



Payment positive and negative path tests:

