





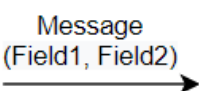
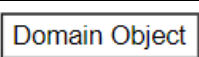



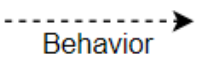

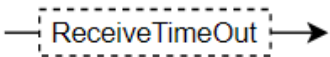
ASSIGNMENT 3: ACTOR-BASED DESIGN PATTERNS

Software Architectures

**NAME: ROBIN DE HAES
STUDENT NUMBER: 0547560**

General

Below you can find the key for all diagrams in this document.

Key	
	A service implemented as an Akka Actor.
	A “user entity”, representing an unspecified sender of the initial messages that start the “purchasing process”.
	A message with the specified fields that is sent from an Actor to another Actor.
	An instance of a domain class (not an Actor).
	A termination or Poison Pill message.
	Child Actor creation.
	A connection between a message field representing an ActorRef and the actual Actor whose address it is.
	<p><i>Between an Actor and a domain object:</i> Communication for requesting specific actions or behavior.</p> <p><i>Reflexive (from an Actor to itself):</i> Receive-behavior changes in an Actor.</p>
	Internal actions performed by an Actor (besides Receive-behavior changes).
	A ReceiveTimeout being set and being received by an Actor.
N	The order of a message or action in the whole of the diagram, e.g., N would specify the Nth message or step that is being executed.

1. Domain classes

Business logic is generally completely taken care of in methods of the domain classes.

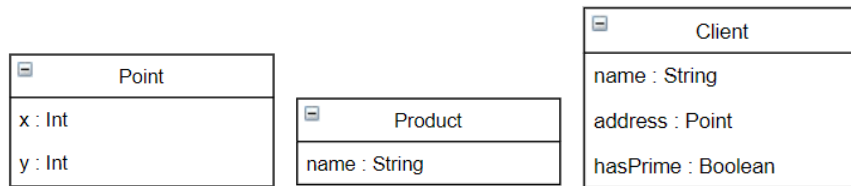


Figure 1.1. Domain classes *Point*, *Product* and *Client*

The **Point** class represents an address with Cartesian coordinates. It is mostly used for computing the distance between two points, i.e. between two addresses (from a *StockHouse* and a *Client*).

The **Product** class basically serves as a type-safe wrapper for a *String* representing a *Product* name. Names are expected to be unique, although this is not explicitly enforced during construction.

Each **Client** instance is expected to have a unique name. The Boolean field *hasPrime* is true if the *Client* is subscribed to Prime.

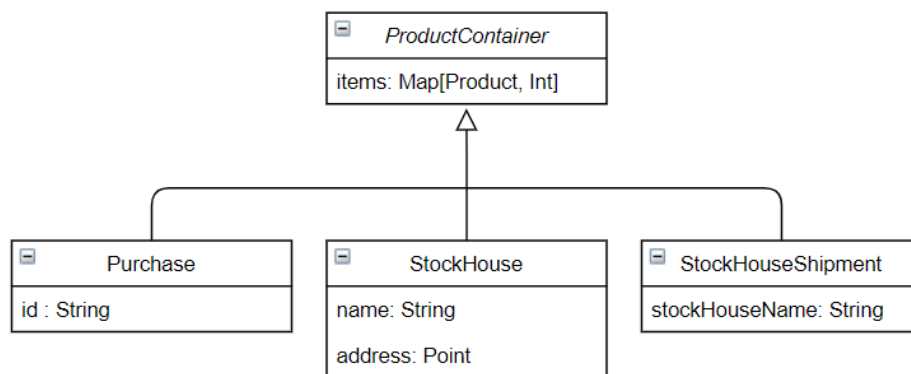


Figure 1.2. Domain class hierarchy of *ProductContainer*

An abstract class **ProductContainer** is implemented to abstract away some duplicate logic from *Purchase* and *StockHouse*. Both of these classes have a lot in common since they mainly represent and manipulate a collection of *Products*.

An abstract type *Container*, upper bounded by *ProductContainer*, is used as return type for most methods. We used a functional approach in our business logic and therefore most methods return a modified copy of the original instance of type *Container*. The extending classes concretize the *Container* type to their own class and provide their own copy-method.

A **Purchase** has a unique ID to be able to identify each *Purchase* throughout processing. An incrementing *AtomicLong* counter is used to prevent any concurrency problems while generating the ID. Since we mostly use ephemeral children and queues, having unique *Purchase* IDs is currently somewhat redundant but they are still used and sent for easier extensibility in the future.

A **StockHouse** is expected to have a unique name. This class adds methods for computing the distance and (un)reserving items.

A **StockHouseShipment** is a helper class used in messages that wraps together the name of a *StockHouse* and items that will be shipped from that *StockHouse*. It has mainly been added for testing purposes to be able to identify which *StockHouses* will send which stock to a *Client* to fulfill a *Purchase*.

2. Services

All main services have a companion object with props and name methods to easily create new instances of that service. Furthermore, they mainly deal with communication logic and request domain objects to take care of the business logic.

ClientService

A ClientService is made for every client.

Each ClientService knows the address of the ProcessingService and has 2 Receive behaviors.

The initial Receive behavior represents the behavior of a “shopping cart” with a Purchase as its contents. The Purchase can be updated until a MakePurchase message is received at which point the Purchase is sent to a ProcessingService for processing and the ClientService’s behavior changes to waitForOrder.

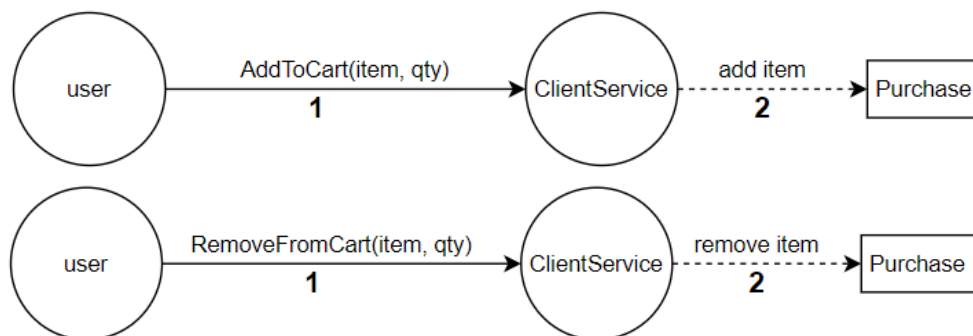


Figure 2.1. Adding/Removing items from the client’s “shopping cart”.

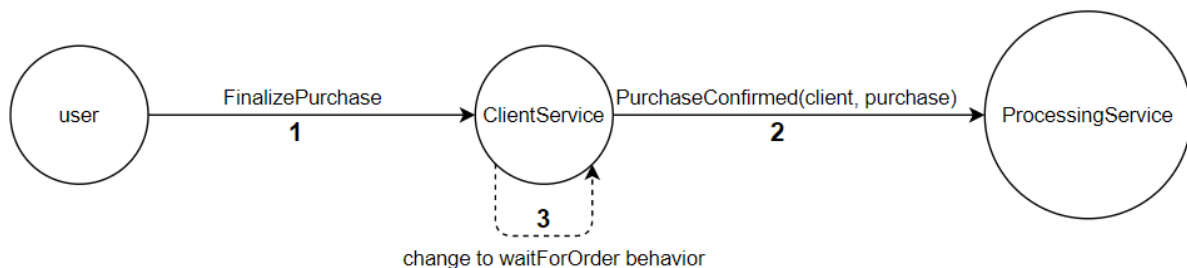


Figure 2.2. Confirming the “shopping cart” to make the actual purchase.

The waitForOrder behavior waits for a confirmation message from the ProcessingService. If OrderShipped is received, shipment information is logged via ActorLogging and the behavior changes back to an empty “shopping cart” so new Purchases can be made. If OrderDelayed is received, this is also logged but the behavior changes back to a “shopping cart” with your previous Purchase so you can easily try again.

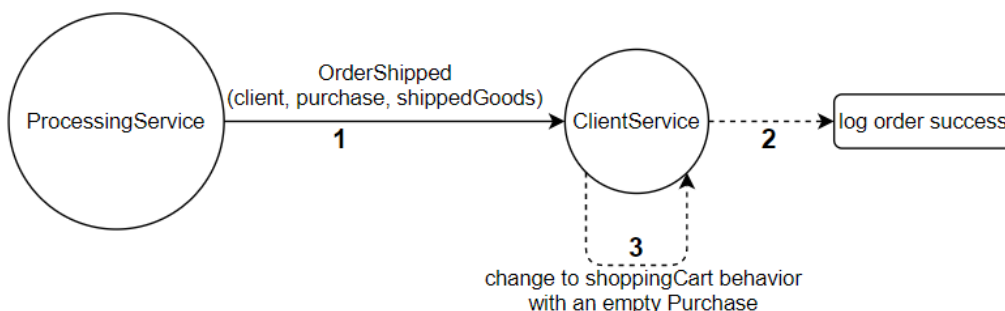


Figure 2.3. Receiving an OrderShipped confirmation after the purchase was successfully processed.

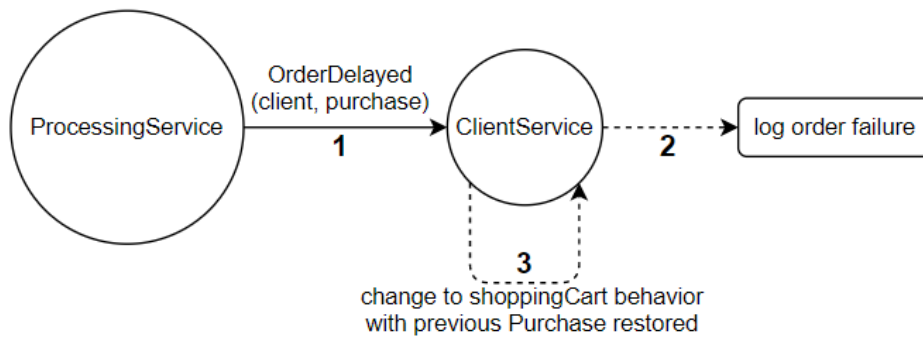


Figure 2.4. Receiving an OrderDelayed confirmation when the purchase could not be successfully processed.

StockHouseService

Although business logic and communication logic are separated in the other services as well, the StockHouseService is the clearest example of the **Domain Object Pattern**. This service manages a reference to the current state of a StockHouse. It receives Manager Messages that contain a Domain Message for modifying or querying the StockHouse (and it will possibly also reply with a Manager Message). The StockHouse it manages performs the business logic required for producing the answer. In addition to the actual Domain Message, the Manager Messages also contain an ID and a replyTo reference for message deduplication.

In our case, the ID of the Manager Messages is the Purchase ID and replyTo is the address of an ephemeral aggregator from the ProcessingService.

There is one ManagerQuery supported, which is sent by the ProcessingService to all StockHouseServices to compute the distance to a Client. The ManagerResults are then aggregated by an ephemeral NearestNeighborsAggregator.

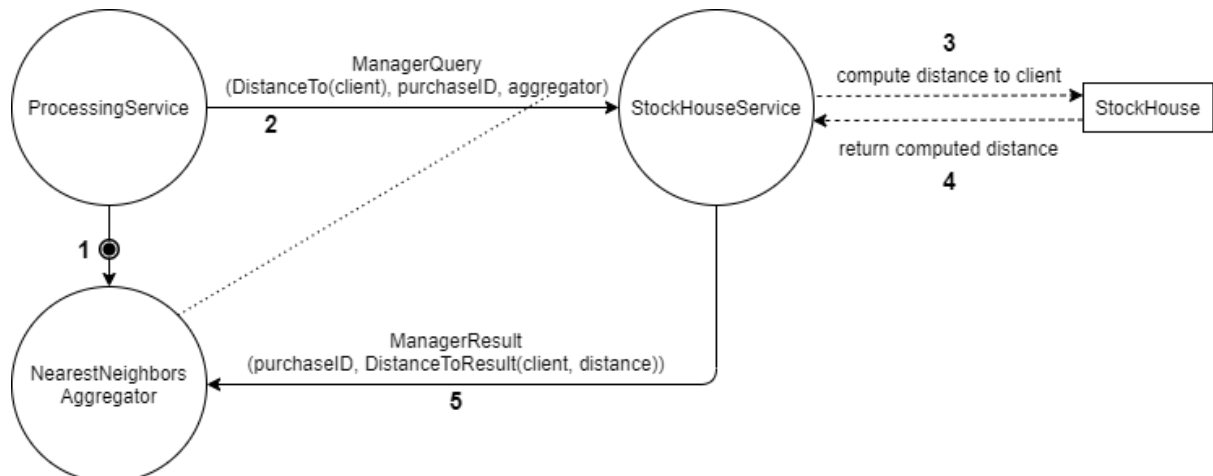


Figure 2.5. A DistanceTo-ManagerQuery sent to compute the distance to a given client.

There are two ManagerCommands supported. The first one is for reserving goods through a FillOrder Command, which returns a ManagerEvent with an InStock Event saying which StockHouse reserved which items. The second one is for undoing the reservation of goods in case too much was reserved or the order is delayed. The latter Command does not return a ManagerEvent as it is seen as a “background operation”. Both Commands will have the ephemeral child FillOrderAggregator as replyTo address in our case.

ManagerRejections are sent back to the NearestNeighborAggregator or FillOrderAggregator in case an exception is thrown by the StockHouse, in which case the aggregators simply log the error.

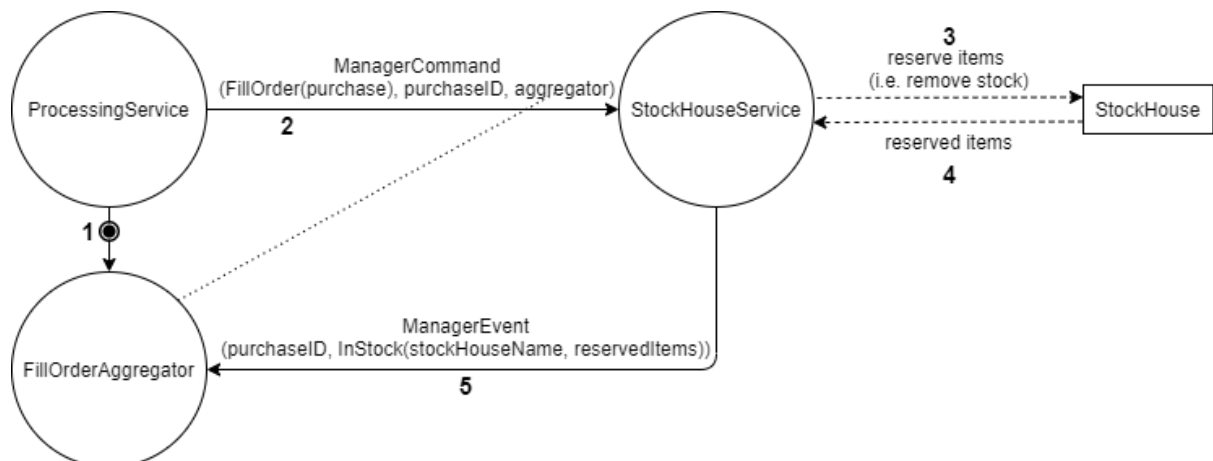


Figure 2.6. A FillOrder-ManagerCommand sent to reserve stock from a StockHouse.

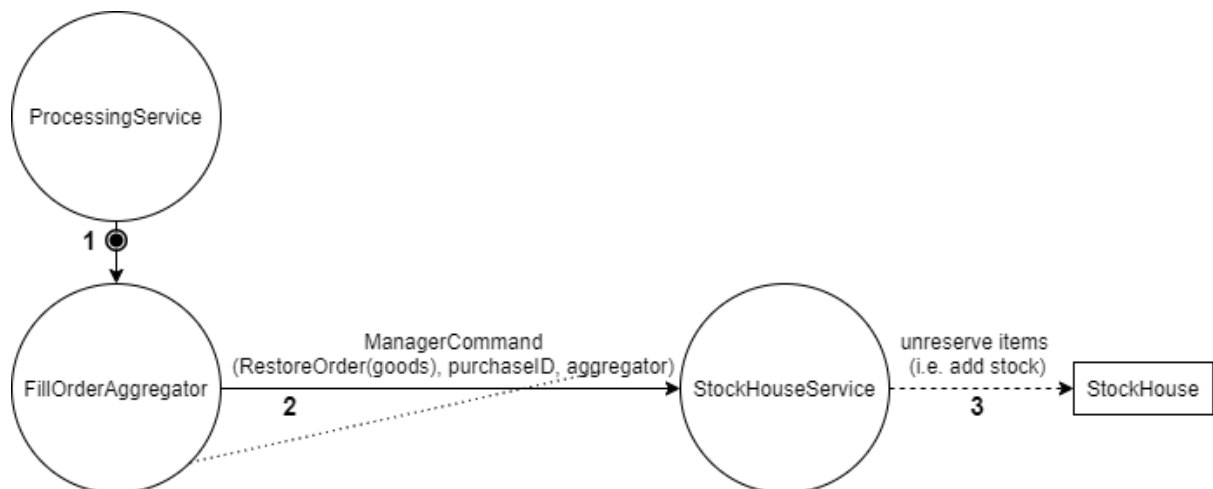


Figure 2.7. A RestoreOrder-ManagerCommand sent to unreserve stock from a StockHouse.

ProcessingService

The ProcessingService is responsible for all communication concerning the actual processing of a made Purchase. It communicates with ClientServices and knows all the addresses of available StockHouseServices. Ephemeral children are used to aggregate the responses from these StockHouseServices.

A ProcessingService only processes one Purchase at a time. It uses an Option and a PrimePriorityQueue field for this purpose.

The Option field keeps track of the purchase request that is currently being processed. This allows us to check if the ProcessingService is currently busy (Some) or not (None) and to always be able to validate messages and know who to inform when needed.

The PrimePriorityQueue is a custom priority queue that prioritizes on the hasPrime field of Clients to ensure their Purchases are prioritized. It internally uses 2 separate queues to also ensure the FIFO behavior is preserved per priority (in contrast to regular PriorityQueues). Alternatively, we could have first prioritized on hasPrime and then on a timestamp, but our approach supports our functional programming style while the built-in mutable PriorityQueue does not.

Every time the ProcessingService receives a new PurchaseConfirmed order it adds it to the queue (or processes it immediately if the queue is empty). When the ProcessingService is finished with a Purchase, it will start processing the next request from the queue.

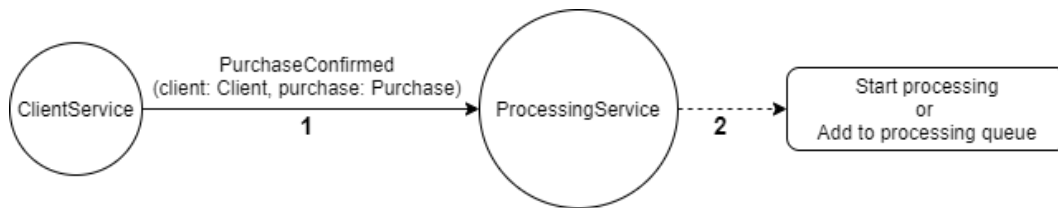


Figure 2.8. First phase of processing: Receiving *PurchaseConfirmed* messages.

Processing a Purchase happens mainly with the help from ephemeral child aggregators, following the principles of the **Aggregator Pattern**.

Processing a request starts by creating an ephemeral child *NearestNeighborsAggregator*, after which the *ProcessingService* will query all *StockHouseServices* to find their distance to the Client making the Purchase. The ephemeral child receives all the responses and then aggregates them into a single response for the *ProcessingService*. The message to the *ProcessingService* contains only the *ActorRefs* of the *StockHouseServices* for the nearest *StockHouses*. Finally, the *NearestNeighborsAggregator* will terminate itself as it has served its purpose.

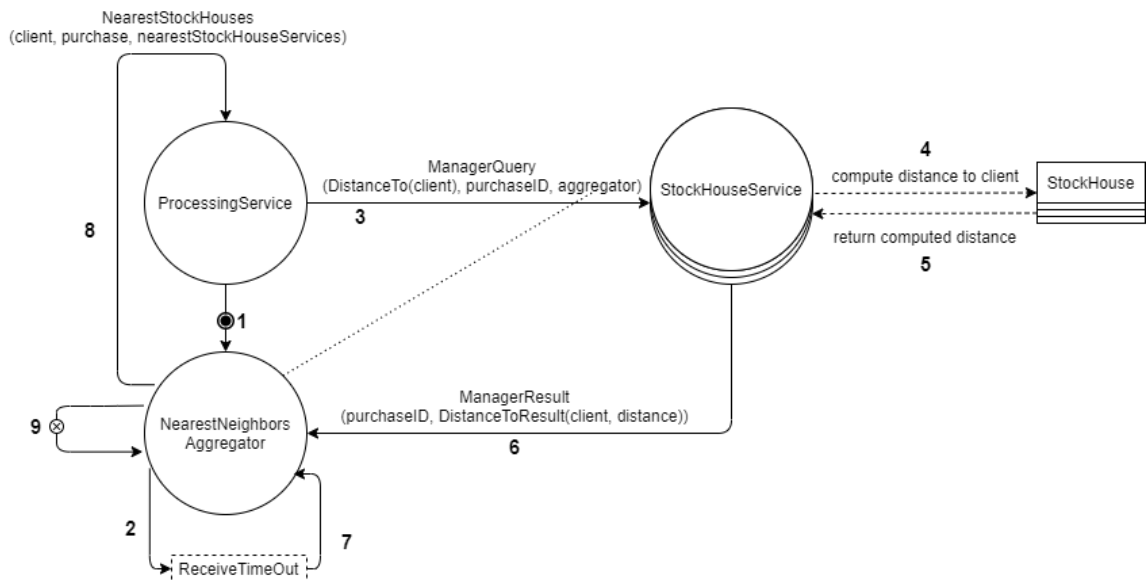


Figure 2.9. Second phase of processing: Computing the nearest *StockHouses*.

Once the *ProcessingService* received the *ActorRefs* of the *StockHouseServices* with the nearest *StockHouses*, it will create an ephemeral child *FillOrderAggregator* which is made for one specific Client and Purchase (i.e. for one specific order). The *ProcessingService* will message the nearest *StockHouseServices* to reserve stock to fill this order and the *FillOrderAggregator* receives and aggregates the responses. The **Aggregator Pattern** is clearly present here. The aggregator uses a *FillOrderResultBuilder* to decide when enough responses have been received to fill the order. Late responses or responses with reserved surpluses are unreserved again by sending a *RestoreOrder* message to those *StockHouseServices*. These *RestoreOrder* messages are the reason the *FillOrderAggregator* only terminates itself after a *ReceiveTimeOut* instead of immediately after filling the order. This way, late *InStock* messages can still easily be reversed by *RestoreOrder* messages.

InStock messages could still be coming after the *FillOrderAggregator* is terminated (or could simply be lost). However, this is expected to be fixed at a later point when the *StockHouseService* asks for confirmation before actually sending the goods (instead of just reserving the goods for shipment). Sending the actual goods is expected to use reliable message sending patterns like business handshake. However, that phase is out of scope for the assignment and has not been implemented.

Alternatively, *FillOrder* could also make use of a (perhaps modified) business handshake pattern, but failure when simply reserving stock is less detrimental so this has not been done.

Once the order is completed, the FillOrderResultBuilder will return a tuple with two results. It will return a list of StockHouseShipments (to eventually be sent back to the Client) and a list of surplus items that were reserved in the different StockHouses (to send RestoreOrders for reversing redundant reservations).

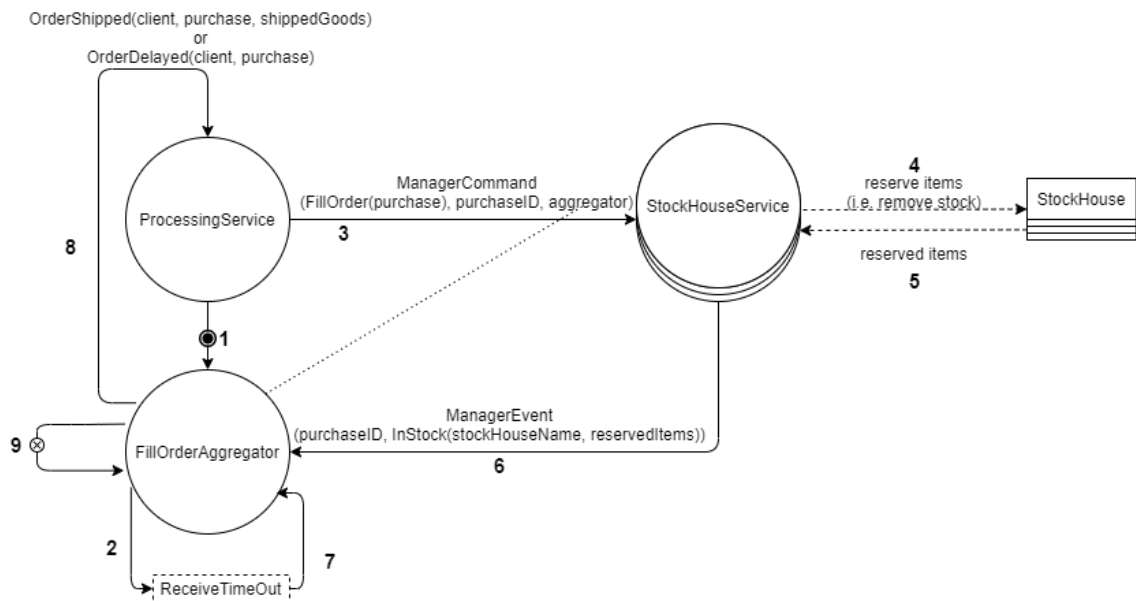


Figure 2.10. Third phase of processing: Requesting the nearest StockHouses to reserve items.

OrderShipped and OrderDelayed messages are first sent to the ProcessingService, so it knows the processing of the Purchase is finished and it can start processing a new one. The ProcessingService is the service that actually replies to the ClientService by sending through the same OrderShipped or OrderDelayed message. This does not follow the principles of the Forward Flow pattern, but otherwise an additional message had to be sent to the ProcessingService to indicate that the Purchase is fully processed. Furthermore, it seemed like a good idea to let the ProcessingService work as an intermediary for all processing-related communication.

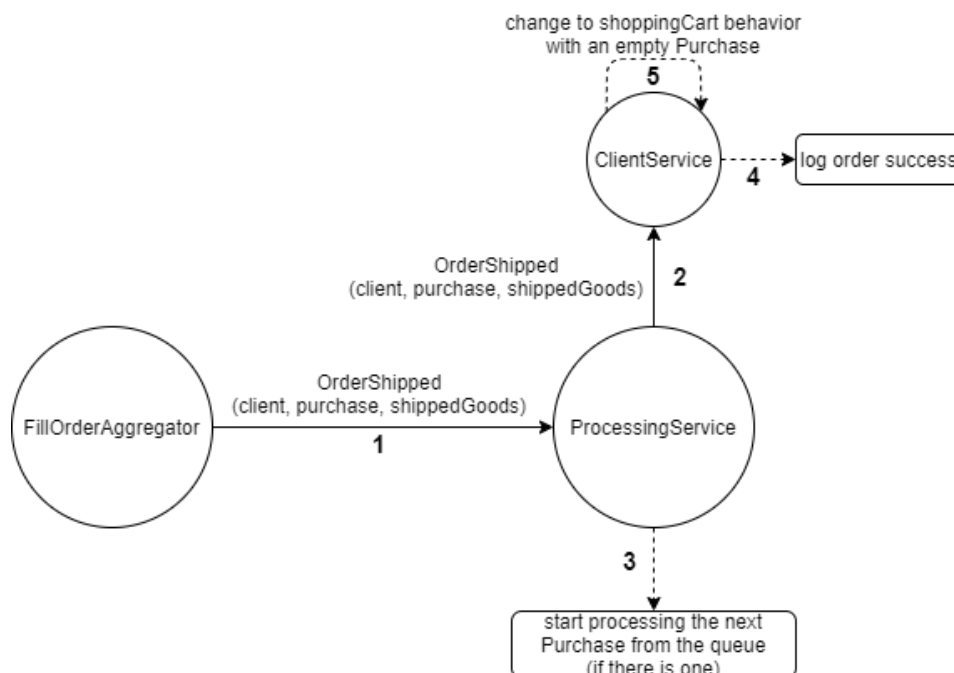


Figure 2.11. Fourth and final phase of processing: Receiving an OrderShipped confirmation. Analogous behavior occurs when an OrderDelayed message would be received.