Katholieke
Universiteit
Leuven

**Department of
Computer Science**

DOCUMENT PROCESSING

The complete architecture

Software Architecture (H09B5a and H07Z9a) – Part 2b

Jeroen Reinenbergh (r0460600)
Jonas Schouterden (r0260385)

Academic year 2014–2015

Contents

1 Introduction

The goal of this project was to design a document processing system. In this document, we describe the final architecture, which was designed based on the requirements from the domain analysis and the priorities of these requirements given by our stakeholders. The provided initial architecture was used as a starting point to achieve this. Section 2 lists the architectural decisions for all non-functional requirements and discusses the final architecture. Section 3 provides and discusses the main context and decomposition diagrams of our architecture (i.e. the context and primary diagram of the component-and-connector view) along with a discussion of the main architectural decisions involved. Section 4 provides and discusses the more fine-grained decompositions of some of the major components in the main decomposition. Section 5 provides and discusses the deployment of the components of the component-and-connector view on physical nodes. Finally, Section 6 illustrates how our architecture accomplishes the most important functionality and data flows using sequence diagrams. Afterwards, Appendix A lists and describes all components of the component-and-connector view and their interfaces and Appendix B lists and describes the data types used in these interfaces.

2 Overview

This section gives a high-level but complete overview of the system: it lists the design decisions for all non-functional requirements and provides a discussion concerning the strong and weak points of the architecture.

2.1 Architectural decisions

In this section, we give an overview of the architectural decisions made in our architecture in order to achieve the requirements given in the domain analysis and the residual drivers given in the provided initial architecture. However, we will not repeat any decisions that were already documented in this provided initial architecture as they can be found there.

Av1a & Av2a: Notifying the appropriate operator within 1 minute *Av1a* and *Av2a* require the system to notify the eDocs operator within 1 minute in case of failure of the internal infrastructure responsible for generating documents or the internal (sub-)system responsible for storing documents in personal document stores. To achieve this, the `NotificationHandler` handles all incoming notifications and forwards those that are destined for an eDocs operator to the `EDocsAdminFacade`, which in turn delivers these to the external `EDocsAdminClient`. To ensure that notifications get sent to the eDocs operator in a timely fashion, both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated from all other nodes to increase communication performance between the two as well as individual performance of both components separately.
For more details, we refer to Section 3.1.1 and Section 5.

Av1b: Storing the status of an individual job *Av1b* requires the system to store the status of an individual job. To achieve this, the `JobManager` accepts all read and write requests regarding the storage of job data, including status information.
For more details, we refer to Section 3.1.2 and Section 4.5.

Av2b: Temporary storage and user notification upon PDSDB failure *Av2b* requires (1) documents that should be delivered via the personal document store to be cached for at least 3 hours in case of unavailability and (2) a clear message to be provided to the recipient in this case. To achieve this, we introduced the `DocumentStorageCache`: this component temporarily stores the `DocumentIds` and corresponding `RecipientIds` of all generated documents that are to be delivered to the `PDSDB` during downtime of the latter component up to a maximum of 3 hours. When the `PDSDB` turns operational again, it notifies the `DocumentStorageManager` with a sign of life, which in turn retrieves all documents and their corresponding meta data from the `DocumentDB` using the previously mentioned ids and subsequently stores them in the `PDSDB` along with the converted `DocumentMetaData`.
Note that the user therefore perceives a maximum total downtime of 3 hours and that the `RecipientFacade` is in charge of presenting the user with a clear error message after having performed a read request in his or her behalf that has failed due to the unavailability of the `PDSDB`. However, during the time needed for the `DocumentStorageManager` to transfer all documents and meta data that the cached ids refer to, it is possible that the user is not able to access (part of) his or her documents via the `PDSDB`, more specifically those that are

still being transferred. Since the requirements do not demand the user to be able to regain full functionality of his or her personal document store at once, finding a better alternative for this gradual revival of the PDSDB is out of scope.
Finally, we would like to stress that the DocumentStorageManager and the PDSDB are necessarily deployed on different nodes for the former to be able to do its caching job while the latter is not operational during downtime. More precisely, the DocumentStorageManager implicitly pings the PDSDB when storing document data in it. The echo message then, in turn, consists of the write confirmation that is subsequently received. If one of those writes should fail, all subsequent writes are internally converted into pairs of DocumentId-RecipientId writes to the aforementioned cache. Once the PDSDB is operational again and all cached ids are processed, subsequent writes to the PDSDB will no longer be redirected through the DocumentStorageCache.
For more details, we refer to Section 3.1.3, Section 4.4 and Section 5.

Av3: Zoomit failure *Av3* requires the system to be able to (1) autonomously detect when an invoice is not accepted by Zoomit, (2) temporarily store at least 2 days of these invoices locally until they are accepted by Zoomit, (3) keep on retrying to deliver a failed invoice to Zoomit in a proper fashion and (4) notify an eDocs operator after 5 failed attempts of at least 10 different invoices. To achieve this, the ZoomitFacade is able to receive an initial delivery confirmation from the ZoomitChannel (i.e. the invoice was accepted by Zoomit). Furthermore, the ZoomitDeliveryCache caches all Documents and corresponding ZoomitIds, DocumentIds, sender names and receipt tracking information for which the ZoomitFacade did not (yet) receive a delivery confirmation up to a maximum of 2 days of documents.
The ZoomitFacade then periodically retries to deliver the next document in the cached list (note that the ZoomitDeliveryCache thus stores its entries in a circular manner) to the ZoomitChannel in a proper fashion by using exponential back-off. Once the ZoomitFacade has received an initial delivery confirmation regarding one of those cached documents, the corresponding entry is deleted from the ZoomitDeliveryCache and the ZoomitFacade tries to deliver every other document in the list to the ZoomitChannel right after, disregarding the previously employed method of exponential back-off (because the external ZoomitChannelNode on which the ZoomitChannel resides, is now presumed to be operational again).
Finally, the entries in the ZoomitDeliveryCache need to be stored along with a counter that the ZoomitFacade increments after each failed attempt to send the respective document. Additionally, the ZoomitFacade keeps track of all cached entries whose counters have reached the value of 5 and sends a notification to an eDocs operator through the NotificationHandler when the amount of such entries reaches the value of 10. For more details, we refer to Section 3.1.1, Section 4.1 and Section 5.

P2: Document lookups *P2* requires the system to be able to (1) respond to all incoming document requests in a timely fashion, (2) throttle excessive requests when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements) so that the non-excessive ones continue to be handled in a timely fashion and (3) make sure that the performance of all other functionality of the system remains unaffected in case of a large number of requests. To achieve this, all potential bottleneck components that support the document lookup process are deployed on separate nodes. More precisely, the RecipientFacade and the LinkMappingFunctionality both reside on the same exclusive node as to increase the performance of these components and the communication between them (i.e. when a recipient has presented the former component with a link for which it has to determine the document it maps to). Both the AuthenticationHandler and the SessionDB are also deployed on the same individual node as to not let the authentication subprocess decrease the performance of the document lookup process. The PDSFacade is deployed on the PDSDBManagerNode (discussed in the provided initial architecture), along with the PDSLongTermManager and the two PDSReplicationManagers, as to increase communication performance between this PDSFacade and the PDSDB supercomponent. Notice that *Av1* already demands the PDSDB to be deployed separate from all other functionality according to the provided initial architecture and that *P2* can rely on this design decision to deliver an optimally increased (taking the availability constraints of *Av1* into account) performance of this supercomponent.
Because the DocumentDB, like the PDSDB, stores a large amount of documents and correspondingly handles a large amount of document requests, its documents are partitioned across several DocumentDBShards that each reside on their own DocumentDBShardNode. This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is significantly less costly regarding storage costs (which is more of a concern for documents than for other kinds of data). In order to further increase performance of the DocumentDB, the DocumentDBShardingManager, which is in charge of managing all DocumentDBShards and forwarding read and write requests to the appropriate ones, is also deployed on an individual node.

Finally, both the DocumentDB and the PDSDB have the ability to throttle excessive read requests from the RecipientFacade (all other requests remain unaffected) when the rate of these incoming requests exceeds a certain value that is specific to the kind of request (determined by the requirements). Also note that the performance of all other functionality of the system remains unaffected even at this maximum, since all potential bottleneck components that support the document lookup process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the document lookup process and a sufficient level of performance for all other functionality. For more details, we refer to Section 3.1.4, Section 3, Section 4.8, Section 4.7, Section 4.2 and Section 5.

P3: Status overview for customer administrators *P3* requires (1) the status overview of all document processing jobs initiated by a certain customer organization to be delivered to the respective customer administrator(s) in a timely fashion (i.e. the provided status of a document processing job should be consistent up to 1 minute ago) and (2) the construction of this status overview not to hinder other parts of the system. To achieve this, all potential bottleneck components that support the status overview process are deployed on separate nodes as to increase their performance. More precisely, both the AuthenticationHandler and the SessionDB are deployed on the same individual node as to not let the authentication subprocess decrease the performance of the status overview process. Furthermore, the CustomerOrganizationFacade and the JobManager supercomponent are also deployed separate from all other components in the system. More specifically, the former occupies a single node by itself, while the latter is responsible for storing all jobs, which are partitioned across several JobDBShards that each reside on their own JobDBShardNode too. This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is less costly regarding storage costs. In order to further increase performance of the JobManager and the corresponding speed at which job statuses can be requested, the JobDBShardingManager, which is in charge of managing all JobDBShards and forwarding read and write requests to the appropriate ones, is also deployed on an individual node.
Note that the performance of all other functionality of the system remains unaffected during the construction of such a status overview, since all potential bottleneck components that support this status overview process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the status overview process and a sufficient level of performance for all other functionality.
Also note that the JobDBShardManager and corresponding JobDBShards do not distinguish between jobs of different age during the construction of an overview. Although the requirements mention a different minimal response time for each type of job with regard to its age, our system does not offer this distinction. Instead, the JobDBShardManager and corresponding JobDBShards simply present the customer administrator(s) with a response time that is equal to the lowest of all minimal response times mentioned in the requirements. This strategy was chosen due to the fact that (1) different dynamic storage requirements for jobs along with their periodical synchronization would largely outweigh this simple storage approach with respect to implementation and deployment costs and (2) the same level of performance per type of job can easily be provided by this simple storage approach (i.e. the uniform response time per job should be less than the lowest of all minimal response times) because of the small storage capacity that is needed for job data. For more details, we refer to Section 4.8, Section 4.5 and Section 5.

M1: New type of document: bank statements *M1* requires (1) the interface for initializing document processing jobs to be easily extendible with the functionality to provide the raw data for other types of documents, (2) the interface for registering companies to be easily extendible with the functionality to enable the new types of documents for new customer organizations and configure their template for these types of documents, (3) the functionality for generating documents to be easily extendible with the new generation steps for the new types of documents, (4) the generation of the new types of documents to reuse as much of the existing functionality for generating documents as possible, (5) the storage of generated documents of the new types of documents not to require any changes to the existing system for storing generated documents and (6) the interface for consulting the personal document store and specific documents to be easily extendible with the functionality to show the new types of documents. To achieve this, all relevant interfaces of the components involved in the data flows of these processes, including the specified data types, are generic enough to cope with different document types. Furthermore, all potential changes are encapsulated by the RawDataHandler and the DocumentGenerationManager supercomponent. More specifically, the former will now also need to handle the verification of raw data that is characteristic for these new types of documents. The generic internal structure of the latter also ensures that the generation of new document types will reuse as much of the existing functionality for generating documents as possible. Note that the type and the format of a document are two

different things: the first being the focus of *M1* and the second being the internal structure of a `Document` data type, which remains untouched in the context of *M1*. Since the `RecipientFacade` only has knowledge about the latter, no changes need to be made to the `RecipientFacade` upon introduction of a new document type. For more details, we refer to Section 3, Section 4.3, Section 4.8 and Appendix B.

M2: Multiple print & postal services *M2* requires (1) the system to be able to easily switch print & postal service in the future, (2) the incorporation of this new print & postal service in the system to only affect the final steps of the document processing flow. To achieve this, the `Print&PostalServiceFacade` encapsulates all changes that might need to be made in order to seamlessly incorporate a new print & postal service in the system, e.g. additional methods, external interfaces, ... For more details, we refer to Section 4.1.

M3: Dynamic selection of the cheapest of print & postal services *M3* requires the system to be easily extendible in order to be able to (1) employ multiple print & postal services across the world, (2) receive the daily price and capacity of each print & postal service and (3) dynamically select (i.e. on a daily basis) the most suited print & postal service based on the system's document processing load, the price of the service and the location to which the documents should be sent. To achieve this, the `Print&PostalServiceFacade` encapsulates all changes that might need to be made in order to seamlessly incorporate this new functionality in the system. More precisely, this component will only require (1) an additional external interface to each of the employed print & postal services, (2) an internal interface to the `OtherDB` to consult all SLAs in order to retrieve the approximate location and size of all batches for the day and (3) its internal structure to be adapted to the new functionality. For more details, we refer to Section 4.1.

2.2 Discussion

2.2.1 The proposed eDocs architecture in retrospect

In this document, we describe the architecture of the eDocs system, which was designed based on the requirements from the domain analysis and the priorities of these requirements given by our stakeholders. The provided initial architecture was used as a starting point to achieve this. In terms of the requirements, there were no hard trade-offs. The requirements were handled based on their relative priorities, but in the end, all requirements were compatible and are supported in the final architecture.
In terms of availability, we tackled the requirements given by our stakeholders. As such, no persistent data will be lost as a result of the failure of a single `GeneratorNode` and the system will continue to function in the presence of failure of the personal document store (i.e. the `PDSDB` supercomponent). However, as the architects of this system, we want to state that there are still single points of failure in our architecture. For example, none of the end user facades (i.e. the `RecipientFacade,` and the `CustomerOrganizationFacade`) are replicated and as a result, the failure of a single node (on which one of those facades resides) can make the system unavailable for a large number of those end users. Similarly, databases other than the personal document store (i.e. the `PDSDB` supercomponent) are not replicated and as a result, the failure of a single database node (e.g. the `OtherDBNode`) can lead to the loss of, for example, all credential data. Recovering this data without a back-up is nearly impossible and will lead to large economical costs for our company.
In terms of performance and scalability, the largest bottlenecks in the system are able to scale out because of their individual deployment. As a result, this architecture can be expected to support a large number of users when the system grows. As architects of this system, we believe that the first limits to scalability will be the end user facades, which are now each located on a single node. However, we believe that replicating these nodes in order to scale out to a larger number of end users will not lead to high costs, since all except the `RecipientNode` are stateless because of the separate `AuthenticationHandler` and `SessionDB`. Note, however, that the `RecipientNode` can easily be made stateless by relocating the `LinkMappingFunctionality` component to another node. Another scalability issue rises from the fact that the `JobFacade`, which is also a major bottleneck component, is deployed together with components that require a lower level of scaling due to a significant difference in potential load (because the `JobFacade` also handles all requests to the `OtherDB` that are given a `JobId` as an argument). To remedy this, the `JobFacade` should merely be deployed on its own individual node, upon which it can be replicated to support scalability measures.
As such, we believe that this architecture supports all given requirements and will be able to support a growing number of end users, but that more care should be given to availability in order for the system and our company to survive in the long term.

2.2.2 Encountered inconsistencies and corresponding improvements

Upon documenting the final steps in our architectural process, we encountered some inconsistencies. Because restarting this process was no longer a realistic option at this point, we decided to include those inconsistencies in this section along with possible improvements in the architecture to remedy them.

Joint deployment of DocumentGenerationManager Upon documenting the final steps in our architectural process, it became clear to us that the deployment configuration of the `DocumentGenerationManager` does not fully comply with the requirements from the domain analysis. More precisely, according to *Av1*, failure of the infrastructure for generating documents "should not affect the availability of any type of persistent data" and "should not affect the availability of other functionality of the system". Since some components, residing on the same node as the `DocumentGenerationManager` (i.e. the `ResidualNode`), do store persistent data (e.g. the `DocumentStorageCache`) and other system functionality is obviously not unaffected by the failure of the `DocumentGenerationManager`, this component should be deployed on its own individual node.

Unidentifiable email delivery failure Upon documenting the final steps in our architectural process, it became clear to us that the email delivery failure notification, originating from the external `EmailChannel` (i.e. the email provider), does not contain sufficient information for the `EmailFacade` to properly act upon. More precisely, in our current architecture, the `EmailFacade` is not able to look up and notify the corresponding customer organization, since the received email delivery notification cannot possibly contain any reference to the respective customer organization (due to the fact that this information was never sent to the `EmailChannel` in the first place).
To remedy this situation, the `EmailFacade` should temporarily store a mapping between the email (along with the corresponding customer organization id) and a email id. Upon sending the email to the `EmailChannel`, the `EmailFacade` sends this email id along with it. Upon sending an email delivery failure notification to the `EmailFacade`, the `EmailClient` is now able to indicate which particular email the notification is about by sending its email id along with it. The `EmailFacade` can subsequently look up the corresponding customer organization id and send an appropriate notification to the customer administrator that represents this customer organization. Note that this improvement assumes that the email provider notifies the `EmailFacade` of an email delivery failure in a timely fashion as to not let the latter component be overloaded with this temporary storage of emails.

Incomplete job overviews Upon documenting the final steps in our architectural process, it became clear to us that the creation of job status overviews does not fully comply with the requirements from the domain analysis. More precisely, the requesting `CustomerOrganizationFacade` cannot be presented with any timestamps, since none are stored along with the job status in our current architecture.
To remedy this, it suffices to store a timestamp along with every write request that is made to a job status in the `JobManager`.

3 Client-server view (UML Component diagram)

Context diagram The context diagram of the component-and-connector view is given in Figure 1. As shown, six distinct types of external components communicate with the system: (i) the EDocsAdminClient represents a client device of an administrator of eDocs that communicates with the eDocs System, (ii) the CustomerOrganizationClient represents a client device of a Customer Organization (i.e. Customer Administrator and Customer Information System) that communicates with the eDocs System, (iii) the RecipientClient represents a client device of an unregistered or registered recipient of eDocs that communicates with the eDocs System, (iv) the ZoomitChannel represents the servers of Zoomit to which documents can be sent for further delivery, (v) the Print&PostalServiceChannel represents the servers of a print & postal service to which documents can be sent for further delivery, (vi) the EmailChannel represents a mail server of an e-mail provider to which documents can be sent for further delivery. Notice that each external interface on the client side is provided by a separate facade component and that each delivery channel interfaces with the `DeliveryFunctionality` component.

Primary diagram The main component-and-connector view is shown in Figure 2. All components that were decomposed further are highlighted. The DeliveryFunctionality is further decomposed in Section 4.1, the

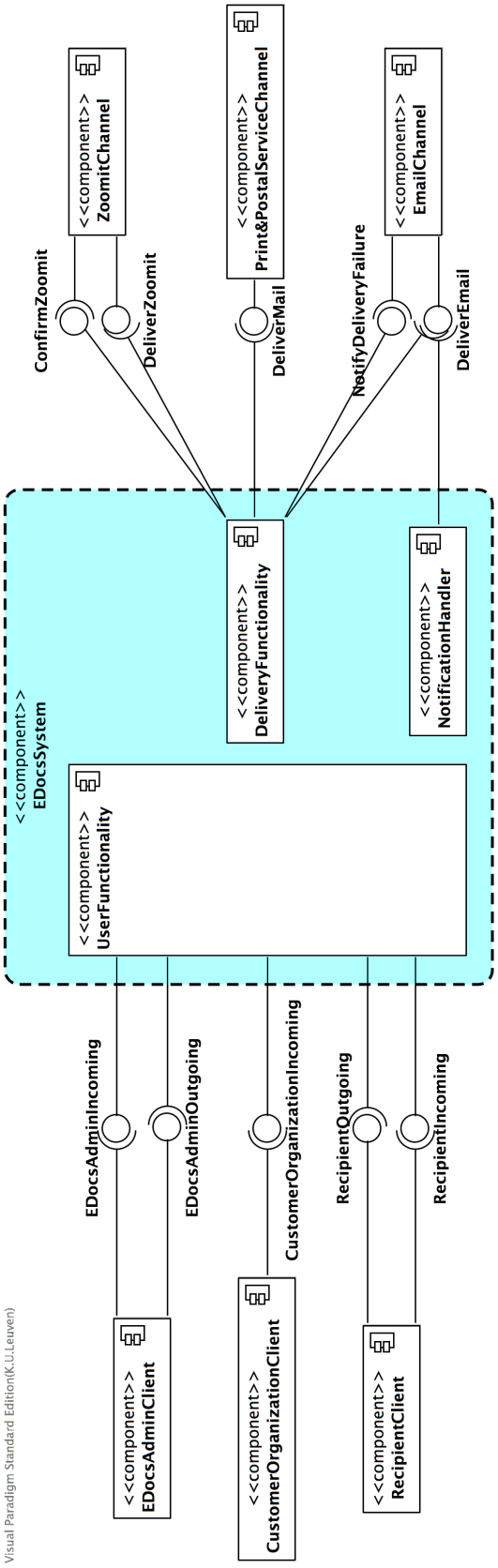

Figure 1: Context diagram for the client-server view.

DocumentDB in Section 4.2, the DocumentGenerationManager in Section 4.3, the DocumentStorageFunctionality in Section 4.4, the JobManager in Section 4.5, the LinkMappingFunctionality in Section 4.6, the PDSDB in Section 4.7 and the UserFunctionality in Section 4.8. The descriptions of all components in Figure 2, their sub-components and their interfaces are given in Appendix A.

3.1 Main architectural decisions

Figure 2 provides the overall description of the architecture and illustrates the most important architectural decisions for achieving the required qualities. Here we document how our architecture fulfils the most important qualities (based on the priorities given by the stakeholders). Although the qualities that were already discussed in the provided initial architecture will not be analysed any further here, their residual drivers will. The resulting qualities thus consist of Av1a & Av2a: Notifying the appropriate operator within 1 minute, Av1b: Storing the status of an individual job and Av2b: Temporary storage and user notification during PDSDB failure. In addition, we also highlight P2: Document lookups, because of the importance of this quality to a significant part of the recipient base.

3.1.1 Av1a & Av2a: Notifying the appropriate operator within 1 minute

Firstly, *Av1a* and *Av2a* require the system to notify the eDocs operator in case of failure of the internal infrastructure responsible for generating documents or the internal (sub-)system responsible for storing documents in personal document stores (cf. the analysis of *Av1* and *Av2* in the provided initial architecture respectively). Therefore, the `NotificationHandler` handles all incoming notifications from the respective components (i.e. the `DocumentGenerationManager` and the `PDSDB`) and forwards those that are destined for an eDocs operator to the `EDocsAdminFacade`, which in turn delivers these to the external `EDocsAdminClient`.

Secondly, to ensure that notifications get sent to the eDocs operator in a timely fashion (i.e. within 1 minute), both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated from all other nodes to increase communication performance between the two as well as individual performance of both components separately (see the deployment view in Section 5). As was already pointed out and discussed in the provided initial architecture, it is the responsibility of the `DocumentGenerationManager` and the `PDSDB` that the `NotificationHandler` itself gets notified in a timely fashion. The `DocumentGenerationManager` and the `PDSDB` are further decomposed in Sections 4.3 and 4.7 respectively.

Alternatives considered

Alternative for NotificationHandler An alternative for the use of the `NotificationHandler` would be for all components in our system to contact the appropriate userfacades (i.e. the `RecipientFacade`, the `CustomerOrganizationFacade` and/or the `EDocsAdminFacade`) directly and to let them handle all subsequent steps of the notification process. This would entail (1) an increase in coupling between those facades and the rest of the system, (2) a decrease in cohesion of all three userfacades and (3) a decrease in adaptability of the system due to the fact that all notification functionality is now spread across several components. For these reasons, we decided to include the `NotificationHandler` as a component in our architecture.

Alternative for joint deployment of NotificationHandler and EDocsAdminFacade In the current architecture, both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated from all other nodes to increase overall performance. An alternative for this strategy would be to deploy the `NotificationHandler` on its own node, which would increase the individual performance of this component. Unfortunately, this would also decrease the communication performance between this component and the `EDocsAdminFacade`. Since *Av1a* and *Av2a* only dictate an eDocs operator to receive his or her notifications in a timely fashion and the increase in individual performance of the `NotificationHandler` would not outweigh the decrease in communication performance with the `EDocsAdminFacade` in this case, we chose to go with our original approach and deploy both components on the same node (as discussed above).

3.1.2 Av1b: Storing the status of an individual job

Av1b requires the system to store the status of an individual document processing job in order to be able to show the status of the jobs affected by the failure of the internal infrastructure responsible for generating documents (cf. the analysis of *Av1* in the provided initial architecture). To achieve this, the `JobManager` accepts all read and write requests regarding the storage of job data, including status information. More specifically, the

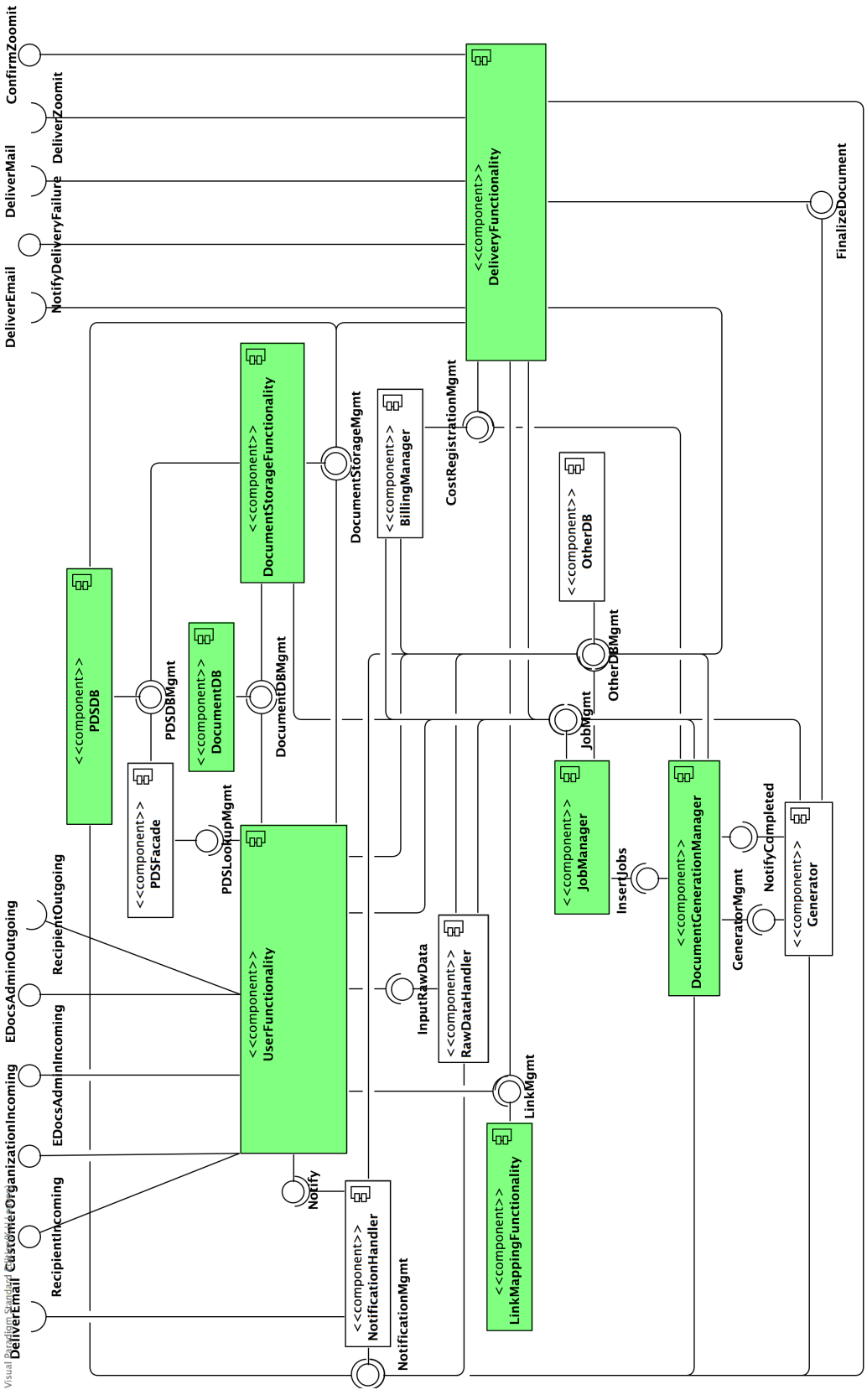

Figure 2: Primary diagram of the client-server view.

JobFacade subcomponent forwards these requests to the JobDBShardingManager, which in turn performs the read or write operation on the appropriate JobDBShard (see the decomposition of the JobManager in Section 4.5).

Alternatives considered

Alternative for JobFacade indirection In our current architecture, the JobFacade subcomponent forwards all read and write requests to the JobDBShardingManager. An alternative for this approach would be remove this extra level of indirection (i.e. the JobFacade) and let the JobDBShardingManager handle all requests directly. Although this would greatly simplify the internal workings of the JobManager supercomponent, this would also complicate the workings of other components (e.g. the DeliveryFunctionality, the DocumentGenerationManager and the UserFunctionality) that call its interface. The reason for this is that, in our current architecture, all calls to the OtherDB based on a JobId (i.e. the JobId is given as an argument in a method call to the interface) are routed through the JobFacade, which in turn fetches the required data from the OtherDB. Removing this level of indirection would result in those components (from which the routed calls originated) calling the OtherDB directly after having fetched the appropriate job information from the JobDBShardingManager themselves. In order to centralize this functionality and alleviate these components from the burden of calling both the JobDBShardngManager and the OtherDB, we decided to include the JobFacade as a fully-fledged component in our architecture.

3.1.3 Av2b: Temporary storage and user notification upon PDSDB failure

Firstly, *Av2b* requires documents that should be delivered via the personal document store to be cached for at least 3 hours in case of unavailability (cf. the analysis of *Av2* in the provided initial architecture). Therefore, we introduced the DocumentStorageFunctionality: the DocumentStorageCache that is part of this component temporarily stores the DocumentIds and corresponding RecipientIds of all generated documents that are to be delivered to the PDSDB during downtime of the latter component up to a maximum of 3 hours. When the PDSDB turns operational again, it notifies the DocumentStorageManager (which is also a subcomponent of the DocumentStorageFunctionality) with a sign of life, which in turn retrieves all documents and their corresponding meta data from the DocumentDB using the previously mentioned ids and subsequently stores them in the PDSDB along with the converted DocumentMetaData (i.e. the original DocumentMetaData, including the corresponding RecipientId but omitting the EmailAddress if present). Note that the DocumentStorageManager and the PDSDB are necessarily deployed on different nodes for the former to be able to do its caching job while the latter is not operational during downtime (see the deployment view in Section 5). More precisely, the DocumentStorageManager implicitly pings the PDSDB when storing document data in it. The echo message then, in turn, consists of the write confirmation that is subsequently received. If one of those writes should fail, all subsequent writes are internally converted into pairs of DocumentId-RecipientId writes to the aforementioned cache. Once the PDSDB is operational again and all cached ids are processed, subsequent writes to the PDSDB will no longer be redirected through the DocumentStorageCache. The DocumentStorageFunctionality is further decomposed in Section 4.4.

Secondly, *Av2b* requires a clear message to be provided to the recipient in case of unavailability of the personal document store. This responsibility is delegated to the RecipientFacade in the following way: this component presents the user with a clear error message after having performed a read request in his or her behalf that has failed due to the unavailability of the PDSDB. Note that the recipient therefore perceives a maximum total downtime of 3 hours. However, during the time needed for the DocumentStorageManager to transfer all documents and meta data that the cached ids refer to, it is possible that the user is not able to access (part of) his or her documents via the PDSDB, more specifically those that are still being transferred.

Alternatives considered

Alternative for sign of life In the current architecture, the PDSDB notifies the DocumentStorageManager with a sign of life when the former turns operational again after failure. An alternative for this strategy would be not to rely on this sign of life, but instead to consider all write attempts to the PDSDB as implicit ping messages and their corresponding confirmations as implicit echo messages. An obvious advantage here is that the PDSDB does now not have to actively contact the DocumentStorageManager upon revival. The problem with this approach, however, is that the view of the personal document store for the recipient cannot be controlled by the system. The reason for this can best be made clear with an example scenario: imagine the PDSDB turning operational again after failure and the DocumentStorageCache still containing some document entries due to

the fact that no writes have occurred for a while (and the DocumentStorageManager has therefore not started transferring any of the cached documents). The recipient will not be able to request those documents via the PDSDB until another write occurs: a situation for which the time of resolution is rather unpredictable. Since these kind of scenarios would most likely result in damage to the reputation of eDocs and actively sending a sign of life upon revival only requires some minor changes to the two components involved, we decided to go with the latter strategy.

Alternative for gradual revival of the PDSDB In our current architecture, the recipient perceives the PDSDB to be down for a maximum of 3 hours. However, during the time needed for the DocumentStorageManager to transfer all documents and meta data that the cached ids refer to, it is possible that the user is not able to access (part of) his or her documents via the PDSDB, more specifically those that are still being transferred. An alternative for this approach would be to let the PDSDB keep blocking all read requests until the DocumentStorageManager has stopped transferring. This would mean that the maximum waiting time for recipients trying to access their documents in the PDSDB is increased with the total transfer time needed by the DocumentStorageManager. However, since the requirements do not demand the user to be able to regain full functionality of his or her personal document store at once, we decided to go with a gradual revival of the PDSDB in order to be able to offer the recipient a view of the PDSDB (though it could be an incomplete one) as soon as possible to increase performance of the document lookup process (discussed in Section 3.1.4).

Alternative for DocumentStorageCache An alternative for the use of the DocumentStorageCache would be to let the DocumentStorageManager actively look for documents in the DocumentDB that are not yet, but should be, present in the PDSDB upon revival of this last component. A clear advantage of this approach is that the downtime of the PDSDB component is no longer restricted by the aforementioned 3-hour cache. An important disadvantage, however, lies in the fact that the DocumentStorageManager is burdened with a significant amount of extra work and will require more expensive hardware to cope with this. Since the support for a longer downtime of the PDSDB is out of scope, this alternative approach was not chosen.

3.1.4 P2: Document lookups

Firstly, *P2* requires the system to be able to respond to all incoming document requests in a timely fashion. To achieve this, all potential bottleneck components that support the document lookup process are deployed on separate nodes (see the deployment view in Section 5). More precisely, the RecipientFacade (which is a subcomponent of the UserFunctionality and is responsible for interfacing with the RecipientClient as can be reviewed in Section 4.8) and the LinkMappingFunctionality both reside on the same exclusive node as to increase the performance of these components and the communication between them (i.e. when the RecipientClient has presented the former component with a link for which it has to determine the document it maps to). Furthermore, the AuthenticationHandler and the SessionDB (both subcomponents of the UserFunctionality) are also deployed on the same individual node as to not let the authentication subprocess decrease the performance of the document lookup process. The PDSFacade is deployed on the PDSDBManagerNode (discussed in the provided initial architecture), along with the PDSLongTermManager and the two PDSReplicationManagers, as to increase communication performance between this PDSFacade and the PDSDB supercomponent (see also the decomposition of the PDSDB in Section 4.7). Notice that *Av1* already demands the PDSDB to be deployed separate from all other functionality according to the provided initial architecture and that *P2* can rely on this design decision to deliver an optimally increased (taking the availability constraints of *Av1* into account) performance of this supercomponent. Furthermore, because the DocumentDB, like the PDSDB, stores a large amount of documents and correspondingly handles a large amount of document requests, its documents are partitioned across several DocumentDBShards that each reside on their own DocumentDBShardNode (see the decomposition of the DocumentDB in Section 4.2). This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is significantly less costly regarding storage costs (which is more of a concern for documents than for other kinds of data). In order to further increase performance of the DocumentDB, the DocumentDBShardingManager, which is in charge of managing all DocumentDBShards and forwarding read and write requests to the appropriate ones, is also deployed on an individual node.

Secondly, *P2* requires the system to be able to throttle excessive incoming document requests when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements), so that the non-excessive ones continue to be handled in a timely fashion. To achieve this, both the DocumentDB and the PDSDB have the ability to throttle excessive read requests from the PDSFacade, through

which the recipient's PDSDB requests are routed, and the RecipientFacade (which is a subcomponent of the UserFunctionality) when the rate of these incoming requests exceeds a certain value that is specific to the kind of request (determined by the requirements).

Thirdly, *P2* requires the performance of all other functionality of the system to remain unaffected in case of a large number of incoming document requests. Therefore, all potential bottleneck components that support the document lookup process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the document lookup process and a sufficient level of performance for all other functionality. Also note that only requests coming from the RecipientFacade and the PDSFacade, which are inherently tied to the document lookup process, can be throttled by the DocumentDB and the PDSDB (as discussed above) and that all other requests thus remain unaffected by this throttling strategy at all times.

Alternatives considered

Alternative for sharding of the DocumentDB An alternative for sharding the DocumentDB would be to actively replicate it. However, although the performance increase is statistically equivalent in both cases, the former strategy is significantly less costly regarding storage costs. Since these cost concerns are particularly important for large data files like documents and there is no other availability requirement for document replication in this case, we chose to shard the DocumentDB instead of actively replicate it.

Alternative for separate deployment of potential bottleneck components in the document lookup process In our current architecture, all potential bottleneck components that support the document lookup process are deployed on separate nodes. An obvious alternative would be to deploy these components on the same node (assuming this would not contradict with any design decisions that were made for other requirements). Although this approach would greatly increase communication performance between all components involved, it would also decrease overall performance of the document lookup process and, more importantly, of the other functionality in the system. For these reasons, we decided to deploy these components on separate nodes as discussed above.

Alternative for the DocumentDB and the PDSDB having throttling responsibility In our current design, both the DocumentDB and the PDSDB have the responsibility to throttle excessive read requests (originating from the RecipientFacade) when the rate of these incoming requests exceeds a certain value that is specific to the kind of request (determined by the requirements). Since all document lookup requests pass through the RecipientFacade, a valid alternative for this approach would be to delegate all throttling responsibility to this component. A clear advantage of the latter strategy is the centralization of all throttling functionality, effectively increasing the modifiability of the system. An important drawback, however, is the fact that the throttling component can now not take into account the PDSDB and DocumentDB requests originating from other components in the system. This lack of knowledge results in a fixed suboptimal upper bound for document lookup requests originating from the RecipientFacade. The reason for this is that the throttling component now needs take into account the maximum amount of residual requests (i.e. requests originating from all other components in the system) in order to not let the throttling affect any other functionality in the system. Since our current architecture does support an optimal load balancing policy for document lookup requests by throttling at the source components (i.e. the PDSDB and the DocumentDB), we chose to go with the latter strategy.

4 Decomposition view (UML Component diagram)

An overview of the main component-and-connector view can be found in Section 3. All components that were decomposed any further are highlighted there and will be discussed below in greater detail. This component group consists of the DeliveryFunctionality (Section 4.1), the DocumentDB (Section 4.2), the DocumentGenerationManager (4.3), the DocumentStorageFunctionality (4.4), the JobManager (Section 4.5), the LinkMappingFunctionality (Section 4.6), the PDSDB (Section 4.7) and the UserFunctionality (Section 4.8).

4.1 DeliveryFunctionality

The decomposition of the DeliveryFunctionality is presented in Figure 3. As shown, the DeliveryFunctionality is decomposed into five submodules: the ChannelDispatcher, the EmailFacade, the Print&PostalServiceFacade,

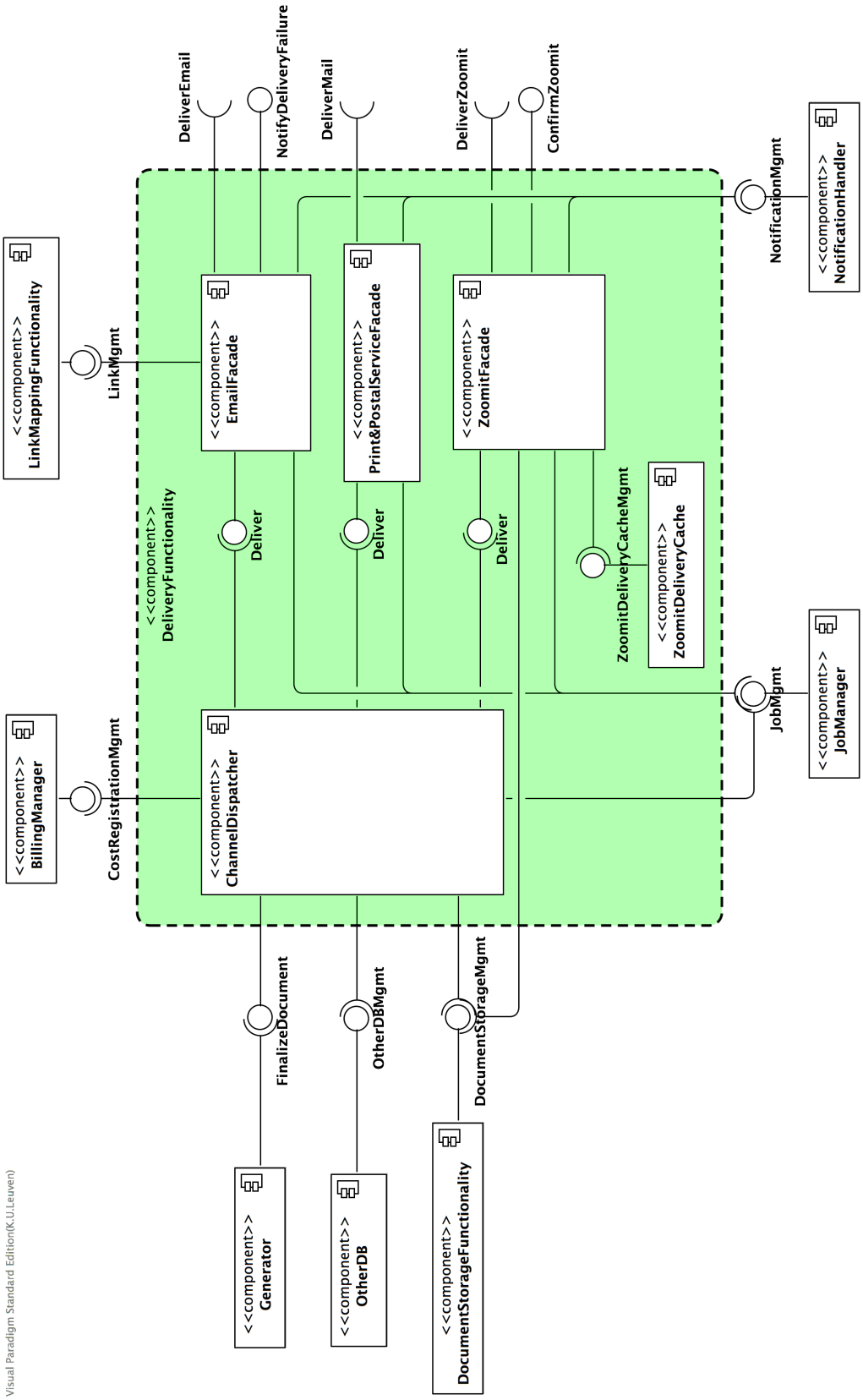

Figure 3: Decomposition of `DeliveryFunctionality`

the ZoomitFacade and the ZoomitDeliveryCache.

The DeliveryFunctionality is responsible for delivering all correctly generated documents to their corresponding recipients. More precisely, upon receiving a document from the Generator, the ChannelDispatcher decides on the channel via which the document should be sent, based on an analysis of the corresponding raw data, receipt tracking information and recipient status (i.e. registered or unregistered) that it has fetched from the JobManager (for the raw data) and the OtherDB (for the other data). Once that it has determined the correct channel for the document, the ChannelDispatcher contacts the BillingManager to register the delivery cost for the respective document. Subsequently, the ChannelDispatcher sends a documents storage request to the DocumentStorageFunctionality, indicating whether it should store the document in both the DocumentDB and the PDSDB (for a registered recipient) or in the DocumentDB alone (for an unregistered recipient). Finally, the ChannelDispatcher forwards the document to the appropriate channel facade along with the necessary information to complete the delivery process.

The EmailFacade is responsible for creating and sending emails containing (links to) the documents that are to be delivered. Upon receiving a document and corresponding delivery information from the ChannelDispatcher, several situations can occur.

- If the recipient of the document is **not registered** and receipt tracking is **disabled**, the EmailFacade constructs an email, adds the document as an attachment.
- If the recipient of the document is **not registered** and receipt tracking is **enabled**, the EmailFacade constructs an email that contains a short description of the document. This description includes a unique link, created by the LinkMappingFunctionality, that can be used to download this document.
- If the recipient of the document is **registered**, the EmailFacade constructs an email that contains a short description of the document. This description includes a link, created by the LinkMappingFunctionality, that can be used by the recipient to download this document after he or she has logged into the system. Afterwards, the EmailFacade marks the corresponding job as sent by contacting the JobManager.

After having created the email, the EmailFacade sends the email to the external EmailChannel and awaits confirmation, upon which it marks the corresponding job as sent by contacting the JobManager.

The Print&PostalServiceFacade is responsible for creating printing jobs (consisting of documents that are to be delivered to unregistered recipients) and sending them to the external Print&PostalServiceChannel for subsequent printing and delivery. This delivery process is set in motion upon receipt of a document and corresponding delivery information from the ChannelDispatcher. Note that the Print&PostalServiceFacade also marks all document processing jobs, for which printing jobs have been issued to the Print&PostalServiceChannel, as sent by contacting the JobManager.

In order to comply with *M2*, the Print&PostalServiceFacade encapsulates all changes that might need to be made in order to seamlessly incorporate a new print & postal service in the system, e.g. additional methods, external interfaces, ... Furthermore, according to *M3*, the Print&PostalServiceFacade encapsulates all changes that might need to be made in order to seamlessly incorporate the dynamic selection of the cheapest print & postal service (on a global scale) in the system. More precisely, this component will only require (1) an additional external interface to each of the employed print & postal services, (2) an internal interface to the OtherDB to consult all SLAs in order to retrieve the approximate location and size of all batches for the day and (3) its internal structure to be adapted to the new functionality.

The ZoomitFacade is responsible for sending documents to the external ZoomitChannel along with the corresponding sender name (i.e. the name of the customer organization that ordered the underlying document processing job) and the ZoomitId of the unregistered recipient to whom the document is to be delivered. This delivery process is set in motion upon receipt of a document and corresponding delivery information from the ChannelDispatcher. After having sent the document data to the external ZoomitChannel, the ZoomitFacade awaits confirmation from this component, upon which it marks the corresponding job as sent by contacting the JobManager. Furthermore, if receipt tracking is enabled, the ZoomitFacade indicates to the ZoomitChannel that it must notify the former component upon actual delivery of the document to the recipient. When the ZoomitFacade receives such a delivery notification, it marks the corresponding job as received by calling the JobManager.

In order to comply with *Av3*, the `ZoomitDeliveryCache` is responsible for caching all `Documents` and corresponding `ZoomitIds`, `DocumentIds`, sender names and receipt tracking information for which the `ZoomitFacade` did not (yet) receive an initial delivery confirmation from the `ZoomitChannel` (i.e. Zoomit did not receive the document) up to a maximum of 2 days of documents. The `ZoomitFacade` then periodically retries to deliver the next document in the cached list (note that the `ZoomitDeliveryCache` thus stores its entries in a circular manner) to the `ZoomitChannel` in a proper fashion by using exponential back-off. Once the `ZoomitFacade` has received an initial delivery confirmation regarding one of those cached documents, the corresponding entry is deleted from the `ZoomitDeliveryCache` and the `ZoomitFacade` tries to deliver every other document in the list to the `ZoomitChannel` right after, disregarding the previously employed method of exponential back-off. To further comply with *Av3*, the entries in the `ZoomitDeliveryCache` need to be stored along with a counter that the `ZoomitFacade` increments after each failed attempt to send the respective document. Additionally, the `ZoomitFacade` keeps track of all cached entries whose counters have reached the value of 5 and sends a notification to an eDocs operator through the NotificationHandler when the amount of such entries reaches the value of 10.

4.2 DocumentDB

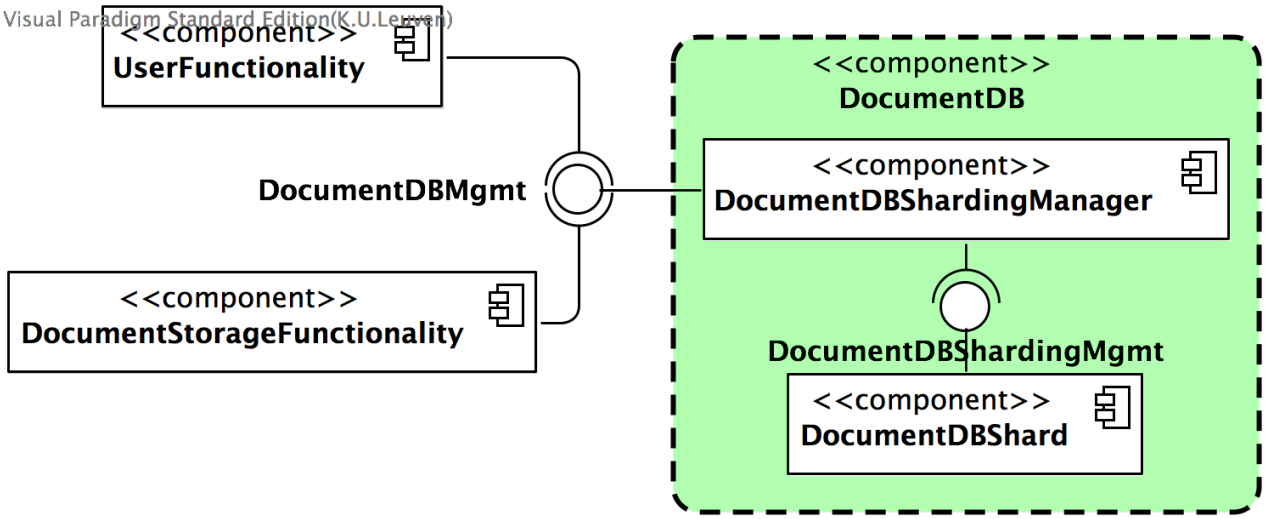

Figure 4: Decomposition of `DocumentDB`

The decomposition of the `DocumentDB` is presented in Figure 4. As shown, the `DocumentDB` is decomposed into only two submodules: the `DocumentDBShardingManager` and the `DocumentDBShard`.

The `DocumentDB` is responsible for the actual storage of all documents, along with their meta data. More specifically, the `DocumentStoragFunctionality` stores documents in it that both this component and the `RecipientFacade` are able to access. Because the `DocumentDB` stores a large amount of documents and correspondingly handles a large amount of document requests, its documents are partitioned across several `DocumentDBShards` that each reside on their own `DocumentDBShardNode` according to *P2*. In order to further increase performance of the `DocumentDB` for *P2*, the `DocumentDBShardingManager` is also deployed on an individual node. It is responsible for managing all `DocumentDBShards` (i.e. maintaining a consistent overview of all shards and their corresponding documents) and forwarding read and write requests to the appropriate ones.

Finally, *P2* requires the `DocumentDB` to be able to throttle excessive read requests from the `RecipientFacade` (see also the decomposition of the `UserFunctionality` in Section 4.8) when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements). This responsibility is delegated to the `DocumentDBShardingManager`, the only component in the document lookup pipeline that has knowledge of all incoming `DocumentDB` requests, to ensure that other `DocumentDB` requests are also taken into account in order to not hinder the other functionality of the system. Note that non-excessive recipient requests to the `DocumentDB` continue to be handled in a timely fashion during this throttling, as is dictated by *P2*.

4.3 DocumentGenerationManager

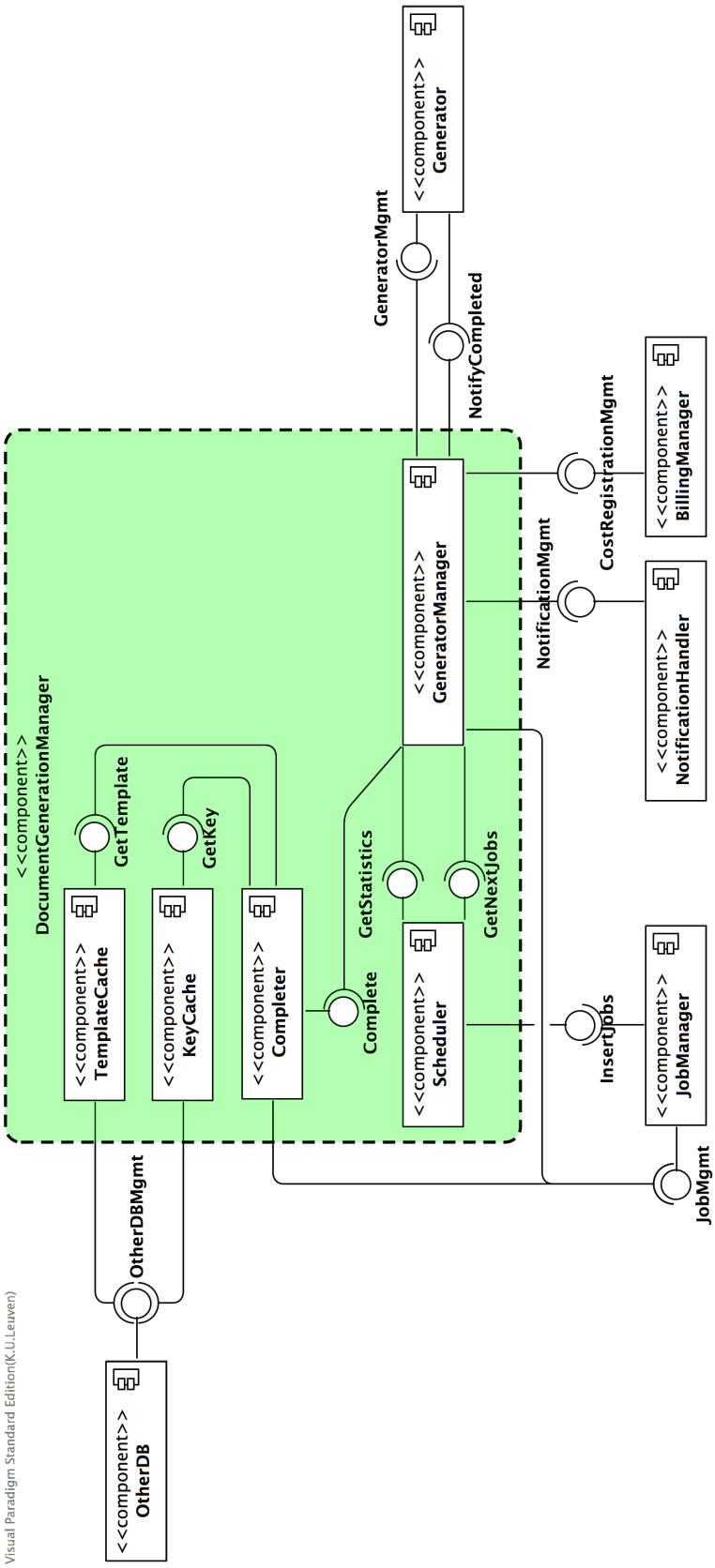

Figure 5: Decomposition of `DocumentGenerationManager`

The decomposition of the `DocumentGenerationManager` is presented in Figure 5. As shown, the `DocumentGenerationManager` is decomposed into five submodules: the `TemplateCache`, the `KeyCache`, the `Completer`, the `Scheduler` and the `GeneratorManager`.

Since this decomposition was already discussed in the provided initial architecture, we will not repeat those results here. Note, however, that some interfaces have been changed, added or even removed for the `DocumentGenerationManager` to be able to properly communicate with the rest of the system (see Appendix A). Also note that, according to *M1*, all potential changes involving a new type of document are encapsulated by both the `DocumentGenerationManager` and the `RawDataHandler`. More specifically, the generic internal structure of the `DocumentGenerationManager` ensures that the generation of new document types will reuse as much of the existing functionality for generating documents as possible.

4.4 DocumentStorageFunctionality

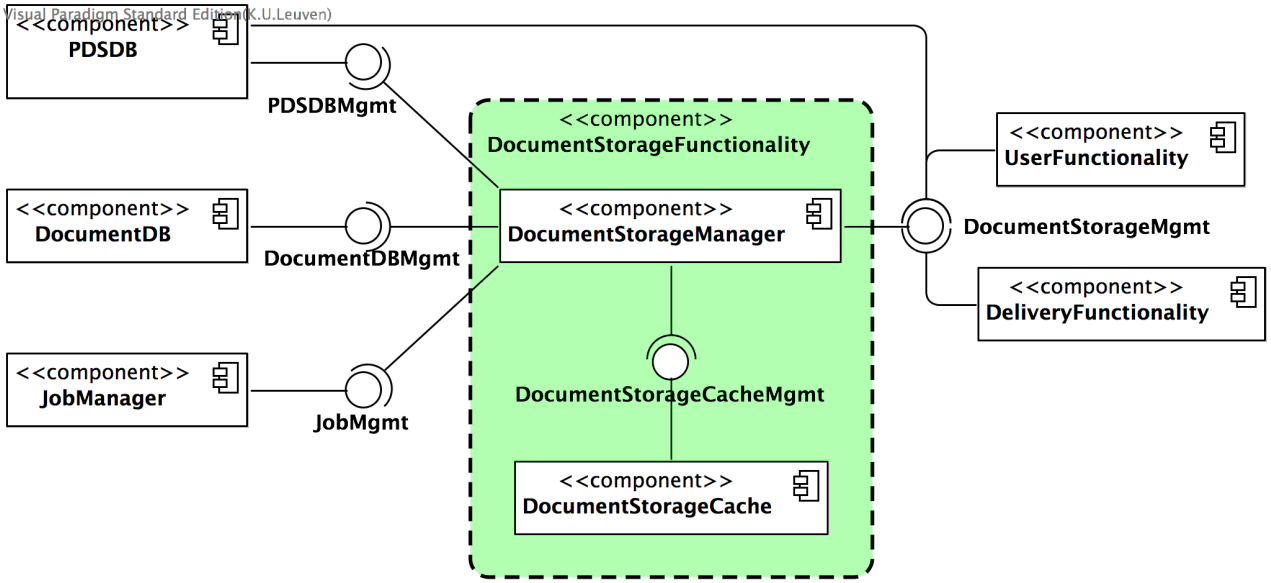

Figure 6: Decomposition of `DocumentStorageFunctionality`

The decomposition of the `DocumentStorageFunctionality` is presented in Figure 6. As shown, the `DocumentStorageFunctionality` is decomposed into only two submodules: the `DocumentStorageManager` and the `DocumentStorageCache`.

The `DocumentStorageManager` is responsible for storing generated documents in the correct database(s) (i.e. the `PDSDB` and the `DocumentDB`) upon the appropriate method call from the `ChannelDispatcher` (see also the decomposition of the `DeliveryFunctionality` in Section 4.1). If the call indicates that the document belongs to a registered recipient, the `DocumentStorageManager` sends a write request to both the `DocumentDB` and the `PDSDB`, accompanied by a call to the `JobManager` to mark the corresponding job as sent. However, if the call indicates that the document belongs to an unregistered recipient, the `DocumentStorageManager` only sends a write request to the `DocumentDB`. Note that in each case, the `DocumentStorageManager` creates the appropriate `DocumentMetaData` (see Appendix B) to store along with the document. Also note that all calls to add a receipt timestamp to the meta data of a document pass through the `DocumentStorageManager` in order to reach both the `DocumentDB` and the `PDSDB` in a controlled manner.

Furthermore, the `DocumentStorageManager` is responsible for transferring an unregistered recipient's documents from the `DocumentDB` to the `PDSDB` along with the converted meta data (i.e. the original `DocumentMetaData`, including the corresponding `RecipientId` but omitting the `EmailAddress` if present) when this recipient registers to the eDocs system, and for deleting a registered recipient's documents from the `PDSDB` when this recipient unregisters. Note that both processes are set in motion by the `RegistrationManager` (see also the decomposition of the `UserFunctionality`).

A major responsibility of the DocumentStorageFunctionality is the temporary storage (i.e. for a maximum of 3 hours) of documents that are to be stored in the PDSDB upon failure of the latter component, according to *Av2b*. To achieve this, the DocumentStorageManager stores the corresponding DocumentId-RecipientId pairs in the DocumentStorageCache during PDSDB downtime (which is detected through a failed write request). Note that all those documents are also stored in the DocumentDB (see the decomposition of the DocumentDB in Section 4.2), so that the DocumentStorageManager is able to transfer all cached documents (i.e. documents for which DocumentId-RecipientId pairs have been cached) from the DocumentDB to the PDSDB upon revival of the latter component (which is detected through a sign-of-life call from the PDSDB itself). In order to successfully complete this transfer, the DocumentStorageManager stores those documents along with their converted meta data (i.e. the original DocumentMetaData, including the corresponding RecipientId but omitting the EmailAddress if present) in the PDSDB. Since the requirements do not specify what happens after 3 hours, the behaviour of the system is undefined in this situation.

4.5 JobManager

The decomposition of the JobManager is presented in Figure 7. As shown, the JobManager is decomposed into three submodules: the JobFacade, the JobDBShardingManager and the JobDBShard.

In order to comply with *Av1b*, the JobManager is responsible for storing all document processing jobs. These jobs are (according to *P3*) partitioned across several JobDBShards that each reside on their own node. In order to further comply with *P3* and thus increase performance of the JobManager and the corresponding speed at which job statuses can be requested, the JobDBShardingManager, which is in charge of managing all JobDBShards and forwarding read and write requests to the appropriate ones, is also deployed on an individual node (see also the deployment view in Section 5).

Note that the JobDBShardManager and corresponding JobDBShards do not distinguish between jobs of different age. Although the requirements for *P3* mention a different minimal response time for each type of job with regard to its age during the construction of a job status overview, our system does not offer this distinction. Instead, the JobDBShardManager and corresponding JobDBShards simply present the customer administrator(s) with a response time that is equal to the lowest of all minimal response times mentioned in the requirements. This strategy was chosen due to the fact that (1) different dynamic storage requirements for jobs along with their periodical synchronization would largely outweigh this simple storage approach with respect to implementation and deployment costs and (2) the same level of performance per type of job can easily be provided by this simple storage approach (i.e. the uniform response time per job should be less than the lowest of all minimal response times) because of the small storage capacity that is needed for job data.

A major responsibility of the JobFacade is forwarding all job requests (e.g. marking a job as sent, received, failed or completed, reading the status of a job, ...) to the JobDBShardingManager. Additionally, all calls to the OtherDB based on a JobId (i.e. the JobId is given as an argument in a method call to the interface) are also routed through the JobFacade. This component then in turn fetches the required data from the OtherDB and sends it back to the original requesting component. Removing this level of indirection would result in those components (from which the routed calls originated) calling the OtherDB directly after having fetched the appropriate job information from the JobDBShardingManager themselves. In order to centralize this functionality and alleviate these components from the burden of calling both the JobDBShardngManager and the OtherDB, we decided to include the JobFacade as a fully-fledged component in our architecture.

Furthermore, the JobFacade is also responsible for creating jobs, a process that is initiated by the RawDataHandler. This process includes storing the corresponding jobs (via the JobDBShardingManager) and inserting them in the DocumentGenerationManager.

4.6 LinkMappingFunctionality

The decomposition of the LinkMappingFunctionality is presented in Figure 8. As shown, the LinkMappingFunctionality is decomposed into only two submodules: the LinkMappingManager and the LinkMappingDB.

The LinkMappingManager is responsible for creating link mappings upon receiving this request from the

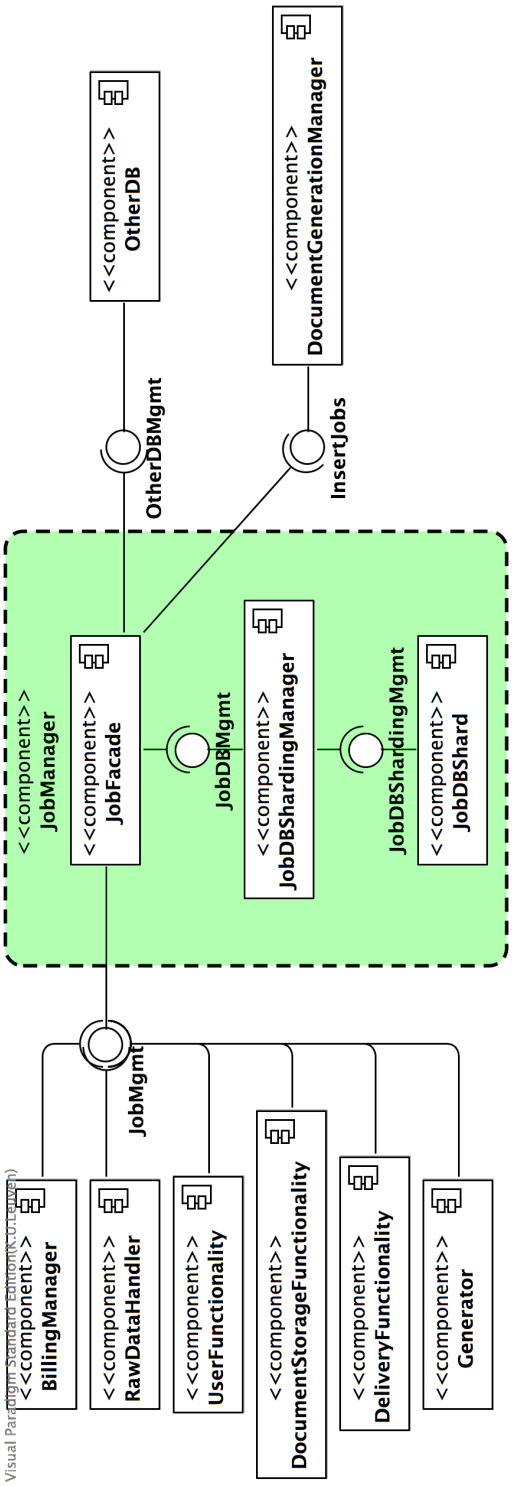

Figure 7: Decomposition of `JobManager`

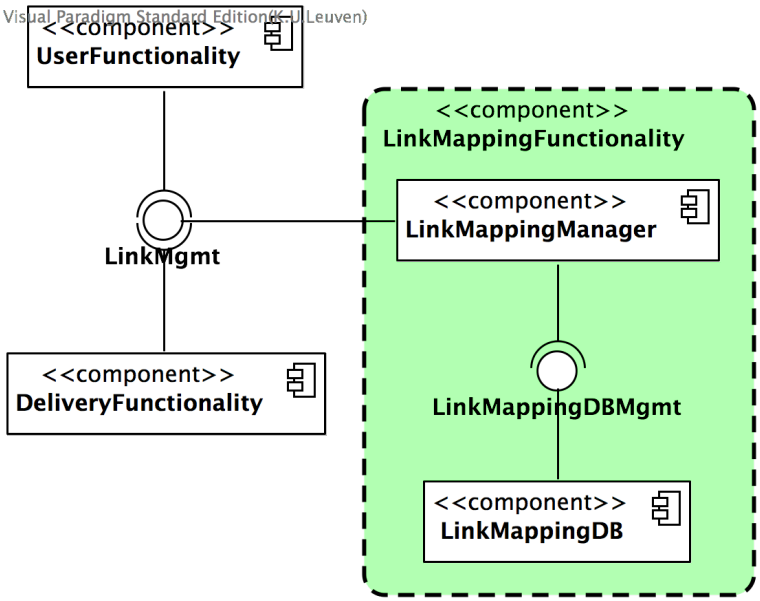

Figure 8: Decomposition of `LinkMappingFunctionality`

`EmailFacade` in the `DeliveryFunctionality` (see the decomposition of the `DeliveryFunctionality` in Section 4.1) and subsequently storing them in the `LinkMappingDB`. More precisely, a link mapping consists of a `Link` and a corresponding `DocumentId-Boolean` pair along with a timestamp, identifying the mapped document, its location (i.e. the `PDSDB` or the `DocumentDB`) and the time at which the mapping was created, respectively.

Upon receiving a link from the `RecipientFacade` (i.e. a document lookup request), the `LinkMappingManager` fetches the appropriate link mapping in the `LinkMappingDB`, after which it determines the location of the corresponding document.

- If the requested document is located in the `DocumentDB` (i.e. the recipient is not registered), the `LinkMappingManager` first checks if the link is not expired (i.e. the corresponding timestamp is older than 30 days). If the link is not expired, the corresponding document is fetched from the `DocumentDB` and sent back to the requesting recipient. However, if the link is expired, this recipient will be presented with the message that his or her link has expired.
- If the requested document is located in the `PDSDB` (i.e. the recipient is registered), the `LinkMappingManager` does not need to check if the link is expired or not. The corresponding document is immediately fetched from the `PDSDB` and sent back to the requesting recipient.

4.7 PDSDB

The decomposition of the `PDSDB` is presented in Figure 9. As shown, the `PDSDB` is decomposed into three submodules: the `PDSLongTermDocumentManager`, the `PDSReplicationManager` and the `PDSDBReplica`.

Since this decomposition was already discussed in the provided initial architecture, we will not repeat those results here. Note, however, that some interfaces have been changed, added or even removed for the `PDSDB` to be able to properly communicate with the rest of the system (see Appendix A).

Also note that, in order to comply with the design decisions that were made in the light of *Av2b*, the `PDSDB` should notify the `DocumentStorageManager` (see also the decomposition of `DocumentStorageFunctionality` in Section 4.4) upon revival of the former after a period of downtime. This responsibility is delegated to the `PDSLongTermDocumentManager` by having it send a sign of life to the `DocumentStorageManager` upon revival of the `PDSDB`.

Finally, *P2* requires the `PDSDB` to be able to throttle excessive read requests from the `PDSFacade`, through

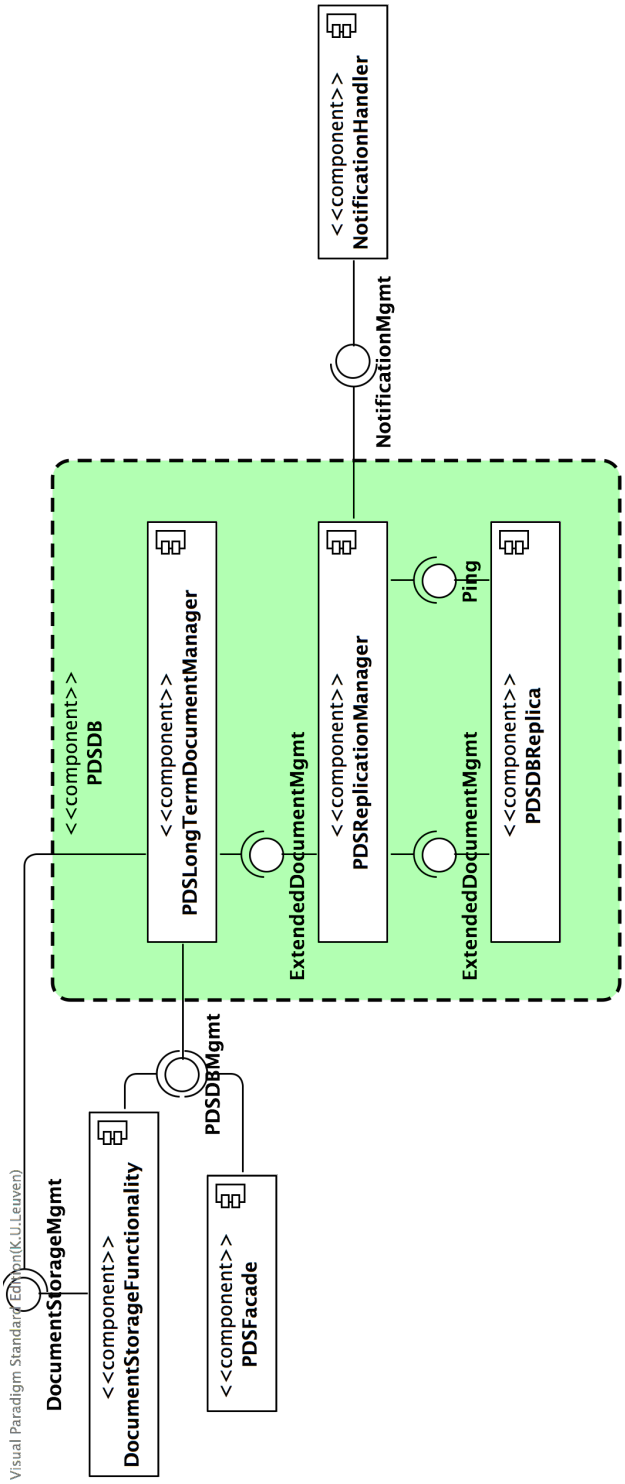

Figure 9: Decomposition of `PDSDB`

which recipients' PDSDB requests are routed, when the arrival rate is larger than a certain value that is specific to the kind of request (determined by the requirements). This responsibility is delegated to the PDSLongTermDocumentManager, the only component in the document lookup pipeline that has knowledge of all incoming PDSDB requests, to ensure that other PDSDB requests are also taken into account in order to not hinder the other functionality of the system. Note that non-excessive recipient requests to the PDSDB continue to be handled in a timely fashion during this throttling, as is dictated by *P2*.

4.8 UserFunctionality

The decomposition of the UserFunctionality is presented in Figure 10. As shown, the UserFunctionality is decomposed into six submodules: the RecipientFacade, the CustomerOrganizationFacade, the EDocsAdminFacade, the AuthenticationHandler, the SessionDB and the RegistrationManager.

The UserFunctionality is responsible for the interaction between (un)registered recipients, customer organizations and eDocs operators on the one side and the eDocs system on the other side. More precisely, the interactive functionality between the eDocs system and each of these three user groups is split up into three corresponding facade components: the RecipientFacade, the CustomerOrganizationFacade and the EDocsAdminFacade.

According to *Av1a* & *Av2a*, the EDocsAdminFacade is to be deployed on its own node, along with the NotificationHandler. The former component is responsible for communicating with the external EDocsAdminClient. More precisely, it handles all incoming requests (i.e. authentication of an eDocs operator and (un)registration of a customer organization) and notifies the EDocsAdminClient in a timely fashion in case of an emergency. Note that the EDocsAdminFacade is also responsible for logging out a customer organization before unregistering it.

In order to comply with *P3*, the CustomerOrganizationFacade is also deployed on its own node. This component is responsible for offering a customer organization the ability to authenticate itself, to send in raw data, to update its templates, to enable or disable receipt tracking and to request job status overviews, all by means of communicating with the external CustomerOrganizationClient and sending corresponding requests to the JobManager (for job status overviews), the OtherDB (for template and receipt tracking management), the AuthenticationHandler (for authentication) and the RawDataHandler (for raw data insertion).

The RecipientFacade, which is also deployed on its own node together with the LinkMappingFunctionality according to *P2*, is responsible for offering a recipient the ability to authenticate or register him- or herself (by calling the AuthenticationHandler or the RegistrationManager respectively) and to request a document. In the latter case, there are three possible situations:

- If the recipient is unregistered, he or she presents the RecipientFacade with a unique link, upon which this component fetches the id and location of the requested document by calling the LinkMappingFunctionality. After subsequently fetching the corresponding document in the DocumentDB, the RecipientFacade presents it to the requesting recipient via the external RecipientClient.
- If the recipient is registered and has presented the RecipientFacade with a link, this component again fetches the id and location of the requested document by calling the LinkMappingFunctionality. This time, however, the RecipientFacade also requests the respective recipient to log in (because the requested document is now located in the recipient's personal document store) before fetching the corresponding document via the PDSFacade and subsequently presenting it to the requesting recipient via the external RecipientClient.
- If the recipient is registered and was presented with an overview of his or her personal document store at an earlier step in the document lookup process, he or she is able to request any document in that overview directly by calling the RecipientFacade. This component subsequently fetches this document by calling the PDSFacade and presents it to the requesting recipient via the external RecipientClient.

Furthermore, the RecipientFacade is able to present a registered recipient with an overview of (part of) his or her personal document store upon receiving a query from this recipient. Finally, after having presented the recipient with his or her requested document, the RecipientFacade is also responsible for marking the corresponding document and job as received (by calling the DocumentStorageFunctionality and the JobManager respectively). This responsibility is delegated to the RecipientFacade because of the fact that this component

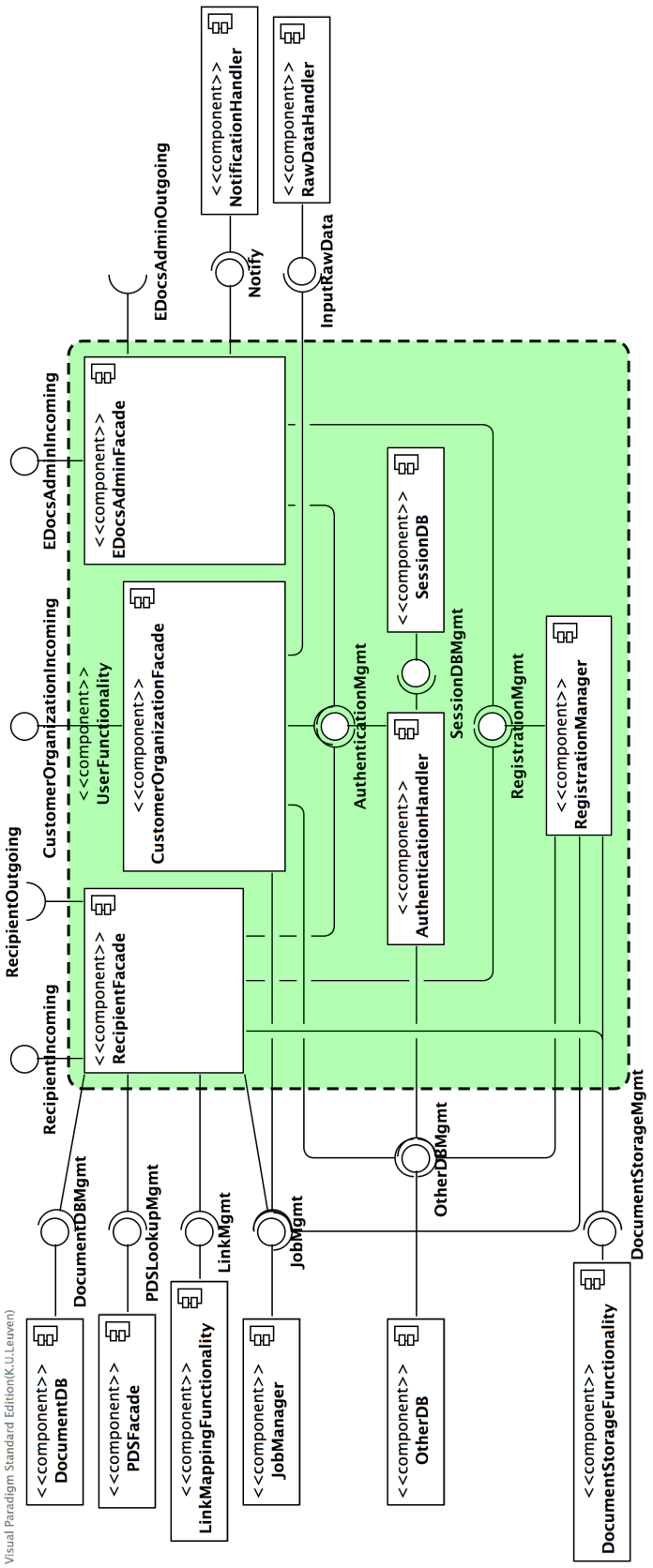

Figure 10: Decomposition of `UserFunctionality`

is the last one in the document lookup pipeline before the requested document crosses the system boundary.

The `AuthenticationHandler` is responsible for authenticating registered recipients and customer organizations upon receiving the proper request from the `RecipientFacade` and the `CustomerOrganizationFacade` respectively. For the `AuthenticationHandler` do its job, it needs to communicate with the `OtherDB` (for fetching user credentials) and the `SessionDB` (for session management). The latter component is solely responsible for storing user sessions and handling all requests from the `AuthenticationHandler` concerning these sessions. Furthermore, in order to comply with *P2* and *P3* by preventing the authentication subprocess from introducing a bottleneck in either the document lookup process (*P2*) or the status overview process (*P3*), both the `AuthenticationHandler` and the `SessionDB` are deployed on the same individual node.

Finally, the `RegistrationManager` is responsible for registering a recipient (upon which it sends a request to the `DocumentStorageFunctionality` to create a new personal document store), unregistering a recipient (upon which it sends a request to the `DocumentStorageFunctionality` to invalidate the recipient's personal document store after having checked if all included documents were received), registering a customer organization and unregistering a customer organization. In each case, the `RegistrationManager` calls the `OtherDB` to store (upon registration) or invalidate (upon unregistration) the respective user credentials. Note that the `RegistrationManager` does not allow a recipient to unregister if he or she has not yet received (i.e. read) all documents in his or her personal document store (as dictated by the requirements).

5 Deployment view (UML Deployment diagram)

The context diagram of the deployment view is given in Figure 11. As shown, four nodes are connected to the outside world: the `EDocsAdminNode`, the `CustomerOrganizationNode`, the `RecipientNode` and the `ResidualNode`. Notice that there will be multiple `CustomerOrganizationTerminals`, `RecipientTerminals` and possibly `EDocsAdminTerminals` communicating with the eDocs system simultaneously.
Due to its very nature, the `EmailChannel` communicates with the eDocs system by means of an SMTP protocol. Both the `ZoomitChannel` and the `Print&PostalServiceChannel`, however, are connected to the eDocs system via a TCP protocol, because this connection only needs to support simple data transfer. For the same reason, both the `EDocsAdminTerminal` and the `CustomerOrganizationTerminal` support a TCP connection to the internal `EDocsAdminNode` and `CustomerOrganizationNode` respectively. Furthermore, the `EDocsAdminTerminal`, the `CustomerOrganizationTerminal` and the `RecipientTerminal` all make use of an HTTP protocol to be able to communicate with their corresponding internal node in a user friendly manner.
The entire deployment diagram of the eDocs system is presented in Figure 12. As shown, the deployment diagram employs both top-level components as sub-components of further decompositions. If a top-level component is deployed on a node, this means that all its sub-components are deployed there as well. Notice that for readability reasons, we only show multiplicities differing from 1. Also notice that we did not specify protocols for internal connections.

Non-functional requirements

The deployment shown in Figure 12 is crucial for achieving some of the non-functional requirements. More specifically:

- For *Av1* and *P1*, it is important that the `Generators` are spread across several nodes. Since this was already discussed in the provided initial architecture, we will not go into it any further here.
- For *Av2*, it is important that the `PDSDBReplicas` are spread across several nodes (i.e. one group residing on `PDSDBReplicaNotDeliveredNodes` and the other residing on `PDSDBReplicaDeliveredNodes`) and that the `PDSLongTermDocumentManager` and the two `PDSReplicationManagers` are deployed altogether on their own separated node. Since this was already discussed in the provided initial architecture, we will not go into it any further here. Note, however, that the `PDSFacade` is deployed on the same `PDSDBManagerNode` in order to increase performance for *P2*.
  Av2 also requires that the `DocumentStorageCache` (which is a subcomponent of the `DocumentStorageFunctionality`) is not deployed together with any of the previously mentioned `PDSDB` subcomponents to be able to temporarily store all documents (destined for the `PDSDB`) in case of failure of the `PDSDB` as a whole.

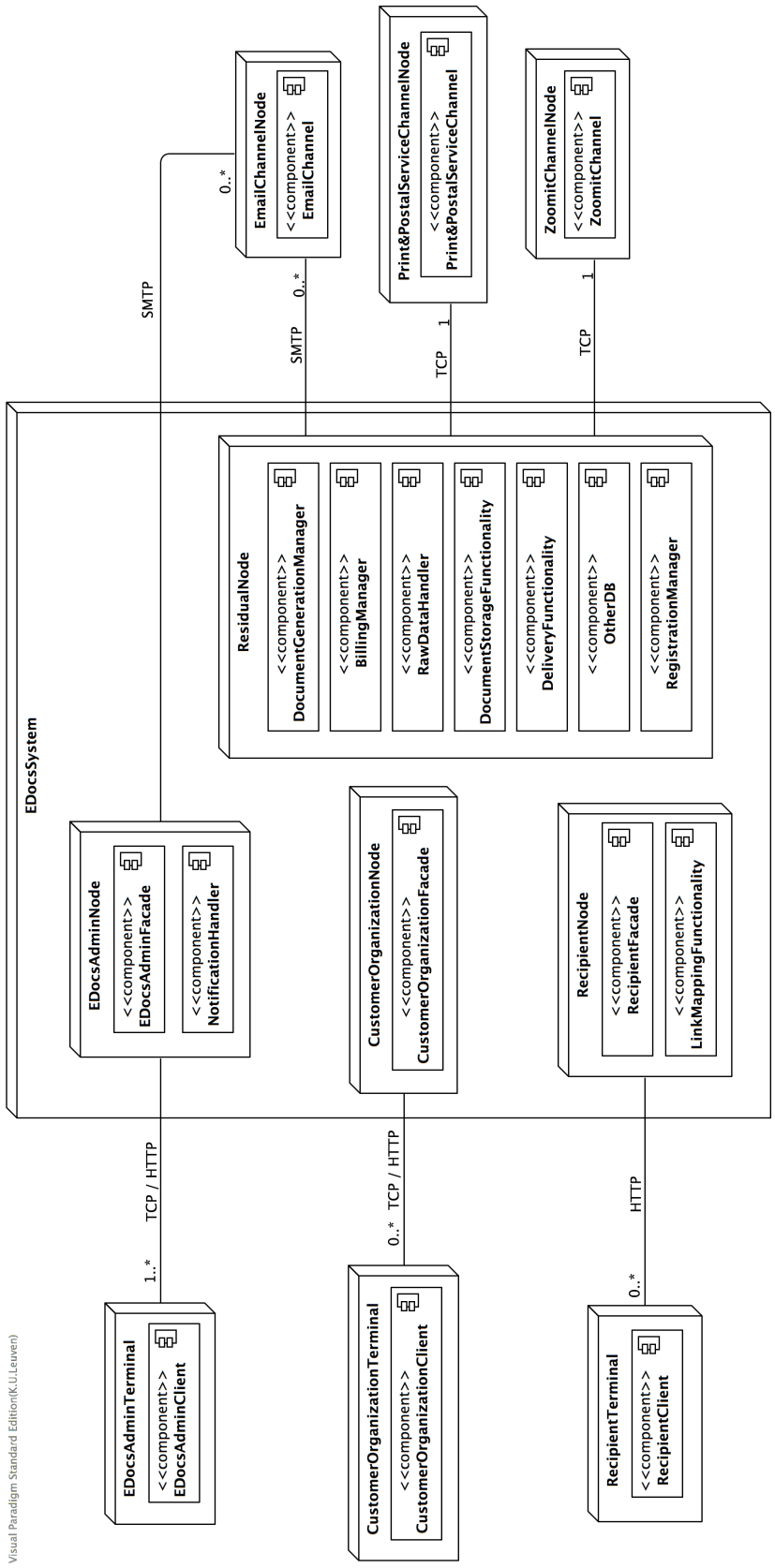

Figure 11: Context diagram for the deployment view.

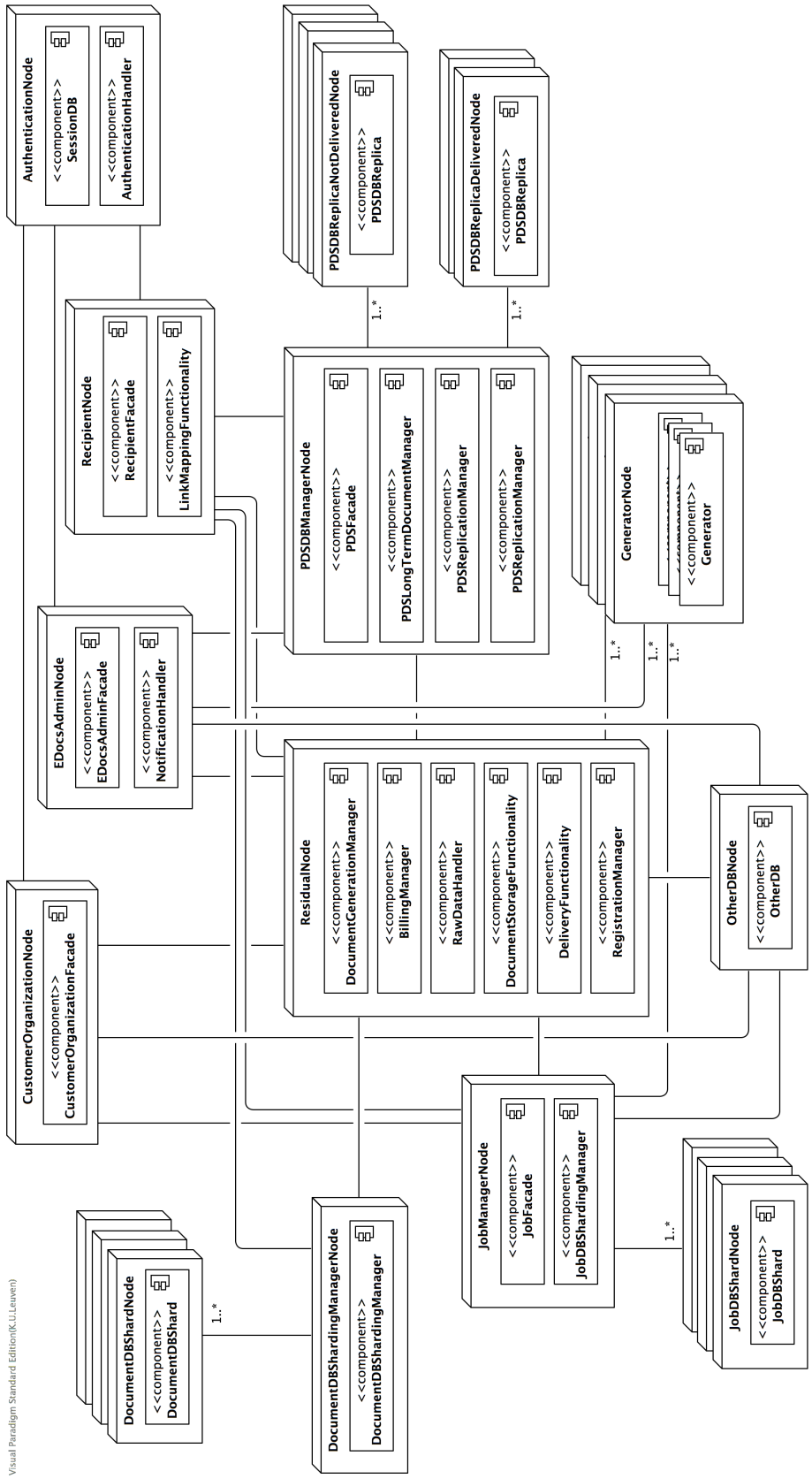

Figure 12: Primary diagram for the deployment view.

- For *Av1* and *Av2*, it is important that notifications (originating either from the `DocumentGenerationManager` or from the `PDSDB`) get sent to the eDocs operator in a timely fashion, both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated from all other nodes to increase communication performance between the two as well as individual performance of both components separately.

- For *P2*, it is important that all potential bottleneck components that support the document lookup process are deployed on separate nodes. More precisely, the `RecipientFacade` and the `LinkMappingFunctionality` both reside on the same exclusive node as to increase the performance of these components and the communication between them (i.e. when the RecipientClient has presented the former component with a link for which it has to determine the document it maps to). Furthermore, the `AuthenticationHandler` and the `SessionDB` are also deployed on the same individual node as to not let the authentication subprocess decrease the performance of the document lookup process. The `PDSFacade` is deployed on the `PDSDBManagerNode`, along with the `PDSLongTermManager` and the two `PDSReplicationManagers`, as to increase communication performance between this `PDSFacade` and the `PDSDB` supercomponent. Notice that *Av1* already demands the `PDSDB` to be deployed separate from all other functionality and that *P2* can rely on this design decision to deliver an optimally increased (taking the availability constraints of *Av1* into account) performance of this supercomponent.
  Furthermore, because the `DocumentDB` stores a large amount of documents and correspondingly handles a large amount of document requests, its documents are partitioned across several `DocumentDBShards` that each reside on their own `DocumentDBShardNode`. This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is significantly less costly regarding storage costs. In order to further increase performance of the `DocumentDB`, the `DocumentDBShardingManager` is also deployed on an individual node.
  Finally, *P2* requires the performance of all other functionality of the system to remain unaffected in case of a large number of incoming document requests. Therefore, all potential bottleneck components that support the document lookup process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the document lookup process and a sufficient level of performance for all other functionality.

- For *P3* it is important that all potential bottleneck components that support the status overview process are deployed on separate nodes as to increase their performance. More precisely, both the `AuthenticationHandler` and the `SessionDB` are deployed on the same individual node as to not let the authentication subprocess decrease the performance of the status overview process. Furthermore, the `CustomerOrganizationFacade` and the `JobManager` supercomponent are also deployed separate from all other components in the system. More specifically, the former occupies a single node by itself, while the latter is responsible for storing all jobs, which are partitioned across several `JobDBShards` that each reside on their own `JobDBShardNode` too. This sharding technique results in a performance increase that is statistically equivalent to the one that is achieved by active replication, but it is less costly regarding storage costs. In order to further increase performance of the `JobManager` and the corresponding speed at which job statuses can be requested, the `JobDBShardingManager` is also deployed on an individual node.
  Finally, *P3* requires the performance of all other functionality of the system to remain unaffected during the construction of such a status overview. Therefore, all potential bottleneck components that support this status overview process are deployed on separate nodes, effectively increasing their performance (as discussed above) and thereby providing both an increased level of performance for the status overview process and a sufficient level of performance for all other functionality.

- For performance reasons, we chose to also deploy the `OtherDB` on its own individual node due to the fact that this component stores a fair amount of data that needs to be accessed regularly (e.g. raw data, credentials, ...).

Alternatives considered

Alternative for joint deployment of NotificationHandler and EDocsAdminFacade In the current architecture, both the `EDocsAdminFacade` and the `NotificationHandler` are deployed on the same node, which is separated from all other nodes to increase overall performance. An alternative for this strategy would be to deploy the `NotificationHandler` on its own node, which would increase the individual performance of this component. Unfortunately, this would also decrease the communication performance between this component and the `EDocsAdminFacade`. Since *Av1a* and *Av2a* only dictate an eDocs operator to receive his or

her notifications in a timely fashion and the increase in individual performance of the `NotificationHandler` would not outweigh the decrease in communication performance with the `EDocsAdminFacade` in this case, we chose to go with our original approach and deploy both components on the same node.

Alternative for separate deployment of potential bottleneck components in crucial processes In our current architecture, all potential bottleneck components that support crucial processes (i.e. the document lookup process and the status overview process) are deployed on separate nodes. An obvious alternative would be to deploy these components on the same node (assuming this would not contradict with any design decisions that were made for other requirements). Although this approach would greatly increase communication performance between all components involved, it would also decrease overall performance of the document lookup process and, more importantly, of the other functionality in the system. For these reasons, we decided to deploy these components on separate nodes as discussed above.

6 Scenarios

In this section, we illustrate the internal behaviour for our system for the most important data flows, i.e. the authentication of the Registered Recipients and the whole document processing flow.
We describe the authentication of Registered Recipients in Sections 6.2 and 6.2.
The document processing flow is covered in multiple sections. The initiation of the document processing upon the delivery of raw data is explained in section 6.6. Section 6.1 recaps the generation of documents as already explained in the initial architecture, but adapts it to the extended architecture. Section 6.9 and its subsections describe the different channels through which a document can be delivered.

Other data flows that are covered are the consultation of the personal document store by a Registered Recipient in Section 6.10, the consultation of a document in the personal document store of a Registered Recipient in Section 6.7. Section 6.8 explains how a document can be downloaded by following a unique link and Section 6.11 describes how an Unregistered Recipient can register himself or herself to get a personal document store.

6.1 Document generation

The generation of documents has already been discussed in the provided initial architecture during the decomposition with as non-functional requirements *Av1: Document generation failure* and *P2: Document generation.* Figures 13 and 14 originate from the original architecture, but are adapted to fit the final architecture. Therefore, we will not give an in depth dicussion of these figures, but focus on the main differences.

Figure 13 shows a `Generator` instance that requires new jobs to process contacting the `GenerationManager`. The `GenerationManager` then asks the `Scheduler` for job identifiers of jobs that need to be processed. The `Scheduler` returns a list of `JobIds` belonging to the same batch (but not necessarily all the jobs of that batch). Next, the `Generator` asks the `Completer` to return all the data necessary to generate the documents. This is data belonging to the jobs and data belonging to the batch of these jobs. The `Completer` first fetches the raw data for the jobs from the `OtherDB` using the `JobFacade`. The `JobFacade` can fetch these raw data entries because a job contains the identifier of the necessary raw data entry.
After the raw data entries, the `Completer` fetches the meta-data of the batch, the cryptographic key of the customer organization initiating the processing job and the template for the document type to be generated. The batch meta-data, key and template are all stored in the `OtherDB`. The batch meta-data contains the customer organization identifier, the time when the batch was received by the eDocs system, whether the batch is recurring, the priority of batch ... (see also the data type `BatchMetaData` in Appendix A).

Figure 14 shows the continuation of figure 14. After fetching the template, the `Completer` returns the list of raw data entries and their job identifiers, together with the batch meta-data, the key and the template to the `GeneratorManager`. The `GeneratorManager` assigns these document processing jobs to a `Generator` instance asking for new processing jobs, giving it this necessary data.
For each received job, the `Generator` instance tries to generate a document. If it succeeds, it tells the `BillingManager` to add the appropriate cost to the bill of the customer organization and forwards the job identifier and document to the `ChannelDispatcher`. The `ChannelDispatcher` will then initiate the delivery of the document, as explained in Section 6.9. If the `Generator` instance fails to generate the document because

it does not have all of the data necessary to fill in the template, it will notify the appropriate Customer Organization Administrator by sending an e-mail with a description of the error using the `NotificationHandler`. The `NotificationHandler` gets the e-mail address from the SLA in the `OtherDB`.

6.2 A registered recipient logs in

Figure 15 depicts the sequence diagram of a Registered Recipient logging in, according to *UC1: Log in*. The Registered Recipient provides his details to the `RecipientFacade`. The `Authentication` compares these credentials to those stored in the `OtherDB`. if the given credentials match the stored credentials, a new session is opened in the `SessionDB` and the corresponding session identifier is returned to the registered recipient. Otherwise, an InvalidCredentialsException is thrown.
Note that the use case *UC1: Log in* also has the Customer Organization as a primary actor. The login procedure for customer organizations is almost identical to the procedure for registered recipient, which is why we do not provide a separate sequence diagram for this primary actor. Compared to figure 15, the Customer Organization sends it login request to the `CustomerOrganizationFacade` instead of the `RecipientFacade`. Also, another method call to `OtherDB` is used to ask for the credentials of a customer organization.

6.3 A registered recipient logs out

Figure 16 shows the behaviour of a Registered Recipient logging out, according to *UC2: Log out*. The Registered Recipient provides his recipient id and session id to the `RecipientFacade`. The `AuthenticationHandler` first verifies the session (figure 16). If the session is valid, it logs out the Registered Recipient.
Note that the logout procedure for Customer Organizations is almost the same. The only difference is that the Customer Organization sends his logout request to the `CustomerOrganizationFacade` instead of the `RecipientFacade`. Because of this reason, we do not give a separate sequence diagram for the Customer Organization logging out.

6.4 Updating a document template

Figure 17 shows the flow when a Customer Administrator updates the template used for generating documents of a given type. It corresponds to *UC20: Update document template*. It consists of two main parts. First, the customer administrator asks what the possible document types are that the customer organization is allowed to generate. The system verifies whether the session of the customer administrator is valid (as detailed in figure 18 and then returns a list of the allowed document types using the `OtherDB`. Together with the allowed document types, it returns the time when the currently stored templates corresponding to the document types were uploaded.
Secondly, the customer administrator provides the document type it wants to update and the new template to the `CustomerOrganizationFacade`. After verifying the validity of the session again (figure 18), the `CustomerOrganization` asks the `OtherDB` to check whether the document type provided by the customer organization is valid and allowed. If it is not allowed or invalid, an InvalidDocumentTypeException gets thrown. Otherwise, the `CustomerOrganizationFacade` stores the template in the `OtherDB` together with the time when it has received the template, the document type and the customer organization identifier.

6.5 Verifying a session

Figure 18 how it is verified whether a session is valid. The `AuthenticationHandler` uses the `SessionDB` to verify whether each incoming session identifier belongs to an existing session. If it belongs to an existing session, the corresponding session attributes are returned to the caller, otherwise a NoSuchSessionException is thrown.

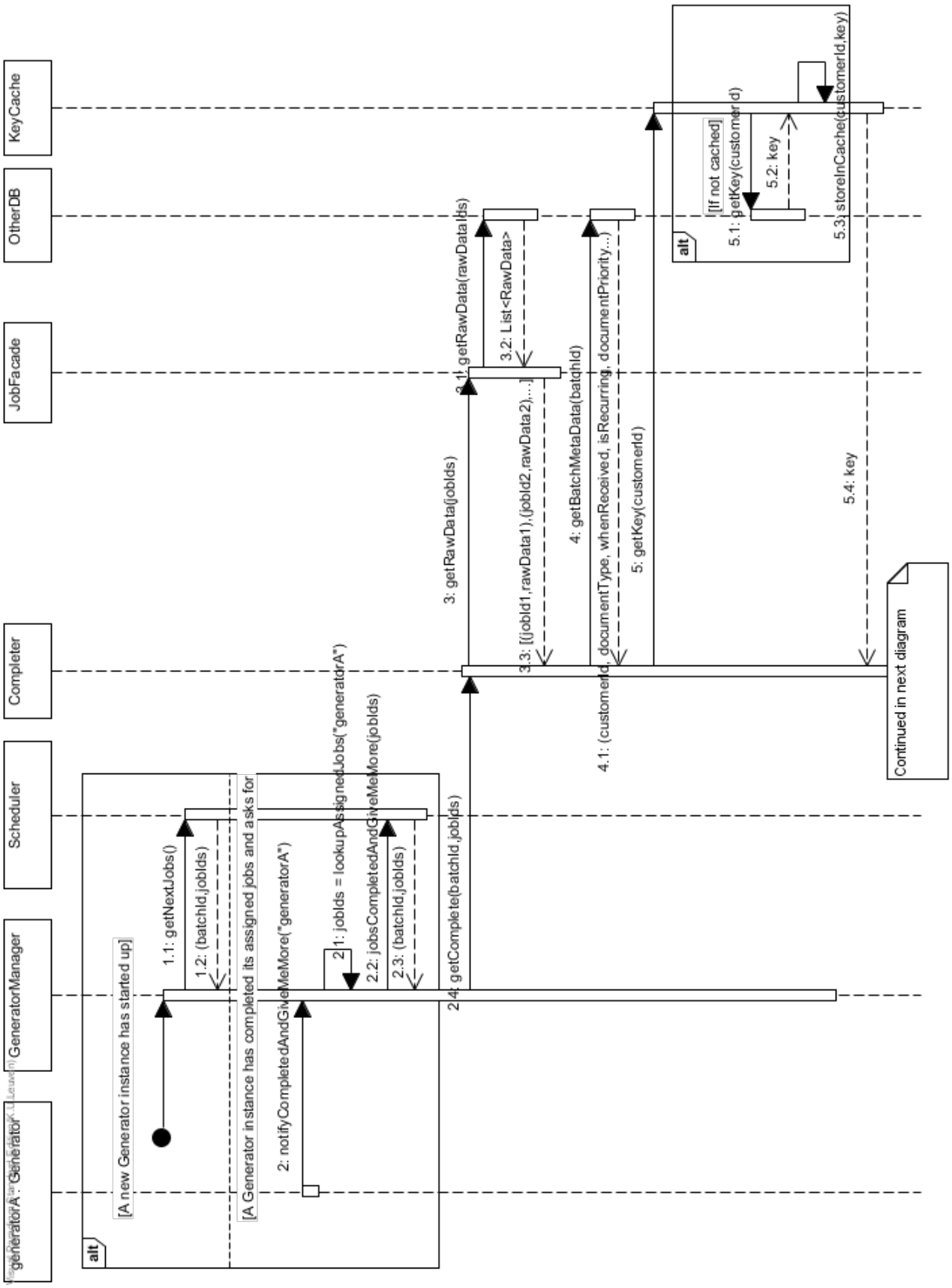

Figure 13: The generation of documents (part 1). Figure 14 shows the continuation of this figure.

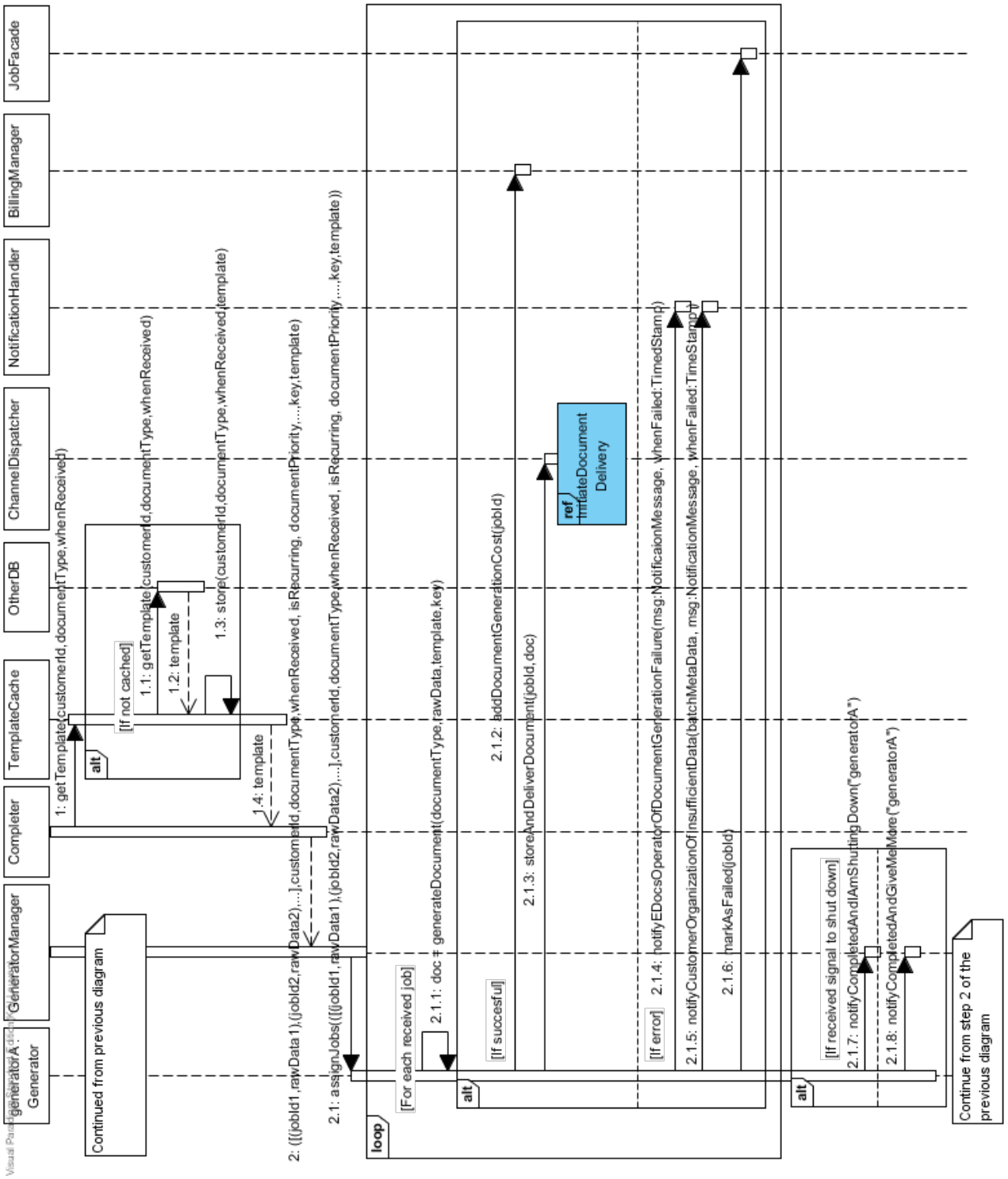

Figure 14: The generation of documents (part 2). This figure is the continuation of figure 14.

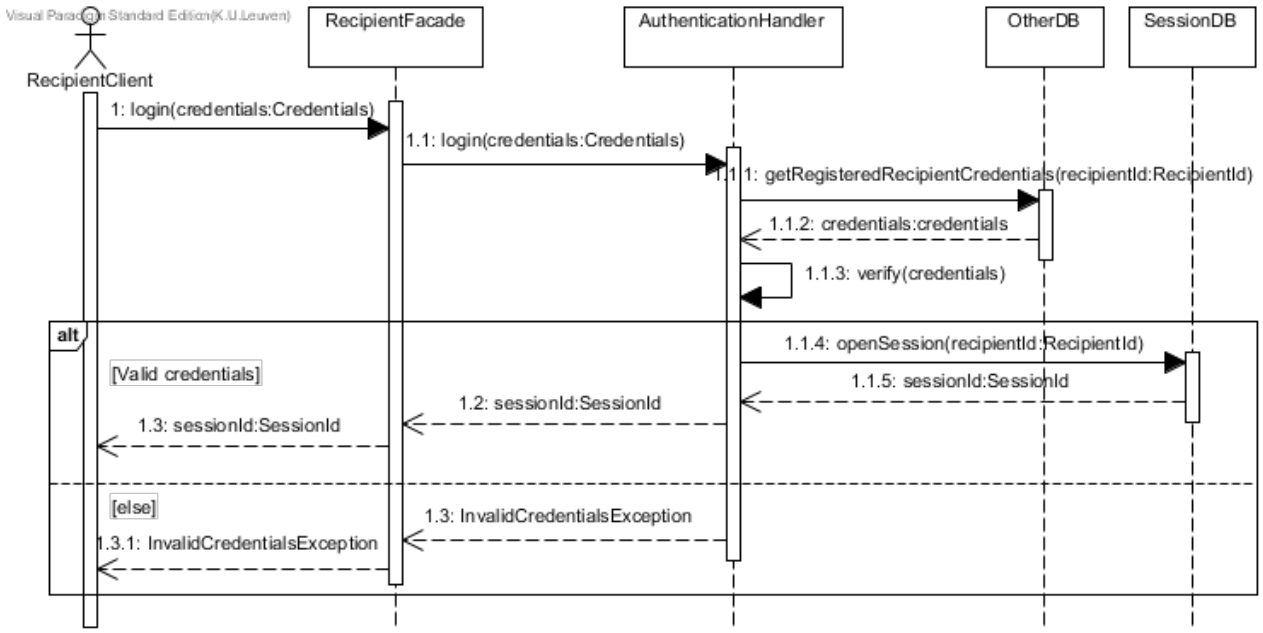

Figure 15: The login behaviour of a Registered Recipient.

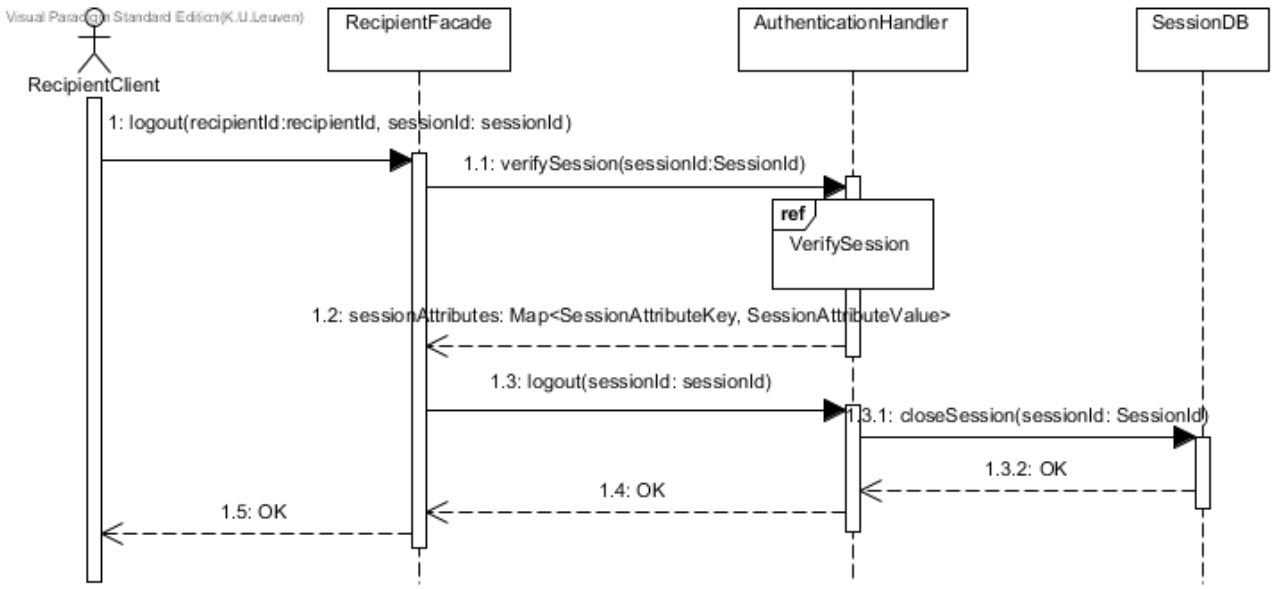

Figure 16: The logout behaviour of the Registered Recipient.

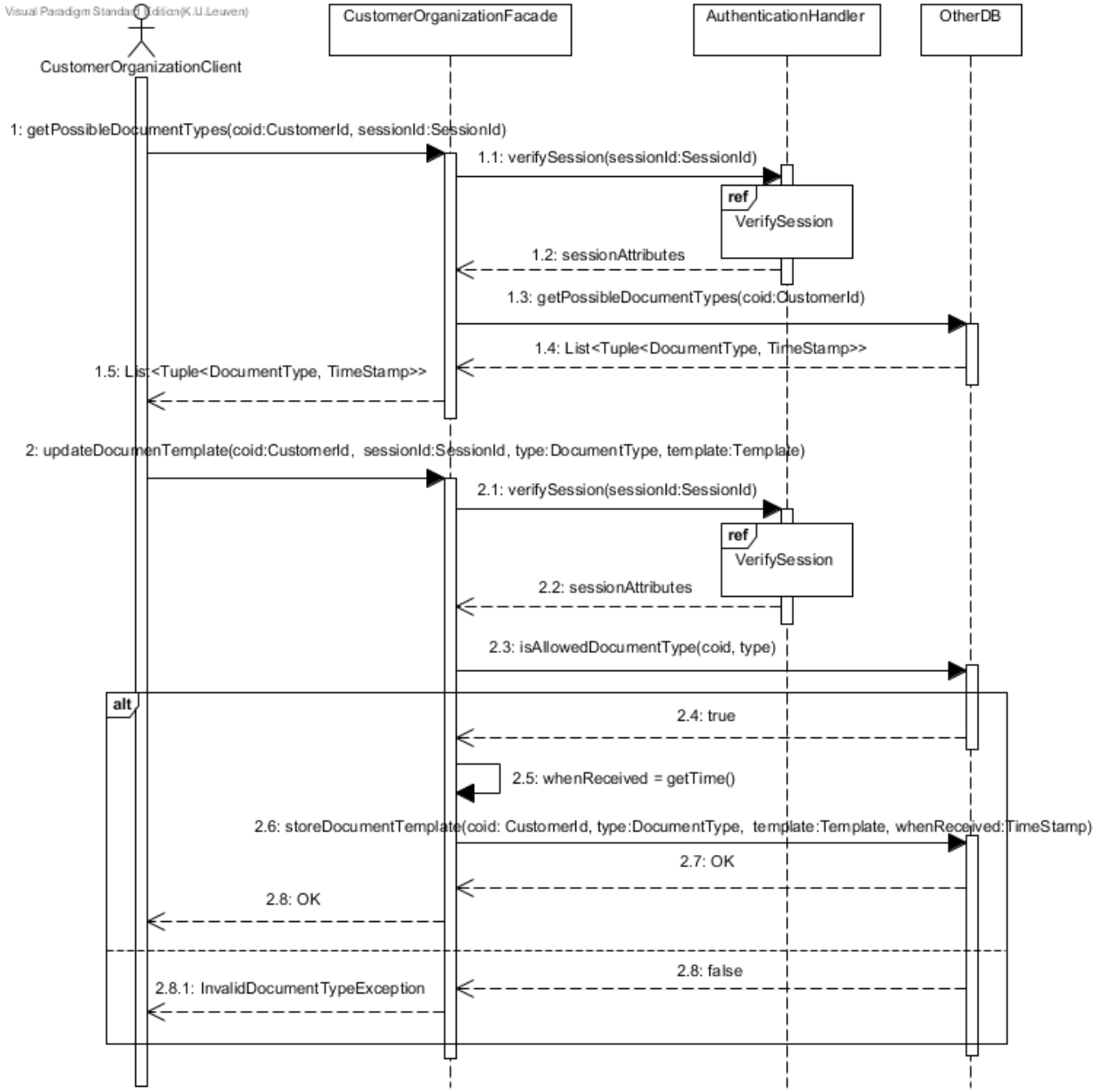

Figure 17: The system behavior for updating a document template.

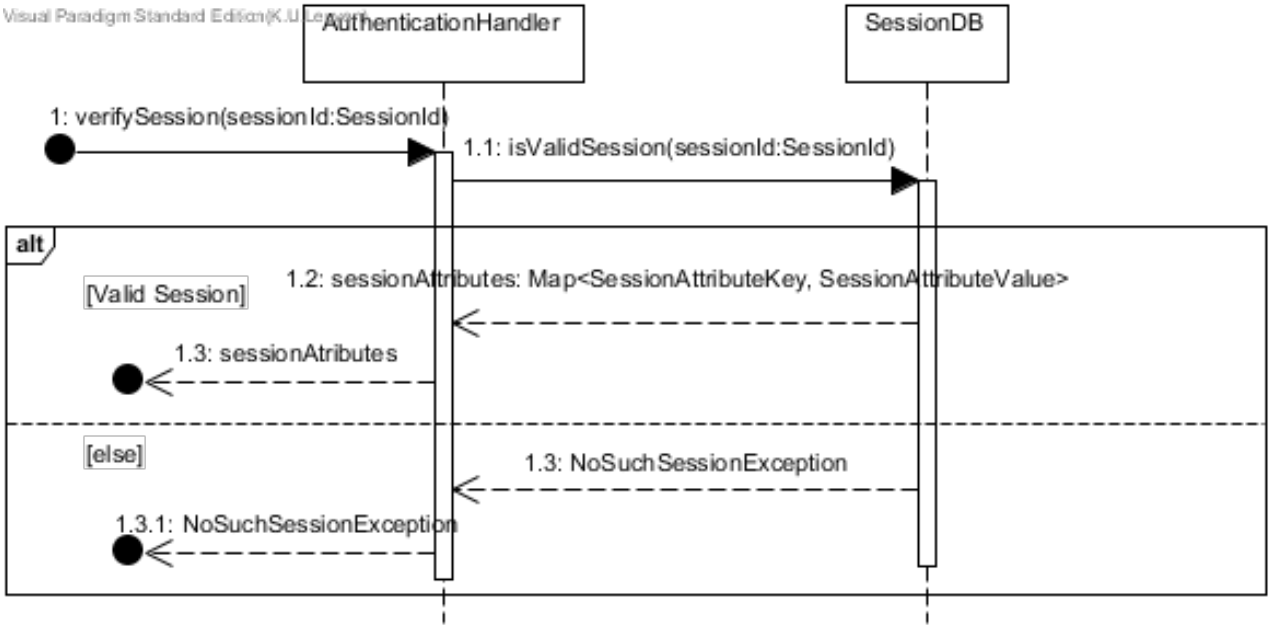

Figure 18: verifySession: sequence diagram depicting the verification whether a session is valid.

6.6 Initiating document processing

Figure 19 shows how a customer organization initiates a batch of document processing jobs. Recurring and non-recurring batches are indicated using the `isRecurring` boolean value. If the batch is non-recurring, the customer organization can indicate another priority than the default priority of that customer organization if the batch should be handled with another priority than the default priority. For recurring batches, it does not matter if a priority is given, as the default priority will be used. The customer organization also delivers the raw data as a `RawDataPackage`. After verification of the session identifier (figure 18), the `CustomerOrganizationFacade` generates a `TimeStamp` of the time when this method is called and forwards this together with all its arguments except for the session identifier to `RawDataHandler`, while also indicating if the batch is (non-)recurring.

Figure 20 depicts the first part of the method call in the `RawDataHandler`, which will verify whether the raw data is valid. First, `RawDataHandler` checks whether the customer organization is allowed to generated the given document type. If it is allowed, the received raw data packet will be validated, e.g there will be checked whether the content of the received Excel file can be read or whether a received XML file is correctly formatted. If it is not valid, an exception is thrown. Otherwise, depending on if the batch is recurring, the `RawDataHandler` will query the SLA in the `OtherDB` to check if the number of raw data entries in the raw data package is allowed. If the number of entries exceeds this maximum, an exception is thrown. Next, each individual entry will be checked on validity, so the `RawDataHandler` knows which `RawData` entries are valid and which are not.

After verification, figure 19 shows that the `RawDataHandler` will only use the given priority if the batch is non-recurring. Otherwise, it will use the default priority. The deadline of the document processing jobs is calculated and a `BatchMetaData` object is generated, bundling the information about the batch. The `RawDataHandler` will store the `RawData` entries and `BatchMetaData` in the `OtherDB`. It asks the `JobFacade` to create jobs for the valid raw data entries and to initiate document processing for those jobs. Finally, it will return the invalid raw data entries to the Customer Organization.

Figure 21 shows the `JobFacade` creating and storing jobs for the valid raw data entries, after which it inserts these jobs into the `Scheduler`.

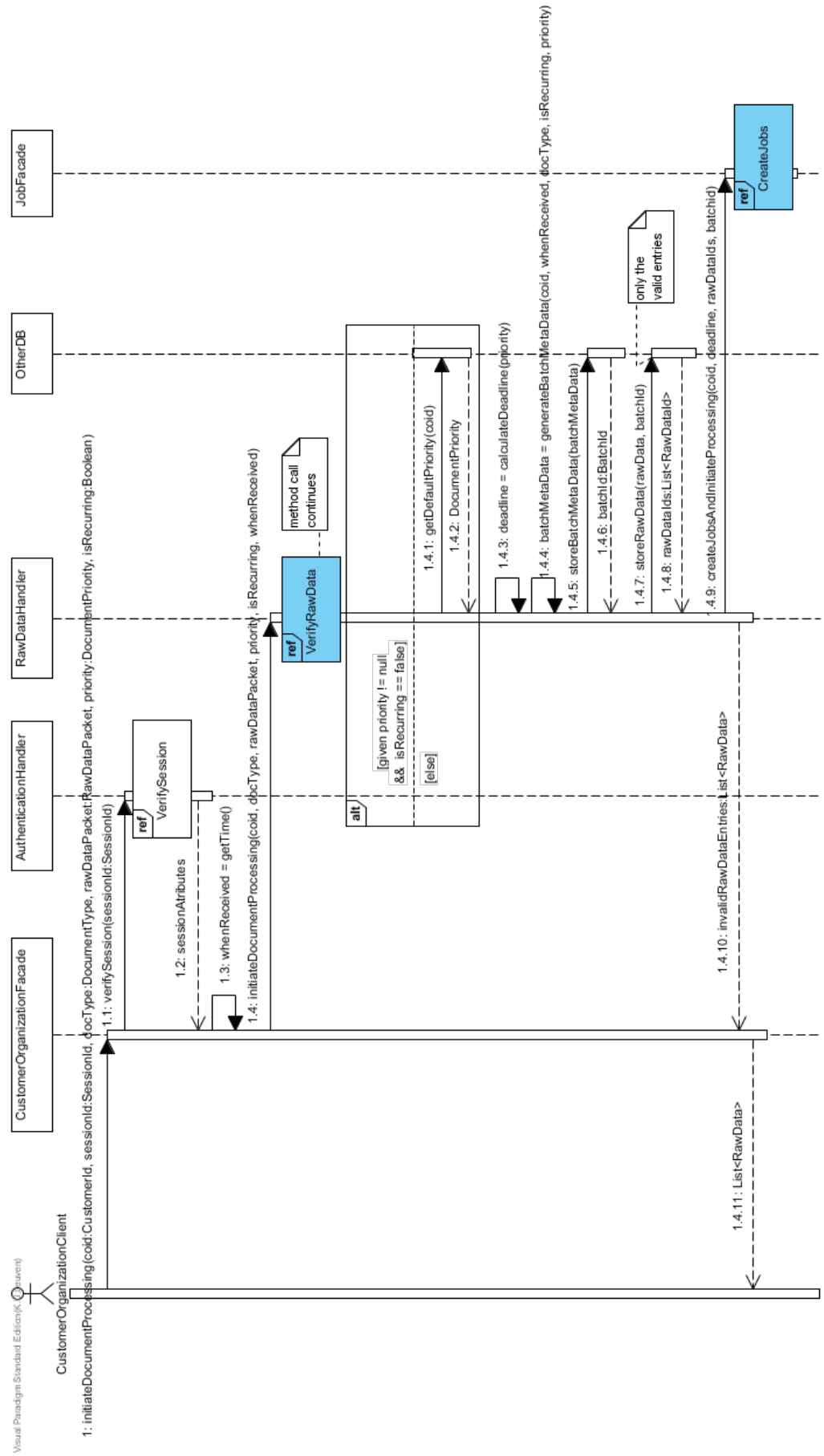

Figure 19: VerifyRawData: verification whether the raw data is valid.

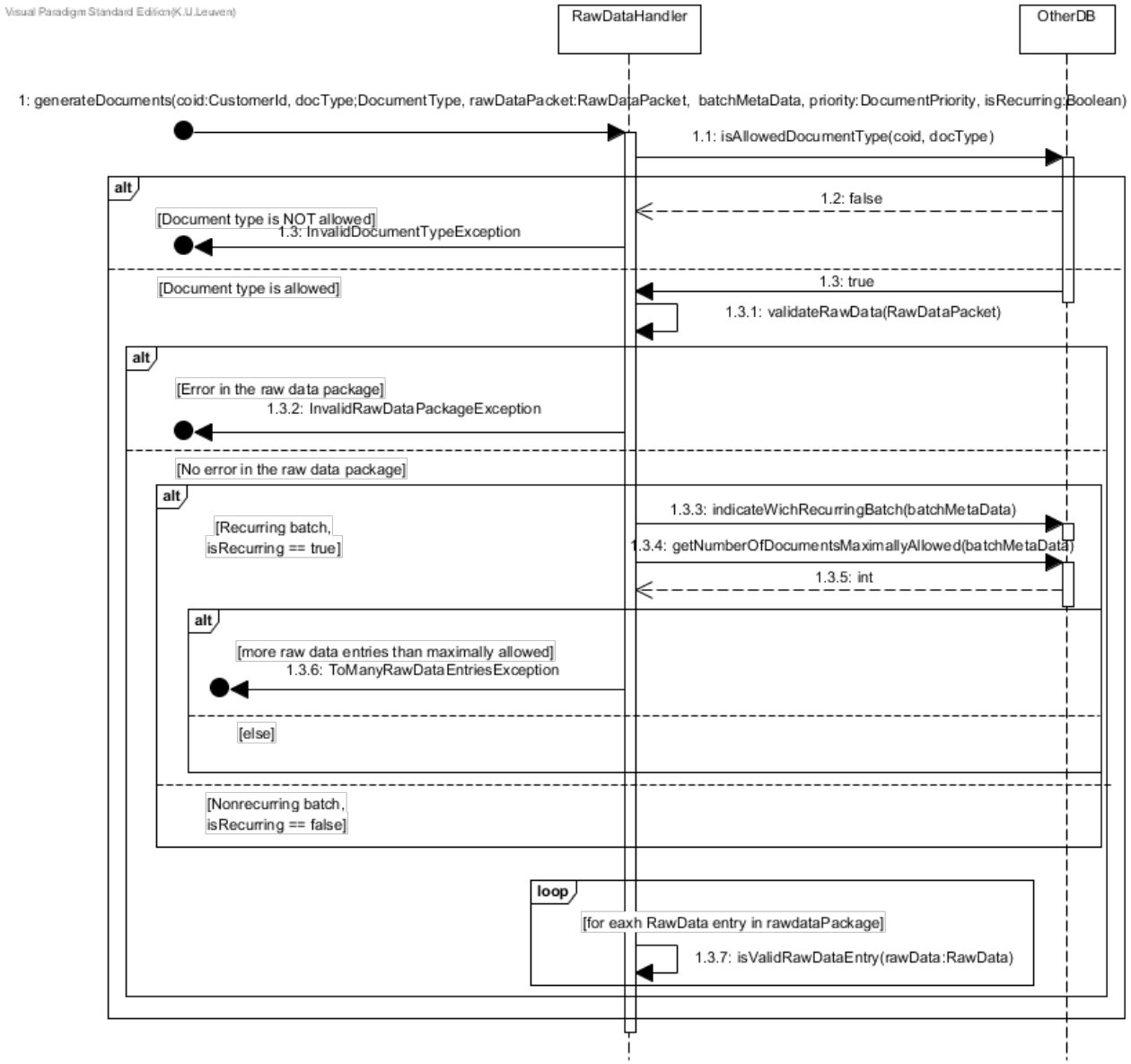

Figure 20: VerifyRawData: verification whether the raw data is valid.

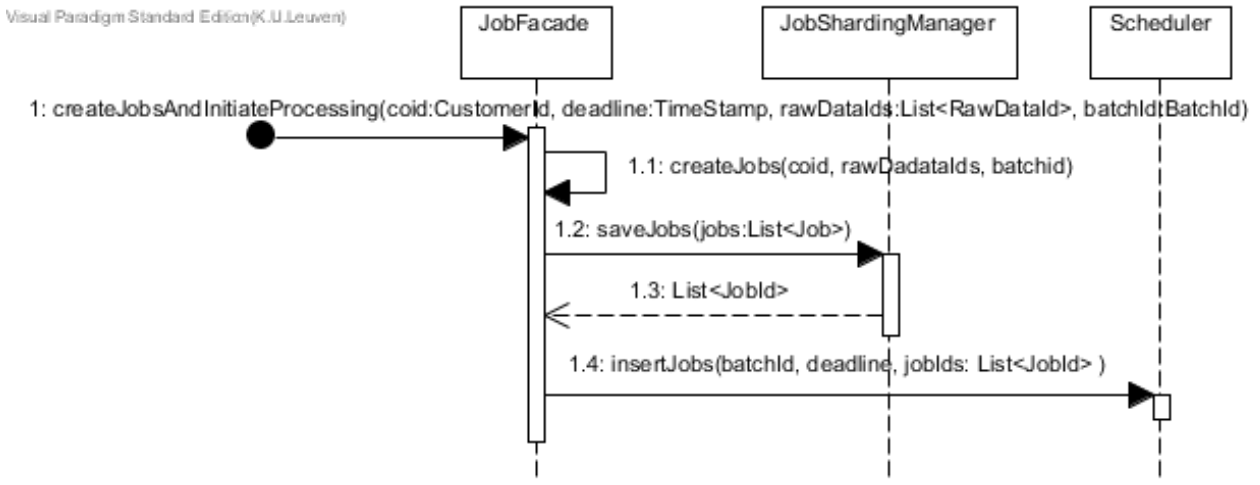

Figure 21: CreateJobs: the `JobManager creates jobs and forwards them to be scheduled by the Scheduler`.

6.7 Consult document in personal document store

Figure 22 shows how a Registered Recipient can consult a document in his or her personal document store. The Registered Recipient asks the `RecipientFacade` to look up the document identified by the given document identifier. There are multiple ways for a Registered Recipient to acquire a document identifier, e.g. by following a unique link (Section 6.8), by searching for document in the personal document store, by consulting the personal document store (Section 6.10). The system first verifies whether the session of the Registered Recipient is valid (as detailed in figure 18 in Section 6.5). After verification of the session identifier, the `RecipientFacade` asks the `PDSFacade` to fetch the document. But before fetching the document, the `PDSFacade` verifies whether the document identified by the given document identifier actually belongs to the Registered Recipient identified by the given recipient identifier. This way, the recipient cannot request documents that do not belong to him or her by giving a chosen document identifier. If the document does not belong to the recipient, an exception is thrown. Otherwise, the `PDSFacade` fetches the document and its meta-data from the `PDSDB`. The `RecipientFacade` returns the document and its meta-data to the recipient after marking the document and its job as received. Note that when marking the document as received, the time of receipt gets stored in the document's meta-data, so the recipient can later on search for documents based on their receipt date.

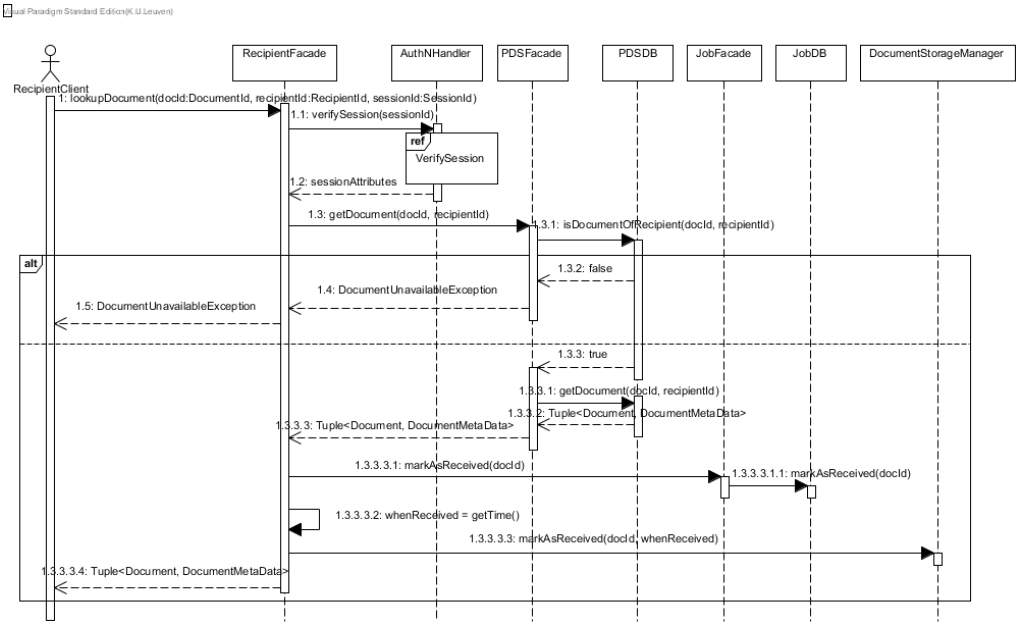

Figure 22: A Registered Recipient consults a document in his or her personal document store.

6.8 Downloading a document via unique link

Figure 23 shows how a recipient can download a document by following a unique link to this document sent by the System via e-mail. It shows both the cases when the recipient is registered and when he or she is not registered.
The recipient gives the link to the `RecipientFacade`, which asks the `LinkMappingManager` whether the link points to a document belonging to a registered or unregistered recipient. This is important, because the document will be downloaded from another database depending on this fact.
If the document belongs to an Unregistered Recipient, the `RecipientFacade` asks the `LinkMappingManager` for the document identifier to which the link maps. The `LinkMappingManager` checks the `TimeStamp` stored with the mapping between the link and the document identifier and uses it check whether the link is 30 days old. If the link is older than 30 days, an exception is thrown. Otherwise, the `LinkMappingManager` fetches and returns the document identifier stored in the mapping. The `RecipientFacade` the asks the `DocumentDB` to fetch the document and its meta-data identified by that document identifier. It will mark the job and the corresponding document as received, after which it returns the document and its meta-data. Note that when marking the document as received, the time of receipt gets stored in the document's meta-data, so the recipient can later on (i.e. after registration) search for documents based on their receipt date.
If the document belongs to a Registered Recipient, the `RecipientFacade` asks the `LinkMappingManager` for

the document identifier to which the link maps. The `LinkMappingManager` does not check the `TimeStamp` of the mapping but just returns the document identifier. The `RecipientFacade` has to be sure that document belongs to the recipient following the link so it requests the session identifier and the recipient identifier of the recipient following the link. If the recipient does not currently have a session, an exception is thrown. If he or she does have a session, the recipient returns his or her recipient identifier and session identifier to the `RecipientFacade`. The `RecipientFacade` then looks up the document using the session, recipient and document identifiers as explained in Section 6.7.

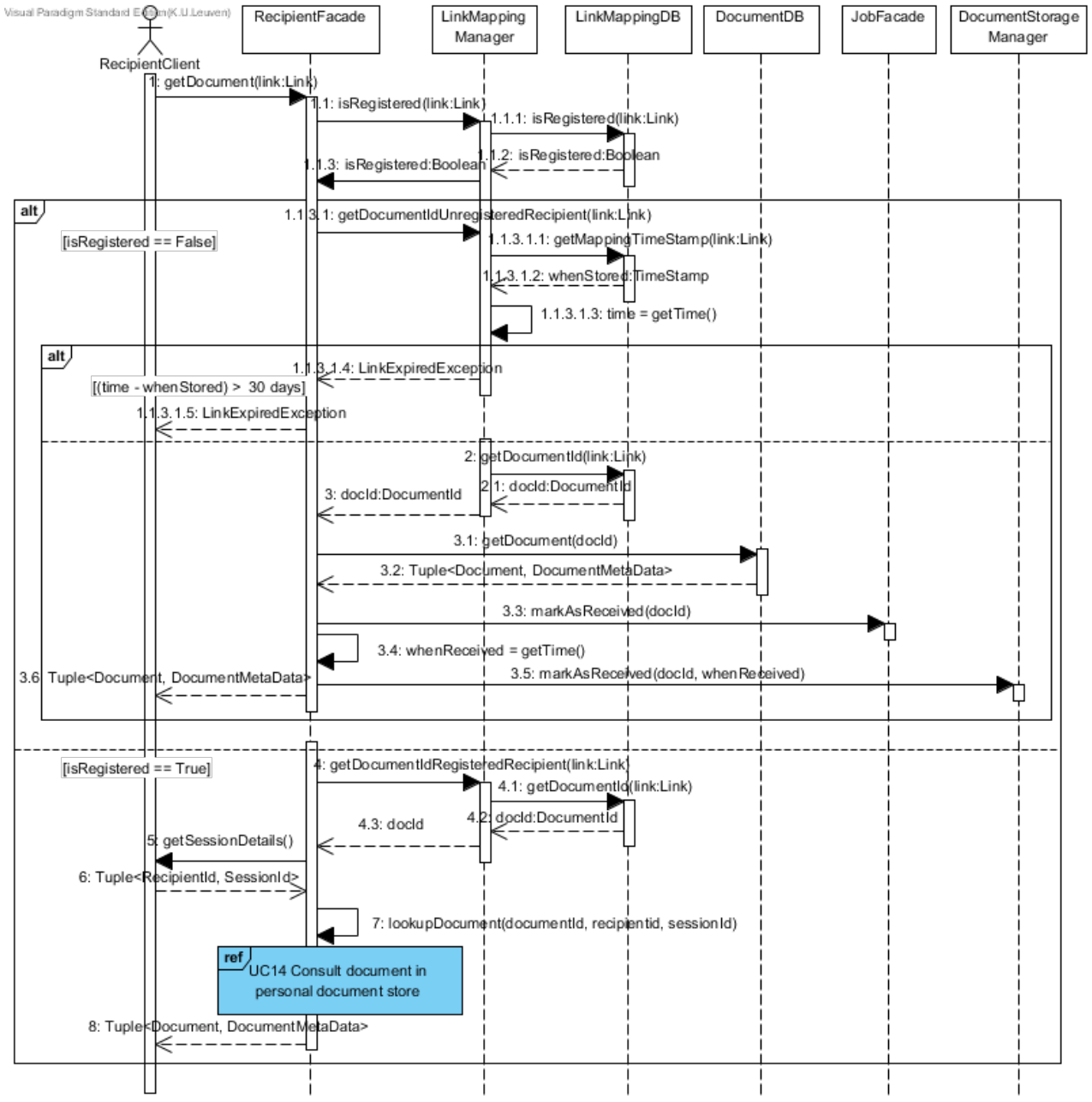

Figure 23: A recipient wants to download a document using a unique link.

6.9 Delivering a document

The initiation of the document delivery process is initiated by a `Generator` instance after the generation of a document, as explained in Section 6.1. This is shown in figure 24, where the `Generator` instance forwards the document and job identifier for the document that have to be delivered to `ChannelDispatcher`. The `ChannelDispatcher` uses the job identifier to ask the raw data entry corresponding to the document from the

`JobFacade`. The `ChannelDispatcher` extracts the delivery method from the raw data entry and choose what to do next with the document depending on that extracted delivery method value.

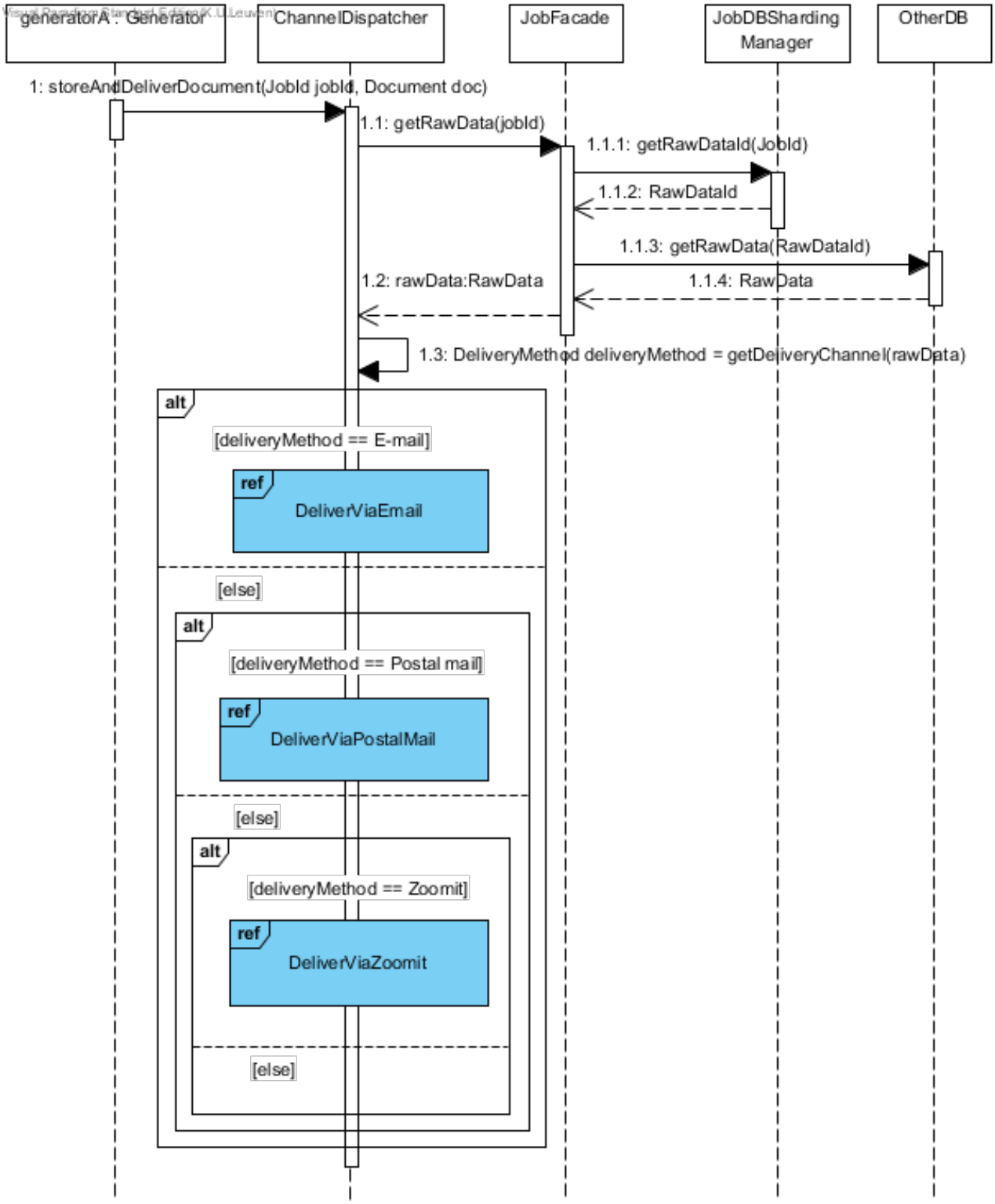

Figure 24: A `Generator` instance asks the `ChannelDispatcher` to store and deliver a document.

6.9.1 Delivering a document via the personal document store or via e-mail

Figure 25 shows the behaviour of the ChannelDispatcher when the delivery method indicated in the raw data is e-mail. The ChannelDispatcher will first extract the e-mail address from the raw data. It then looks up the document type and the name of customer organization for who the mail is sent, since this is information that will be put into the e-mail message. This information is retrieved from the OtherDB using the JobFacade and the job identifier. Next, the ChannelDispatcher checks whether the e-mail address belongs to a recipient who is registered to eDocs.

If the e-mail has to be sent to a Registered Recipient, the ChannelDispatcher first fetches the recipient's identifier. It will then contact the DocumentStorageManager, telling it that a document for a Registered Recipient needs to be stored, as shown in figure 26. The DocumentStorageManager generates a new document identifier for the document to be stored. It also generates the document meta-data for both the DocumentDB and the PDSDB. These meta-data both contain the document type and the name of the sender, but only the meta-data in the PDSDB contains the recipient identifier of recipient of the document, and only the meta-data in the DocumentDB contains the e-mail address of the recipient.
The DocumentStorageManager then stores the the document in both the DocumentDB and the PDSDB with the corresponding document identifier and correct meta-data. A sequence diagram for storing a document in the PDSDB was already made for the initial architecture and is shown in figure 31. Finally, the DocumentStorageManager stores the newly generated document identifier in the job of the document, so later on, the job identifier can be mapped onto the document identifier.
After storing the document for the Registered Recipient, figure 25 shows the ChannelDispatcher contacting the EmailFacade to send an e-mail to the Registered Recipient. As shown in figure 28, the EmailFacade first creates a unique link pointing to the document. The LinkMappingFunctionality takes care of creating and storing the mapping between the link and the document identifier. It also stores a boolean, indicating that the mapping point to a document of a registered recipient. The EmailFacade then generates an e-mail message containing a short description of the received document (i.e. the name of the sender of the document, the type of the document and the date at which the document was sent) and the link to the document. It finally sends the document to the e-mail address of the Registered Recipient using the interface to the EmailChannel and marks the document as sent. Figure 25 shows the ChannelDispatcher telling the BillingManager to add the cost of delivering the document via personal document store to the bill of the customer organization after sending the e-mail.
Note that the BillingManager gets contacted for every document delivery, without first checking whether the document is recurring or not. The BillingManager will then check whether the document belongs to a recurring batch or not and will only bill the customer organization if the document belongs to a non-recurring batch. This behaviour is shown in figure 30 for the case of delivering the document via personal document store.

If the e-mail has to be sent to an Unregistered recipient, the ChannelDispatcher also contacts the DocumentStorageManager, telling it that a document for an Unregistered Recipient needs to be stored, as shown in figure 27. The actions of the DocumentStorageManager are almost the same as those in figure 26, except that it only stores the document in the DocumentDB and not in the PDSDB. The DocumentStorageManager then returns the newly generated document identifier to th ChannelDispatcher.
After storing the document for the Unregistered Recipient, figure 25 shows the ChannelDispatcher using the JobFacade to find out if the customer organization has receipt tracking enabled or not. It then contacts the EmailFacade to send an e-mail to the Unregistered Recipient. As shown in figure 29, the EmailFacade changes its behaviour depending on the given boolean, which tells it whether the customer organization has enabled receipt tracking. If receipt tracking is turned on, the EmailChannel has the same behabiour as in figure 28. The only difference is that the LinkManagerstores a boolean indicating that the recipient is not registered. If receipt tracking is turned off, the EmailChannel send the full document as an attachment to the e-mail to the Unregistered Recipient using the interface to the EmailChannel. Figure 25 shows the ChannelDispatcher telling the BillingManager to add the cost of delivering the document via e-mail to the bill of the customer organization after sending the e-mail.

6.9.2 Delivering a document via postal mail

Figure 32 shows the behaviour of the ChannelDispatcher when the delivery method indicated in the raw data is postal mail. The ChannelDispatcher will first extract the postal address from the raw data. It then looks up the document type and the name of customer organization for which the mail is sent, since this is information that will be stored as document meta-data when storing the document. This information is

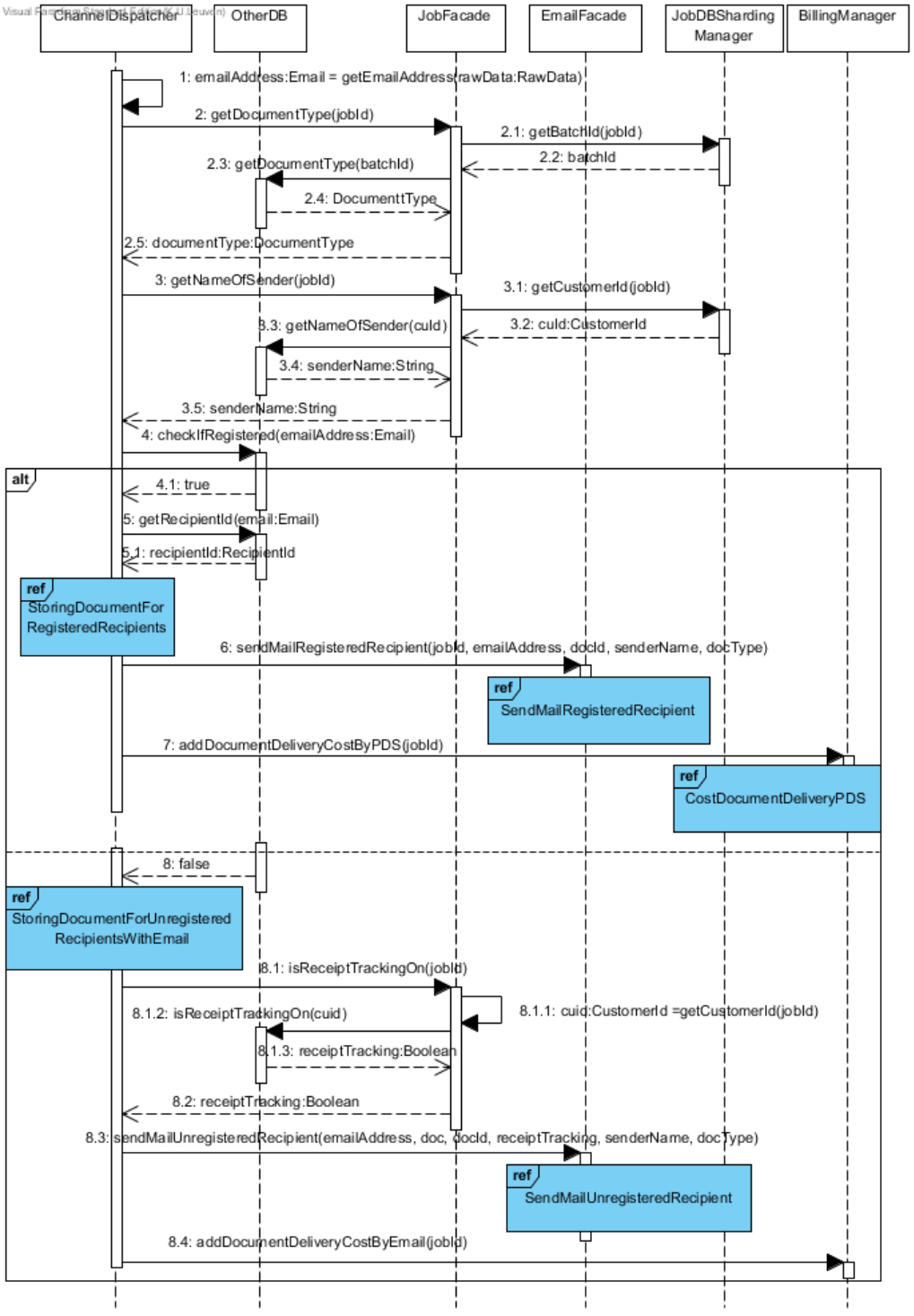

Figure 25: DeliverViaEmail: (Continuation of fig. 24) The `ChannelDispatcher` uses e-mail as a delivery method and will have a different flow when the e-mail should be delivered to a Registered or Unregistered Recipient.

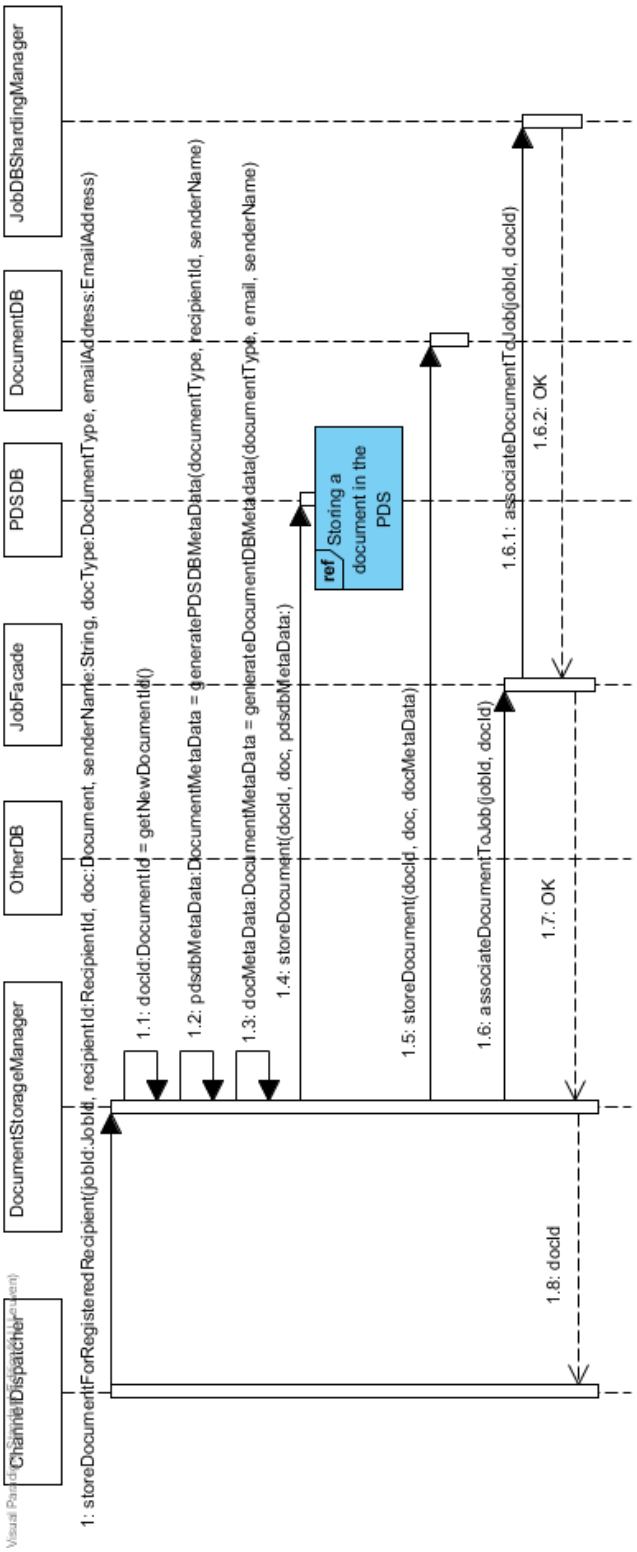

Figure 26: StoringDocumentForRegisteredRecipients: (continuation of fig. 25) The `ChannelDispatcher` requests the `DocumentStorageManager` to store a document for a Registered Recipient.

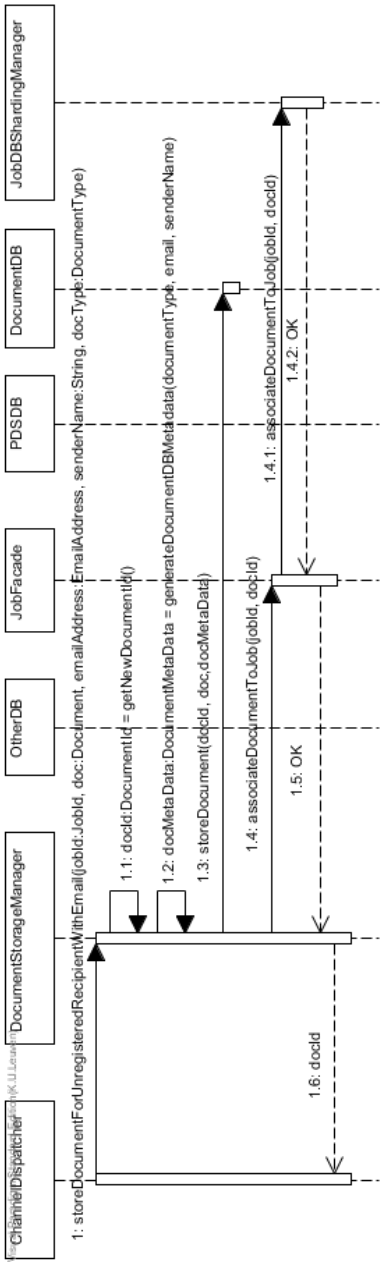

Figure 27: StoringDocumentForUnregisteredRecipientsWithEmail: (continuation of fig. 25) The `ChannelDispatcher` requests the `DocumentStorageManager` to store a document for an Unregistered Recipient.

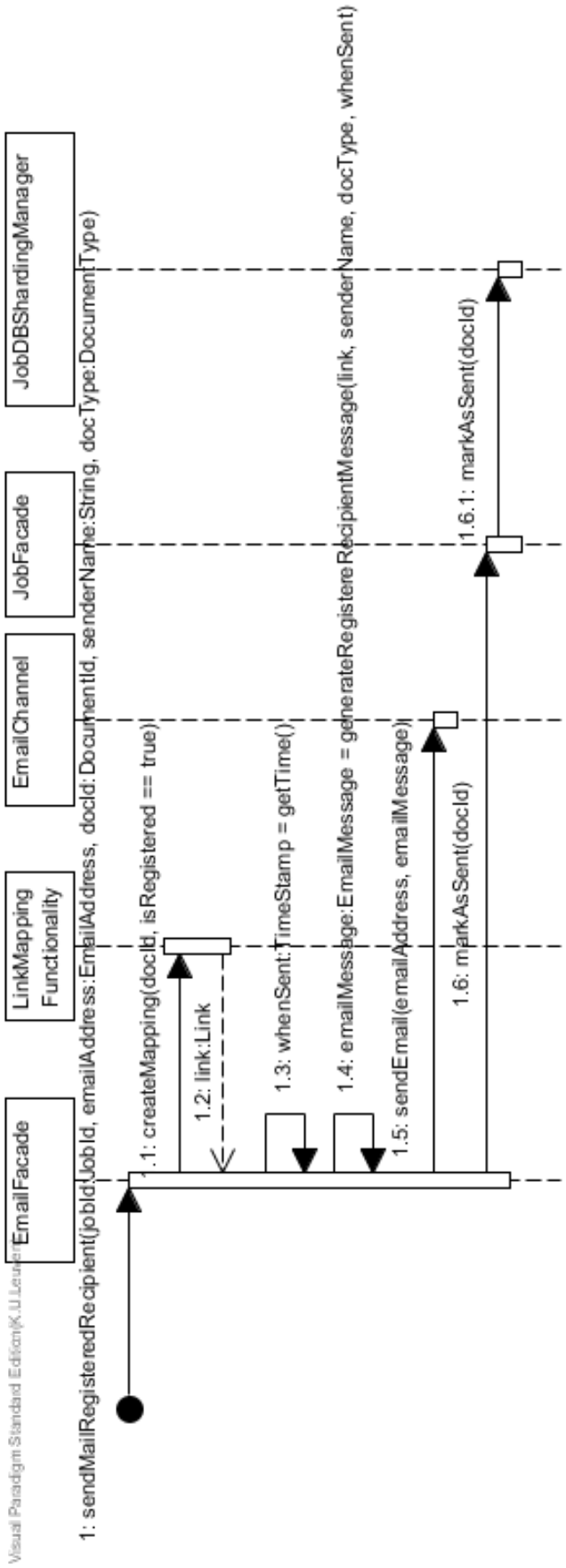

Figure 28: SendMailRegisteredRecipient: (continuation of fig. 25) The `ChannelDispatcher` requests the `EmailFacade` to send an e-mail to a Registered Recipient.

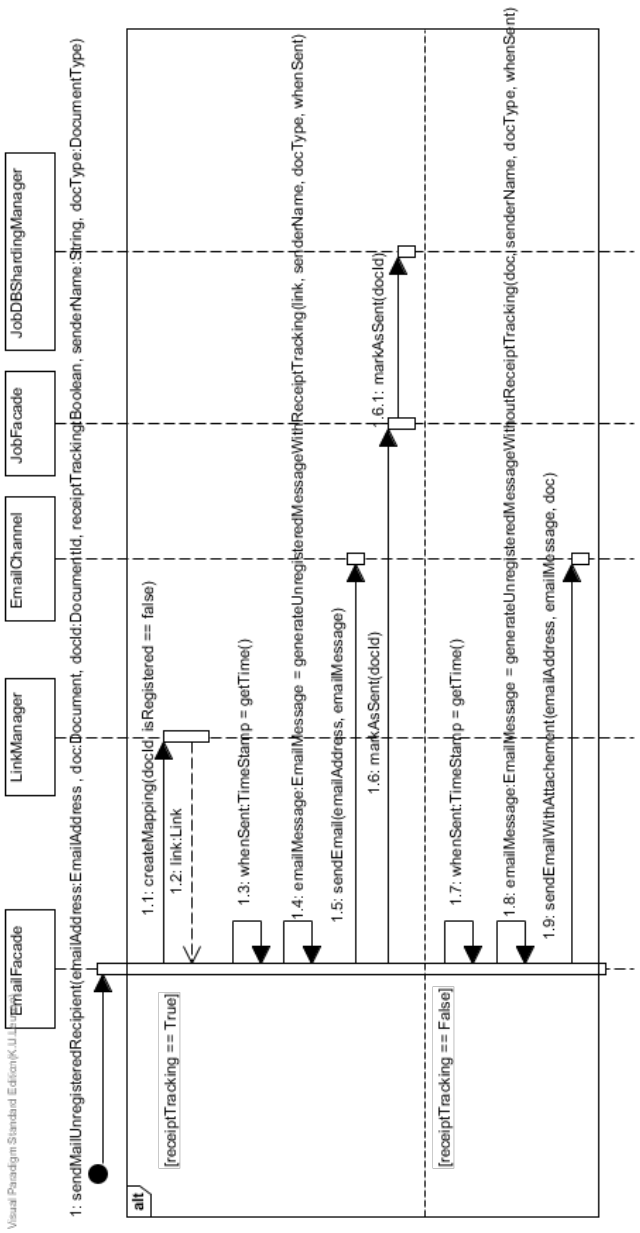

Figure 29: SendMailUnregisteredRecipient: (continuation of fig. 25) The `ChannelDispatcher` requests the `EmailFacade` to send an e-mail to an Unregistered Recipient.

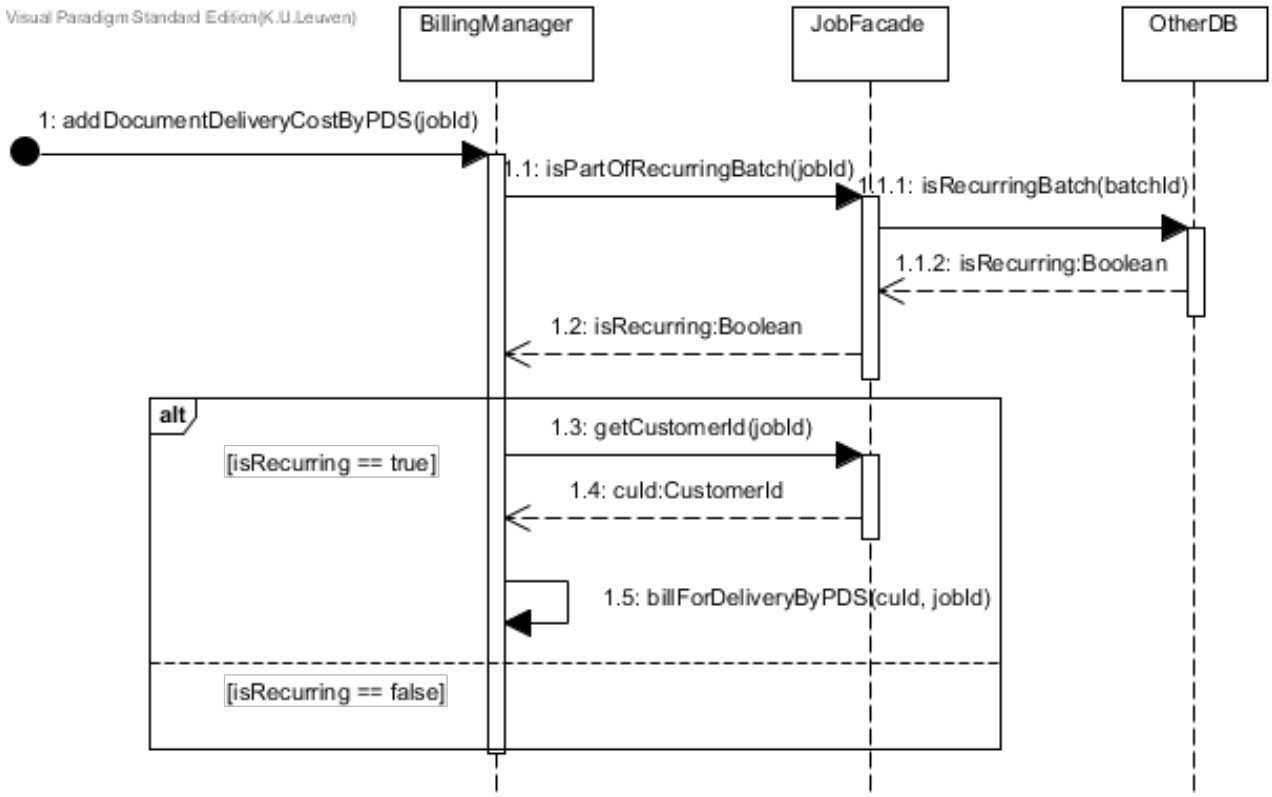

Figure 30: CostDocumentDeliveryPDS: The `BillingManager` adds the cost of delivering a document via the personal document store to the bill of the customer organization if the document is part of a non-recurring batch.

retrieved from the `OtherDB` using the `JobFacade` and the job identifier. In the same way, `ChannelDispatcher` also looks up the print parameters for the customer organization, which are stored in the SLA in the `OtherDB`. Next, the `ChannelDispatcher` contacts the `DocumentStorageManager`, telling it that a document for an Unregistered Recipient needs to be stored, as shown in figure 27. This is also discussed in Section 6.9.1.
After storing the document for the Unregistered Recipient, figure 25 shows the `ChannelDispatcher` contacting the `Print&PostalServiceFacade`, forwarding it the postal address, the postal parameters and the document. The `Print&PostalServiceFacade` sends this data to the external `Print&PostalServiceClient`, which will print the document using the given print parameters and send it to the given postal address. Next, `Print&PostalServiceFacade` marks the document as sent using the `JobFacade`. Finally, the `ChannelDispatcher` tells the `BillingManager` to add the cost of delivering the document via postal mail to the bill of the customer organization.

6.9.3 Delivering a document via Zoomit

Figure 33 shows the behaviour of the `ChannelDispatcher` when the delivery method indicated in the raw data is Zoomit. The `ChannelDispatcher` will first extract the Zoomit identifier of the recipient from the raw data. It then looks up the name of customer organization for which the document is sent, since this is information that will be stored as document meta-data when storing the document. This information is retrieved from the `OtherDB` using the `JobFacade` and the job identifier. In the same manner, the `ChannelFacade` uses the `JobFacade` to check whether the customer organization has receipt tracking turned on or off.
Next, the `ChannelDispatcher` contacts the `DocumentStorageManager`, telling it that a document for an Unregistered Recipient needs to be stored, as shown in figure 27. This is also discussed in Section 6.9.1.
After storing the document for the Unregistered Recipient, figure 33 shows the `ChannelDispatcher` contacting the `ZoomitFacade`, forwarding it the Zoomit identifier, the document identifier, the document and the name of the sending customer organization. The `ChannelDispatcher` also sends it a boolean indicating whether receipt tracking is turned on. If this boolean indicates that receipt tracking is turned on, the Zoomit Service knows that it has to notify the eDocs system when it has actually delivered the document to the addressee, i.e., when the addressee has opened the document in the web application of Zoomit. The `ZoomitFacade` sends

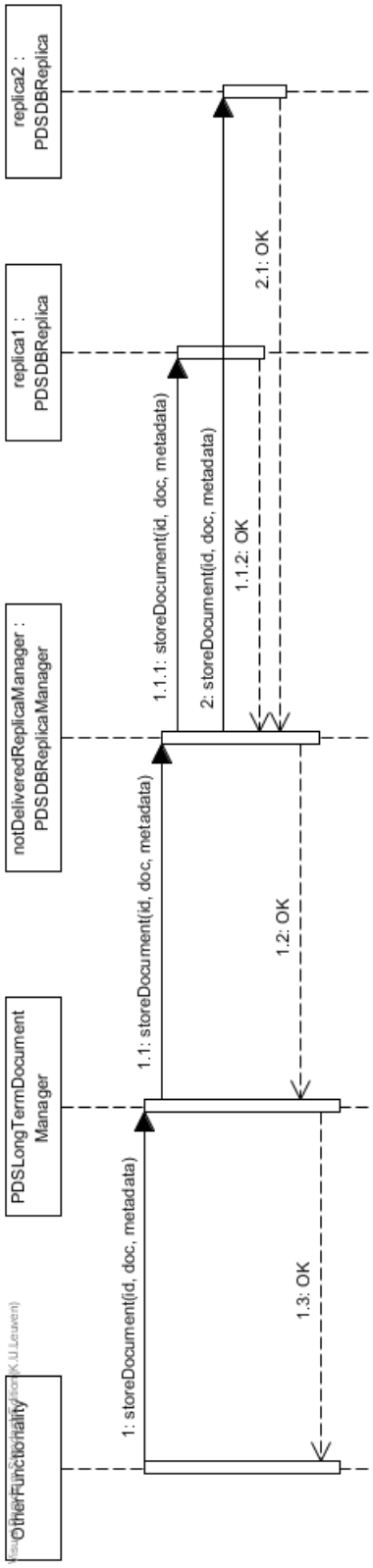

Figure 31: Storing a document in the PDS (continuation of fig. 26).

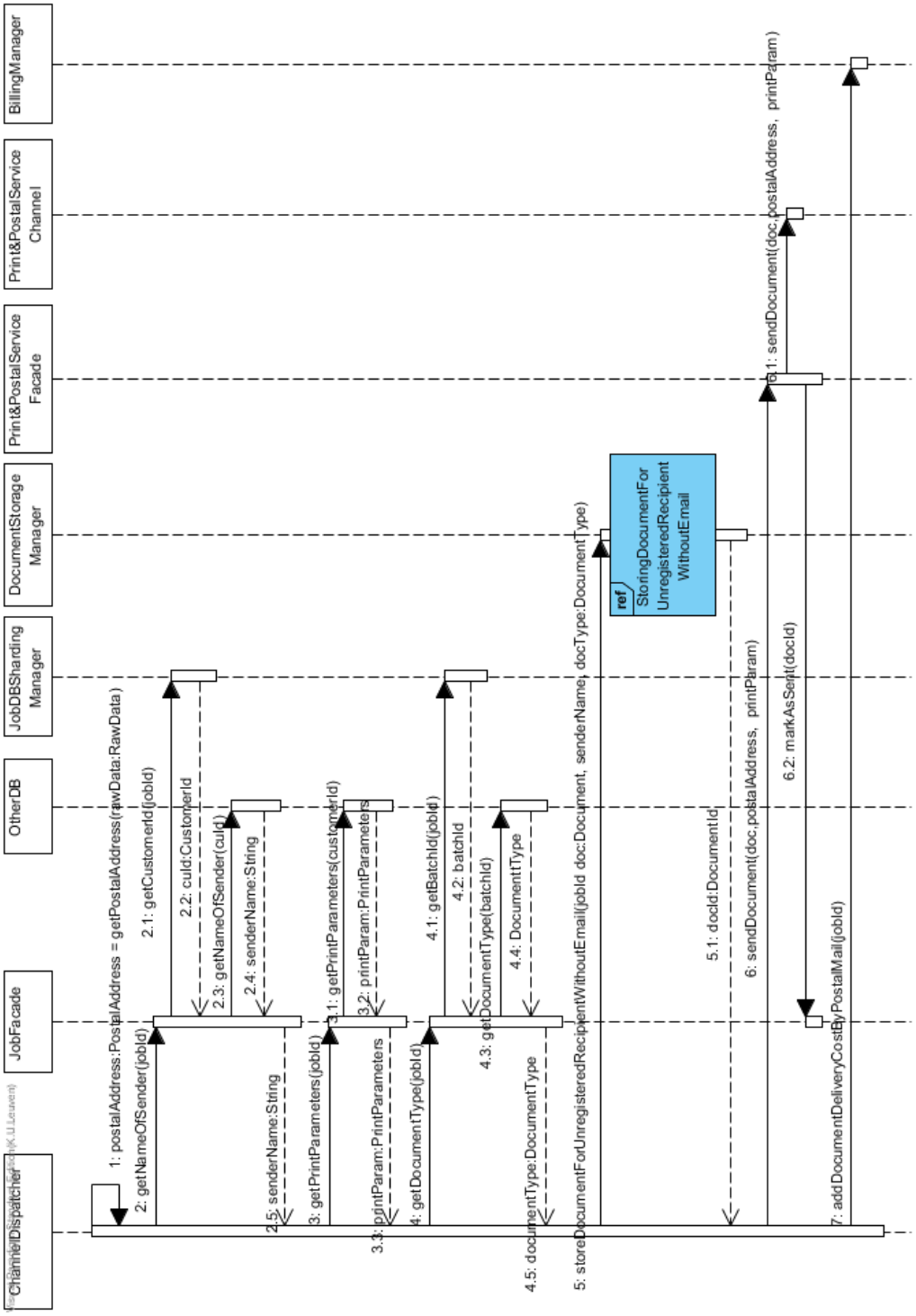

Figure 32: DeliverViaPostalMail: (Continuation of fig. 24) The `ChannelDispatcher` uses postal mail as a delivery method.

this information to the external `ZoomitClient`.
The `ZoomitChannel` can throw an exception when the receipt fails, e.g. because of an incorrect Zoomit account identifier or because this Zoomit user has indicated that he or she does not want to receive documents (of the customer organization identified by the given customer organization identifier) via Zoomit any more. In this case, the `ZoomitChannel` looks up the customer organization identifier of the customer organization and notifies it of the failure using the `NotificationHandler`. But if the receipt does not fail, the `ZoomitFacade` marks the document as sent. Finally, the `ChannelDispatcher` tells the `BillingManager` to add the cost of delivering the document via Zoomit to the bill of the customer organization.

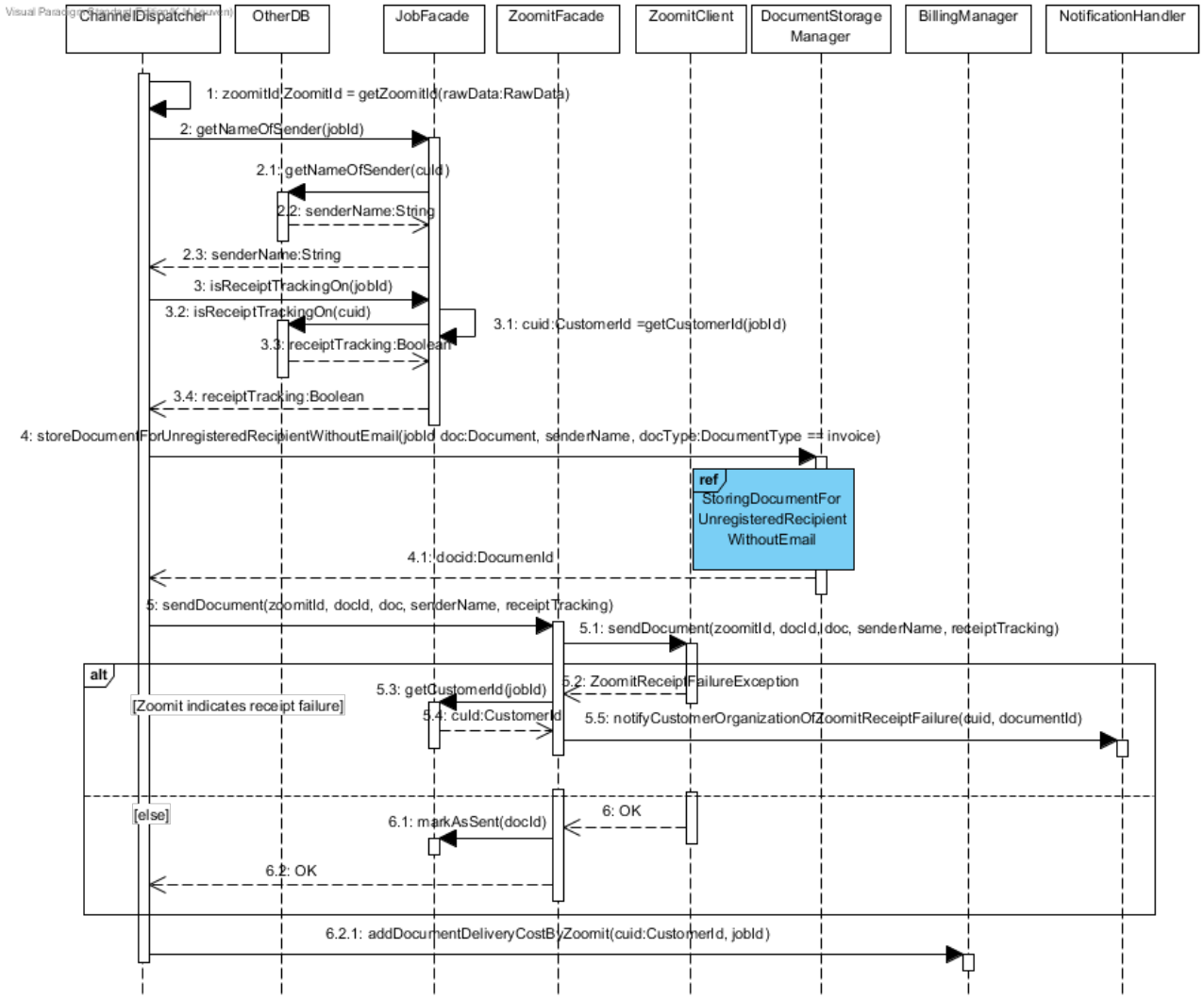

Figure 33: DeliverViaZoomit: (Continuation of fig. 24) The `ChannelDispatcher` uses Zoomit as a delivery method.

6.10 Consulting the personal document store

Figure 34 shows a Registered Recipient using a `RecipientClient` to get an overview of all received documents in his or her personal document store. First, the `RecipientFacade` verifies whether the session of the Registered recipient is valid (as detailed in figure 18). It then uses the `PDSFacade` to get the document identifiers and document meta-data of the received documents of the Registered Recipient. The `RecipientFacade` wraps this info into an overview and returns this overview to the `RecipientClient`. If the `PDSDB` has failed, an exception is thrown instead. The `RecipientClient` can uses the document identifiers wrapped in the overview to download a specific document, as explained in Section 6.7.

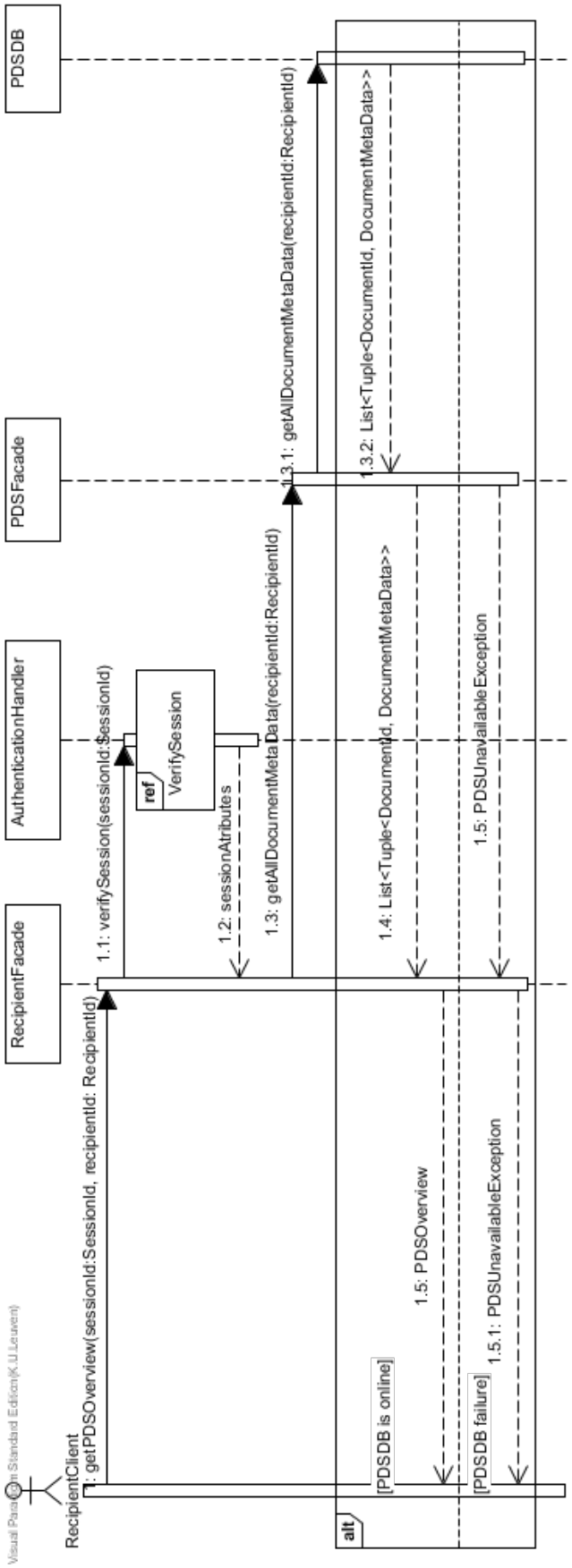

Figure 34: A Registered Recipient consults his or her personal document store.

6.11 A recipient registers to the PDS

Figure 35 shows how an Unregistered Recipient can register himself or herself to have a personal document store. First, the recipient indicates that he or she wants to register himself or herself by giving his or her details and the credentials he or she wants to use to the `RecipientFacade`. Theses details include the first name, last name, e-mail address and postal address of the recipient. The `RecipientFacade` forwards this information to the `RegistrationManager`, which then verifies whether the given `RecipientDetails` contain all the necessary information. If these do not contain all necessary information, an exception is thrown. Otherwise, the `RegistrationManager` checks whether given the e-mail address is already in use by a Registered Recipient. If it is already in use, an exception is thrown. Otherwise, the `RegistrationManager` orders the `OtherDB` to create a new Registered Recipient in the system bys storing the given `RecipientDetails` and `Credentials`. The `OtherDB` returns the recipient identifier of the newly created Registered Recipient, which the `RegistrationManager` then gives to the `DocumentStorageManager`, together with the the e-mail address. Next, the `DocumentStorageManager` copies all documents that were previously sent to the given e-mail address from the `DocumentDB` to the `PDSDB`. Before storing a document in the `PDSDB`, it inserts the recipient identifier in the meta-data of the document. After the copying of the documents, the `RecipientFacade` logs the newly created session in, as explained in Section 6.2. Finally, the `RecipientFacade` returns the recipient identifier and session identifier of the newly created logged in Registered Recipient.

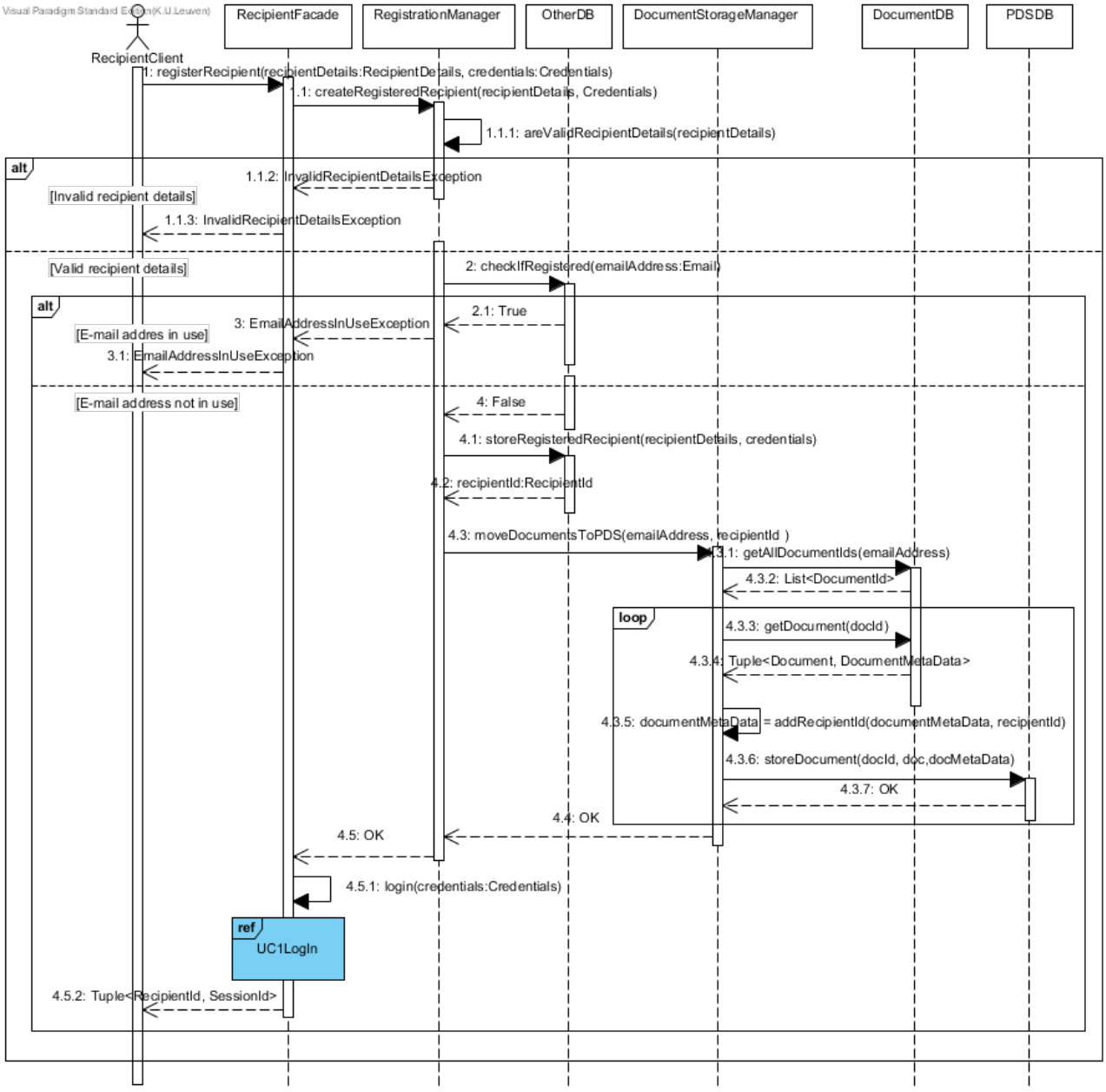

Figure 35: An Unregistered Recipient registers to the personal document store.

A Element catalog

In this section, we list all the components and the interfaces they provide. Per component, we describe its responsibilities, declare its super-component (if any) and list its sub-components (if any). Per interface, we list its methods by means of syntax. Per method, we list its effect (mostly in terms of subsequent actions) and possible exceptions. For the syntax of a method, we employ a Java-like notation. Since most data types are used by multiple methods, we give the details of these data types in a separate list in Section B and only refer to them in the interface specification.

A.1 AuthenticationHandler

- **Description:** The `AuthenticationHandler` is responsible for authenticating Registered recipients and Customer organizations. The architecture does not specify the means of authentication (e.g. the type of credentials). The credentials are stored in the `OtherDB`.
- **Super-component:** `UserFunctionality`
- **Sub-components:** None

Provided interfaces

- AuthN
  - `RecipientId getRecipientId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `AuthenticationHandler` fetches and returns the Registered Recipient's identifier corresponding to the `sessionId` from the `SessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers, or if the session belongs to a customer organization.
  - `CustomerId getCustomerId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `AuthenticationHandler` fetches and returns the Customer Organization's identifier corresponding to the `sessionId` from the `SessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier, or if the session belongs to a registered recipient.
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `AuthenticationHandler` will remove the session with the given id from the `SessionDB`. If no such session exists, nothing is changed and no exception is thrown.
    - Exceptions: None
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    - Effect: The `AuthenticationHandler` verifies the `credentials` using the `OtherDB`. If they are correct, the `AuthenticationHandler` creates a new session using the `SessionDB`, stores the id of the user (i.e. the Registered Recipient id or Customer Organization id) as an attribute in this session and returns the id of the new session. The id of the user is present in the given credentials.
    - Exceptions:
      - InvalidCredentialsException: Thrown if the given credentials are invalid.
- CheckSession
  - `Map<SessionAttributeKey, SessionAttributeValue> verifySession(SessionId sesionId) throws NoSuchsSessionException`
    - Effect: The `AuthenticationHandler` verifies whether a session with the given id exists in the `SessionDB` and if so, returns all its associated attributes.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers.

A.2 BillingManager

- **Description:** The `BillingManager` is responsible for all billing tasks. This includes billing the Customer Organization for the generation and delivery of non-recurring document processing jobs.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- CostRegistrationMgmt
  - `void addDocumentGenerationCost(JobId jobId)`
    - Effect: The `BillingManager` adds the cost for generating a document corresponding to the job identified by `jobId` to the bill of the customer organization if this job is part of a non-recurring batch. This method is called by the `GenerationManager`.
    - Exceptions: None
  - `void addDocumentDeliveryCostbyEmail(JobId jobId)`
    - Effect: The `BillingManager` adds the cost for delivering the document by e-mail to the bill of the customer organization if the job is part of a non-recurring batch. The document corresponds to the job identified by `JobId`. This method is called by the `ChannelDispatcher`.
    - Exceptions: None
  - `void addDocumentDeliveryCostbyPostalMail(JobId jobId)`
    - Effect: The `BillingManager` adds the cost for delivering the document by postal mail to the bill of the customer organization if the job is part of a non-recurring batch. The document corresponds to the job identified by `JobId`. This method is called by the `ChannelDispatcher`.
    - Exceptions: None
  - `void addDocumentDeliveryCostbyZoomit(JobId jobId)`
    - Effect: The `BillingManager` adds the cost for delivering the document by Zoomit to the bill of the customer organization if the job is part of a non-recurring batch. The document corresponds to the job identified by `JobId`. This method is called by the `ChannelDispatcher`.
    - Exceptions: None
  - `void addDocumentDeliveryCostbyPDS(JobId jobId)`
    - Effect: The `BillingManager` adds the cost for delivering the document by personal document store to the bill of the customer organization if the job is part of a non-recurring batch. The document corresponds to the job identified by `JobId`. This method is called by the `ChannelDispatcher`.
    - Exceptions: None

A.3 ChannelDispatcher

- **Description:** The `ChannelDispatcher` is responsible for choosing the correct delivery channel for a generated document. It also forwards the document to the `DocumentStorageManager`, which will store the document.
- **Super-component:** `Deliveryfunctionality`
- **Sub-components:** None

Provided interfaces

- FinalizeDocument
  Note that the methods in this interface are made idempotent. The methods of this interface are called by `Generator` instances.
  - `void storeAndDeliverDocument(JobId jobid, Document doc)`

* Effect: The `ChannelDispatcher` will store the given document `document` and deliver it. This method is made idempotent. To filer duplicate method calls, it has the `JobId` of the document as an argument. This idempotence is to account for the case when case when a `Generator` fails after forwarding the document and before reporting completion to the `DocumentGenerationManager`. In this case, it can be that the `DocumentGenerationManager` restarts jobs for which a document has already been stored or delivered.
* Exceptions: None

A.4 CustomerOrganizationClient

- **Description:** The `CustomerOrganizationClient` is external to the eDocs system and represents a client device of a Customer Organization (i.e. Customer Administrator and Customer Information System) that communicates with the eDocs System.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- None

A.5 CustomerOrganizationFacade

- **Description:** The `CustomerOrganizationFacade` provides the main interface of the system to the Customer Organization (i.e. Customer Administrator and Customer Information System).
- **Super-component:** `UserFunctionality`
- **Sub-components:** None

Provided interfaces

- CustomerOrganizationIncoming
  - `Boolean logout(SessionId sessionId)`
    * Effect: The `AuthenticationHandler` will remove the session with the given id from the `SessionDB`. If no such session exists, nothing is changed and no exception is thrown.
    * Exceptions: None
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    * Effect: The `AuthenticationHandler` verifies the `credentials` using the `OtherDB`. If they are correct, the `AuthenticationHandler` creates a new session using the `SessionDB`, stores the id of the user (i.e. the Registered Recipient id or Customer Organization id) as an attribute in this session and returns the id of the new session. The id of the user is present in the given credentials.
    * Exceptions:
      · InvalidCredentialsException: Thrown if the given credentials are invalid.
  - `List<Tuple<DocumentType,TimeStamp>> getPossibleDocumentTypes(SessionId sessionId, CustomerId cuId) throws NotAuthenticatedException`
    * Effect: The `CustomerOrganizationFacade` will return a list of the document types that the customer organization is allowed to generate. It also indicates the current template for each document type by returning for each document type the time when the last template was uploaded.
    * Exceptions:
      · NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `Boolean updateDocumentTemplate(SessionId sessionId, CustomerId cuId, DocumentType documentTyp Template template) throws NotAuthenticatedException, InvalidDocumentTypeException`

    - Effect: The `CustomerOrganizationFacade` will update the current template for the given `documentType` to the given `template` for the customer organization identified by `cuId`. Returns true when it succeeds.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
- `List<RawData> initiateDocumentProcessing(CustomerId cuid, SessionId sessionId, DocumentType docType, RawDataPacket rawDataPacket, DocumentPriority priority, Boolean isRecurring) throws NotAuthenticatedException, InvalidDocumentTypeException, InvalidRawDataPackageException, ToManyRawDataEntriesException`
    - Effect: The `CustomerOrganizationFacade` gets an indication from a customer organization with customer identifier `cuid` to start a batch of document processing jobs. The documents to be generated should be of document type `docType`. The boolean `isRecurring` indicates whether the batch is recurring or non-recurring. The information necessary to generate the documents is given in a `RawDataPacket` containing all the info about the raw data entries of the batch. After verification of the session identifier, the `CustomerOrganizationFacade` generates a `TimeStamp` of the time when this method is called and forwards it together with most of its arguments to the `RawDataHandler`.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
        - InvalidRawDataPackageException: Thrown if the raw data package contains an error (e.g. an invalid Excel file or incorrectly formatted XML).
        - ToManyRawDataEntriesException: Thrown if the batch is a recurring batch and contains more raw data entries than maximally allowed for this batch.
- `JobStatusOverview getJobStatusOverview(SessionId sessionId, CustomerId cuid) throws NotAuthenticatedException`
    - Effect: The `CustomerOrganizationFacade` first verifies the given session identifier `sessionId` using the `AuthenticationHandler`. The `CustomerOrganizationFacade` then requests to get an overview of all the document that the customer organization identified by the given customer organization identifier has ever initiated. It does this by querying the `JobManager` to get all status messages for each job belonging to the customer organization. It wraps these statuses in a `JobStatusOverview` object, which it returns.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.

A.6 Completer

- **Description:** The `Completer` is responsible for fetching the raw data an applicable meta-data for a group of `JobIds` when a `Generator` instance requires a new group of jobs.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- Complete
    - `CompletePartialBatchData getComplete(BatchId batchId, List<JobId> jobIds )`
        - Effect: The `Completer` fetches data needed by a `Generator` for generation of the documents corrseponding to the`JobIds` belonging to the same batch, which is identified by `BatchId`.
        - Exceptions: None

A.7 DocumentDB

- **Description:** The `DocumentDB` is responsible for actually storing all the documents. It stores documents regardless of the fact that these documents are also stored in the PDSDB. It receives read and write requests from the `DocumentStoragManager`.
- **Super-component:** None
- **Sub-components:** `DocumentDBShardingManager` and `DocumentDBShard`

Provided interfaces

- DocumentDBMgmt
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `DocumentDB` will store the given document `doc` together with the provided meta-data `md`.
    - Exceptions: None
  - `Tuple<Document, DocumentMetaData> getDocument(DocumentId id)`
    - Effect: The `DocumentDB` will return the document identified by the given document identifier together with its meta-data.
    - Exceptions:
      - NoSuchDocumentException: Thrown if there is no document identified by the given document identifier in the `DocumentDB`.
  - `List<DocumentId> geAllDocumentIds(EmailAddress emailAddress)`
    - Effect: The `DocumentDB` will return the document identifiers of all documents that where previously sent to the given `emailAddress`.
    - Exceptions: None
      - NoSuchDocumentException: Thrown if there is no document identified by the given document identifier in the `DocumentDB`.

A.8 DocumentDBShard

- **Description:** A `DocumentDBShard` is responsible for storing a partition of all the documents.
- **Super-component:** `DocumentDB`
- **Sub-components:** None

Provided interfaces

- DocumentDBShardingMgmt
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `DocumentDBShard` will store the given document `doc` together with the provided meta-data `md`.
    - Exceptions: None
  - `Tuple<Document, DocumentMetaData> getDocument(DocumentId id)`
    - Effect: The `DocumentDBShard` will return the document identified by the given document identifier together with its meta-data.
    - Exceptions:
      - NoSuchDocumentException: Thrown if there is no document identified by the given document identifier in the `DocumentDBShard`.
  - `List<DocumentId> geAllDocumentIds(EmailAddress emailAddress)`
    - Effect: The `DocumentDBShard` will return the document identifiers of all documents it contains that where previously sent to the given `emailAddress`.
    - Exceptions: None
      - NoSuchDocumentException: Thrown if there is no document identified by the given document identifier in the `DocumentDBShard`.

A.9 DocumentDBShardingManager

- **Description:** The `DocumentDBShardingManager` manages the storage of the documents over multiple `DocumentDBShards`.
- **Super-component:** The `DocumentDB`
- **Sub-components:** None

Provided interfaces

- DocumentDBMgmt
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `DocumentDBShardingManager` will store the given document `doc` together with the provided meta-data `md` in the appropriate `DocumentDBShard`.
    - Exceptions: None
  - `Tuple<Document, DocumentMetaData> getDocument(DocumentId id)`
    - Effect: The `DocumentDBShardingManager` will return the document identified by the given document identifier together with its meta-data.
    - Exceptions:
      - NoSuchDocumentException: Thrown if there is no document identified by the given document identifier in any of the `DocumentDBShards`.
  - `List<DocumentId> geAllDocumentIds(EmailAddress emailAddress)`
    - Effect: The `DocumentDBShardingManager` will return the document identifiers of all documents that where previously sent to the given `emailAddress`.
    - Exceptions: None
      - NoSuchDocumentException: Thrown if there is no document identified by the given document identifier in any of the `DocumentDBShards`.

A.10 DocumentGenerationManager

- **Description:** The `DocumentGenerationManager` monitors the availability of the `Generator` components using the Ping interface. The `DocumentGenerationManager` keeps track of the jobs assigned to and being processed by the `Generators`. To minimize the overhead of the job coordination, the `DocumentGenerationManager` assigns jobs to the `Generators` in groups of more than one job that are part of the same batch. If a `Generator` fails to complete its jobs, the `DocumentGenerationManager` can restart these failed jobs.
  It prioritizes jobs based on thei deadlines ansd schedules them according to *P1*.
- **Super-component:** None
- **Sub-components:** `Completer`, `GenerationManager`, `KeyCache`, `Scheduler`, `TemplateCache`

Provided interfaces

- InsertJobs
  - `void insertJobs(BatchId batchId, TimeStamp deadline, List<JobId> jobIds )`
    - Effect: The `DocumentGenerationManager` adds the jobs identified by their `JobIds` to its queue of all jobs that have not been processed yet. To lower the size of this queue, the `DocumentGenerationManager` is only given the information it needs, i.e., the id of the batch, its deadline and the ids of the individual jobs. This method provides new jobs synchronously to the `DocumentGenerationManager`, which it schedules synchronously. This means that when the method call returns, the given jobs are scheduled.
    - Exceptions: None
- NotifyCompleted

- `void notifyCompletedAndGiveMeMore(GeneratorId id)`
  - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
  - Exceptions: None
- `void notifyCompletedAndIAmShuttingDown(GeneratorId id)`
  - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
  - Exceptions: None

A.11 DocumentStorageCache

- **Description:** The `DocumentStorageCache` is responsible for storing the `DocumentIds` and `RecipientIds` when the `PDSDB` fails. According to Av2, the system should temporarily store at least 3 hours of documents to be delivered via the personal document store. When the `PDSDB` fails, the documents that are supposed to also be saved in the `PDSDB` are saved in the `DocumentDB`, just as usual. But in this case the `DocumentStorageManager` also stores the `DocumentIds` and `RecipientIds` of those documents in the `DocumentStorageCache` for at least 3 hours. This way, the `DocumentStorageManager` can transfer these documents from the `DocumentDB` to the `PDSDB` using this information if the `PDSDB` comes back online within 3 hours. The requirements do not specify what happens after 3 hours, so in this architecture, the behaviour after those 3 hours is undefined.
- **Super-component:** `DocumentStorageFunctionality`
- **Sub-components:** None

Provided interfaces

- DocumentStorageCacheMgmt
  - `void storeDocument(DocumentId docId, RecipientId recipId)`
    - Effect: The `DocumentStorageCache` stores the given `DocumentId` and `RecipientId` in its cache.
    - Exceptions: None
  - `Tuple<DocumentId docId, RecipientId recipId> fetchNextDocument()`
    - Effect: The `DocumentStorageCache` retrieves the `DocumentId` and `RecipientId` corresponding to the next document in its cache.
    - Exceptions: None
  - `List<Tuple<DocumentId docId, RecipientId recipId>> fetchAllDocuments()`
    - Effect: The `DocumentStorageCache` retrieves the `DocumentIds` and `RecipientIds` corresponding to all documents in its cache.
    - Exceptions: None

A.12 DocumentStorageFunctionality

- **Description:** The `DocumentFunctionality` is responsible for storing the generated documents in the correct database. If the document belongs to a registered recipient, it sends a write request to both the `DocumentDB` and the `PDSDB`. Otherwise, it only sends a write request to the `DocumentDB`, which stores all the documents.
  It is also responsible for copying documents from the `DocumentDB` to the `PDSDB` when an Unregistered Recipient registers to the eDocs system.
  Another responsibility of the `DeliveryFunctionality` is storing at least 3 hours of documents when the `PDSDB` fails.
- **Super-component:** None
- **Sub-components:** `DocumentStorageCache` and `DocumentStorageManager`

Provided interfaces

- DocumentStorageMgmt
  - `Documentid storeDocumentForRegisteredRecipient(JobId jobId, RecipientId recipientId, Document document, String senderName, DocumentType docType, EmailAddress emailAddress)`
    - Effect: The `DocumentStorageFunctionality` will try to store the given `document` for the registered recipient identified by `recipientId` in both the `PDSDB` and `DocumentDB`. It generates a new document identifier for the document and returns it.
    - Exceptions: None
  - `Documentid storeDocumentForUnregisteredRecipientWithEmail(JobId jobId, Document document, String senderName, DocumentType docType, EmailAddress emailAddress)`
    - Effect: The `DocumentStorageFunctionality` stores the given `document` for the unregistered recipient in the `DocumentDB`. It stores the given e-mail address in the meta-data of the document. This way, this document can later be added to the recipient's personal document store should he or she subscribe with the given e-mail address. The `DocumentStorageFunctionality` generates a new document identifier for the document and returns it.
    - Exceptions: None
  - `Documentid storeDocumentForUnregisteredRecipientWithoutEmail(JobId jobId, Document document, String senderName, DocumentType docType)`
    - Effect: The `DocumentStorageFunctionality` stores the given `document` for the unregistered recipient in the `DocumentDB`. The `DocumentStorageFunctionality` generates a new document identifier for the document and returns it.
    - Exceptions: None
  - `void markAsReceived(DocumentId documentId, TimeStamp whenReceived)`
    - Effect: The `DocumentStorageFunctionality` marks the document identified by the given document identifier as received, by storing the given time in the document meta-data of the document. The given time is the time when the recipient should have received the document. Note that this method only has effect the first time it is called for a valid document identifier and time stamp.
    - Exceptions: None
  - `void moveDocumentsToPDS(EmailAddress emailAddress, RecipientId recipientId)`
    - Effect: The `DocumentStorageFunctionality` will fetch all documents and their meta-data from the `DocumentDB` that where sent to the given `emailAddress` and store them in the personal document store of the Registered Recipient identified by the given recipient identifier. It will also put the `recipientId` in the meta-data of all those documents.
    - Exceptions: None
  - `void moveDocumentsToPDS(EmailAddress emailAddress, RecipientId recipientId)`
    - Effect: The `DocumentStorageFunctionality` will fetch all documents and their meta-data from the `DocumentDB` that where sent to the given `emailAddress` and store them in the personal document store of the Registered Recipient identified by the given recipient identifier. It will also put the `recipientId` in the meta-data of all those documents.
    - Exceptions: None

A.13 DocumentStorageManager

- **Description:** The `DocumentStorageManager` is responsible for storing the generated documents in the correct database. If the document belongs to a registered recipient, it sends a write request to both the `DocumentDB` and the `PDSDB`. Otherwise, it only sends a write request to the `DocumentDB`, which stores all the documents.
  It is also responsible for copying documents from the `DocumentDB` to the `PDSDB` when an Unregistered Recipient registers to the eDocs system.
  Another responsibility of the `DocumentStorageManager` is storing at least 3 hours of documents when the `PDSDB` fails.
- **Super-component:** `DocumentStoragefunctionlity`
- **Sub-components:** None

Provided interfaces

- DocumentStorageMgmt
  - `Documentid storeDocumentForRegisteredRecipient(JobId jobId, RecipientId recipientId, Document document, String senderName, DocumentType docType, EmailAddress emailAddress)`
    - Effect: The `DocumentStorageManager` will try to store the given `documment` for the registered recipient identified by `recipientId` in both the `PDSDB` and `DocumentDB`. It generates a new document identifier for the document and returns it.
    - Exceptions: None
  - `Documentid storeDocumentForUnregisteredRecipientWithEmail(JobId jobId, Document document, String senderName, DocumentType docType, EmailAddress emailAddress)`
    - Effect: The `DocumentStorageManager` stores the given `document` for the unregistered recipient in the `DocumentDB`. It stores the given e-mail address in the meta-data of the document. This way, this document can later be added to the recipient's personal document store should he or she subscribe with the given e-mail address. The `DocumentStorageManager` generates a new document identifier for the document and returns it.
    - Exceptions: None
  - `Documentid storeDocumentForUnregisteredRecipientWithoutEmail(JobId jobId, Document document, String senderName, DocumentType docType)`
    - Effect: The `DocumentStorageManager` stores the given `document` for the unregistered recipient in the `DocumentDB`. The `DocumentStorageManager` generates a new document identifier for the document and returns it.
    - Exceptions: None
  - `void removeDocumentsAtUnregistration(RecipientId recipientId)`
    - Effect: The `DocumentStorageManager` removes all documents belonging to the Registered Recipient identified by the given recipient identifier from the `PDSDB`.
    - Exceptions: None
  - `void markAsReceived(DocumentId documentId, TimeStamp whenReceived)`
    - Effect: The `DocumentStorageManager` marks the document identified by the given document identifier as received, by storing the given time in the document meta-data of the document. The given time is the time when the recipient should have received the document. Note that this method only has effect the first time it is called for a valid document identifier and time stamp.
    - Exceptions: None
  - `void moveDocumentsToPDS(EmailAddress emailAddress, RecipientId recipientId)`
    - Effect: The `DocumentStorageManager` will fetch all documents and their meta-data from the `DocumentDB` that where sent to the given `emailAddress` and store them in the personal document store of the Registered Recipient identified by the given recipient identifier. It will also put the `recipientId` in the meta-data of all those documents.
    - Exceptions: None

A.14 DeliveryFunctionality

- **Description:** The `DeliveryFunctionality` is responsible for delivering the documents generated by eDocs to the recipients.
- **Super-component:** `ChannelDispatcher`, `EmailFacade`, `Print&PostalServiceFacade`, `ZoomitFacade` and `ZoomitDeliveryCache`
- **Sub-components:** None

Provided interfaces

- ConfirmZoomit
  - `void markAsDelivered(DocumentId document, TimeStamp whenDelivered)`
    - Effect: The `DeliveryFunctionality` gets notified that the document identified by the given document identifier is has been delivered by its addressee. The given `TimeStamp` indicates when the document had been delivered.
- NotifyDeliveryFailure
  - `void notifyOfEmailDeliveryFailure(EmailAddress emailAddress, EmailMessage emailMessage`
    - Effect: The `DeliveryFunctionality` notifies looks up the customer organization
- FinalizeDocument
  Note that the methods in this interface are made idempotent. The methods of this interface are called by `Generator` instances.
  - `void storeAndDeliverDocument(JobId jobid, Document doc)`
    - Effect: The `DeliveryFunctionality` will store the given document `document` and deliver it. This method is made idempotent. To filer duplicate method calls, it has the `JobId` of the document as an argument. This idempotence is to account for the case when case when a `Generator` fails after forwarding the document and before reporting completion to the `DocumentGenerationManager`. In this case, it can be that the `DocumentGenerationManager` restarts jobs for which a document has already been stored or delivered.
    - Exceptions: None

A.15 EDocsAdminClient

- **Description:** The `EDocsAdminClient` is external to the eDocs system and represents a client device of an administrator of eDocs that communicates with the eDocs System.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- EDocsAdminOutgoing
  - `SessionId getSessionDetails() throws NotAuthenticatedException`
    - Effect: The `EDocsAdminClient` returns the session identifier of the current session if the eDocs Administrator is currently logged in. Otherwise, an exception is thrown.
    - Exceptions:
      - NoSessionException: Thrown if the eDocs Administrator does not currently have a session with the eDocs system.
  - `void notifyEdocsOperator(NotificationMessage msg, TimeStamp whenFailed)`
    - Effect: The `EDocsAdminClient` will deliver a textual message `msg` to the eDocs operators, which contains further information about the specific failure.
    - Exceptions: None

A.16 EDocsAdminFacade

- **Description:** The `EDocsAdminClient` is responsible for the interaction of eDocs Administrators with the system.
- **Super-component:** `UserFunctionality`
- **Sub-components:** None

Provided interfaces

- EdocsAdminIncoming
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    - Effect: The `EDocsAdminFacade` forwards the given `credentials` to the `AuthenticationHandler`, which verifies them and returns a new session identifier if correct. This session identifier can be used in future requests to the `RecipientFacade`.
    - Exceptions:
      - InvalidCredentialsException: Thrown if the `AuthenticationHandler` indicated that the given credentials where incorrect.
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `EDocsAdminFacade` removes the session corresponding to the `sessionId` using the `AuthenticationHandler`. As a result, this session cannot be used anymore to access the system without logging in again. If no session corresponds to the `sessionId`, it does not exist, nothing is changed but no exception is thrown.
    - Exceptions: None
  - `void registerCustomerOrganization(SessionId edocsSessionid, CustomerOrganizationDetails custOrgDetails, Credential credentials) throws InvalidCustomerOrganizationDetailsException, EmailAddressInUseException, NotAuthenticatedException`
    - Effect: The `EDocsAdminFacade` registers the Customer Organization using the given details.
    - Exceptions:
      - InvalidCustomerOrganizationDetailsException: Thrown if the provided details are not valid, e.g. there are missing details. The exception lists the missing details to the eDocs Administrator.
      - EmailAddressInUseException: Thrown if the given e-mail address in **custOrgDetails** is already registered.
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `void unregister(SessionId edocsSessionid, RecipientId recipientId) throws NotAuthenticatedException`
    - Effect: After using the `AuthenticationHandler` to verify the session identifier of the eDocs Administrator, the `EDocsAdminFacade` will ask the `EDocsAdminClient` for confirmation of the unregistration. If the eDocs Administrator confirms, the account of the customer organization gets marked as inactive. Finally, the `EDocsAdminFacade` logs the customer organization out.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.

A.17 EmailChannel

- **Description:** The `EmailChannel` is responsible for the delivery emails. It is external to the eDocs system and represents a mail server of an e-mail provider.
- **Super-component:** None
- **Sub-components:** None.

Provided interfaces

- DeliverEmail
  - `void sendEmail(EmailAddress emailAddress, EmailMessage emailMessage)`
    - Effect: The `EmailChannel` sends the given e-mail message `emailMessage` to the given e-mail address `emailAddress`.
    - Exceptions: None

- `void sendEmailWithAttachement(EmailAddress emailAddress, EmailMessage emailMessage, Document document)`
  - Effect: The `EmailChannel` sends the given e-mail message `emailMessage` to the given e-mail address `emailAddress` with the given `document` as an attachment.
  - Exceptions: None

A.18 EmailFacade

- **Description:** The `EmailFacade` is responsible for creating and sending emails used in the delivery of documents. It will send documents to Unregistered recipients by e-mail when receipt tracking is turned off. When receipt tracking is turned on for an Unregistered Recipient, it will send an e-mail containing a short description of the received document and a unique link, which can be followed to get document. For Registered Recipients, it will send an e-mail containing a short description of the document and a link to the document. It also marks jobs as sent using the `JobManager`.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

Provided interfaces

- Deliver
  - `void sendMailRegisteredRecipient(Jobid jobId, EmailAddress emailAddress, DocumentId docId, String senderName, DocumentType docType)`
    - Effect: The `EmailFacade` will send an e-mail to the Registered Recipient with a notification that a document corresponding to the job identified by `jobId` has been added to his or her personal document store. The e-mail will contain a short description of the received document (i.e., the sender of the document (`senderName`), the `documentType` of the document and the date when the document was sent). The e-mail will also contain a unique link pointing to the document in the personal document store. After sending the e-mail to the given `emailAddress`, the `EmailFacade` will mark the document as sent.
    - Exceptions: None
  - `void sendMailUnregisteredRecipient(EmailAddress emailAddress, Document document, DocumentId documentId, Boolean receiptTracking)`
    - Effect: If `receiptTracking` is true, receipt tracking is turned on. The `EmailFacade` will send an e-mail to the Unregistered Recipient, which will contain a short description of the received document (i.e., the sender of the document (`senderName`), the `documentType` of the document and the date when the document was sent). The e-mail will also contain a unique link that can be used to download the document. After sending the e-mail to the given `emailAddress`, the `EmailFacade` will mark the document as sent.
      If receipt tracking is turned of, the `EmailFacade` will send the document as an attachment with an e-mail.
    - Exceptions: None
- NotifyDeliveryFailure
  - `void notifyOfEmailDeliveryFailure(EmailAddress emailAddress, EmailMessage emailMessage`
    - Effect: The `EmailFacade` looks up the customer organization corresponding to the failed e-mail address and notifies that customer organization of the delivery failure.

A.19 Generator

- **Description:** A `Generator` generates the documents and forwards them to `DeliveryFunctionality` to store and deliver them. Its availability is monitored by the `DocumenGenerationManager` with the Ping interface. A `Generator` is also responsible of notifying the `NotificationHandler` that it does not have all of the data required to fill in the template.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- AssignJobs
  - `void assignJobs(CompletePartialBatchData batchData)`
    - Effect: The `Generator` gets a number of jobs assigned which it has to process. The jobs consist of jobs from the same batch, but not necessarily all the jobs from a that batch. The given data consists of list of job identifiers and their corresponding raw data entries, plus the batch meta data. This method is called by the `GeneratorManager`.
    - Exceptions: None
- Startup/ShutDown
  - `void startUp(GeneratorId generatorId)`
    - Effect: Starts up the `Generator` instance an gives it the given `GeneratorId`.
    - Exceptions: None
  - `void shutDown()`
    - Effect: The `Generator` completes its assigned group of document generation jobs and report back completion to the `DocumentGenerationManager`, after which is shuts down.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `Generator` will respond to the ping request by sending an echo response. This is used by the `GeneratorManager` to check whether the `Generator` is available.
    - Exceptions: None

A.20 GeneratorManager

- **Description:** The `GenerationManager` is responsible for monitoring the `Generator` instances. It starts up or shuts down these instances based on the number of required instances indicated by the `Scheduler`.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- NotifyCompleted
  - `void notifyCompletedAndGiveMeMore(GeneratorId id)`
    - Effect: The `DocumentGenerationManager` gets notified that the document processing jobs assigned to the `Generator` identified by an `id` are completed.
    - Exceptions: None

A.21 JobDBShard

- **Description:** The `JobDBShard` is responsible for storing partition of all the jobs.
- **Super-component:** `JobManager`
- **Sub-components:** None

Provided interfaces

- JobDBShardingMgmt
- `void markAsReceived(DocumentId documentId) throws NoSuchJobExceptionException`
  - Effect: The `JobDBShard` marks the job corresponding to the document identified by the given document identifier as received by the recipient.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobDBShard` cannot find a job associated to a document identified by the given document identifier.
- `void markAsSent(DocumentId documentId) throws NoSuchJobExceptionException`
  - Effect: The `JobDBShard` marks the job corresponding to the document identified by the given document identifier as sent by the eDocs system.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobDBShard` cannot find a job associated to a document identified by the given document identifier.
- `void markAsFailed(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobDBShard` marks the document processing job identified by the given job identifier as failed.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobDBShard` cannot find a job identified by the given document identifier.
- `CustomerID getCustomerId(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobDBShard` returns the customer organization identifier of the customer organization corresponding to the job identified by the given job identifier.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobDBShard` cannot find a job associated to the given job identifier.
- `RawDataId getRawDataId(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobDBShard` returns the raw data entry identifier of the raw data entry corresponding to the job identified by the given job identifier.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobDBShard` cannot find a job associated to the given job identifier.
- `Boolean associateDocumentToJob(JobId jobId, DocumentId documentId) throws NoSuchJobExceptionExcep`
  - Effect: The `JobDBShard` will store the given document identifier in the job identified by the given job identifier. This is to associate a document processing job with the document it has generated. It returns true when it succeeds.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobDBShard` cannot find a job associated to the given job identifier.
- `DocumentId getDocumentId(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobDBShard` returns the document identifier of the document corresponding to the job identified by the given job identifier.
  - Exceptions:

        * NoSuchJobException: Thrown if the `JobDBShard` cannot find a job associated to the given job identifier.

- `BatchId getBatchId(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobDBShard` returns the batch identifier of the batch corresponding to the job identified by the given job identifier.
  - Exceptions:
    * NoSuchJobException: Thrown if the `JobDBShard` cannot find a job associated to the given job identifier.
- `DocumentType getDocumentType(JobId jobId)`
  - Effect: The `JobDBShard` returns the document type of the document corresponding to the job identified by the given job identifier.
  - Exceptions: None

A.22 JobDBShardingManager

- **Description:** The `JobDBShardingManager` manages the storage of the documents over multiple `JobDBShards`.
- **Super-component:** `JobManager`
- **Sub-components:** None

Provided interfaces

- JobDBMgmt
  - `void markAsReceived(DocumentId documentId) throws NoSuchJobExceptionException`
    * Effect: The `JobDBShardingManager` marks the job corresponding to the document identified by the given document identifier as received by the recipient.
    * Exceptions:
      · NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to a document identified by the given document identifier.
  - `void markAsSent(DocumentId documentId) throws NoSuchJobExceptionException`
    * Effect: The `JobDBShardingManager` marks the job corresponding to the document identified by the given document identifier as sent by the eDocs system.
    * Exceptions:
      · NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to a document identified by the given document identifier.
  - `void markAsFailed(JobId jobId) throws NoSuchJobExceptionException`
    * Effect: The `JobDBShardingManager` marks the document processing job identified by the given job identifier as failed.
    * Exceptions:
      · NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job identified by the given document identifier.
  - `CustomerID getCustomerId(JobId jobId) throws NoSuchJobExceptionException`
    * Effect: The `JobDBShardingManager` returns the customer organization identifier of the customer organization corresponding to the job identified by the given job identifier.
    * Exceptions:
      · NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to the given job identifier.
  - `RawDataId getRawDataId(JobId jobId) throws NoSuchJobExceptionException`
    * Effect: The `JobDBShardingManager` returns the raw data entry identifier of the raw data entry corresponding to the job identified by the given job identifier.

    - Exceptions:
        - NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to the given job identifier.
- `Boolean associateDocumentToJob(JobId jobId, DocumentId documentId) throws NoSuchJobExceptionE`
    - Effect: The `JobDBShardingManager` will store the given document identifier in the job identified by the given job identifier. This is to associate a document processing job with the document it has generated. It returns true when it succeeds.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to the given job identifier.
- `DocumentId getDocumentId(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobDBShardingManager` returns the document identifier of the document corresponding to the job identified by the given job identifier.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to the given job identifier.
- `BatchId getBatchId(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobDBShardingManager` returns the batch identifier of the batch corresponding to the job identified by the given job identifier.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobDBShardingManager` cannot find a job associated to the given job identifier.
- `DocumentType getDocumentType(JobId jobId)`
    - Effect: The `JobDBShardingManager` returns the document type of the document corresponding to the job identified by the given job identifier.
    - Exceptions: None

A.23 JobFacade

- **Description:** The `JobFacade` is responsible creating jobs and storing them using the `JobDBShadingManager` over different `JobDBShards`. It is responsible for retrieving data connected to a specific job. It can retrieve the raw data or customer organization info for specific jobs. The `JobFacade` is also used for marking jobs as sent and received.
- **Super-component:** `JobManager`
- **Sub-components:** None

Provided interfaces

- JobMgmt
    - `void setJobStatusAsTemporarilyFailed(List<JobId> statusesOfJobs)`
        - Effect: The `JobFacade` marks the job as "temporarily failed" for each of the jobs identified by the given `JobIds`. Used by the `DocumentGenerationManager` for jobs that where assigned to a failed `Generator` instance.
        - Exceptions: None
    - `RawData getRawData(JobId jobId)`
        - Effect: The `JobFacade` returns the raw data entry corresponding to the job identified by the given job identifier.
        - Exceptions: None
    - `List<Tuple<DocumentId, JobStatus>> getAllJobStatuses(CustomerId customerId)`
        - Effect: The `JobFacade` returns for all the documents ever initiated by the customer organization identified by the given customer organization identifier the document identifier and the job status.

    - Exceptions: None
- `List<Tuple<JobId, RawData>> getRawData(List<JobId> jobIds)`
    - Effect: The `JobFacade` returns the raw data entries corresponding to the jobs identified by the given job identifiers.
    - Exceptions: None
- `BatchMetaData getBatchMetaData(JobId jobId)`
    - Effect: The `JobFacade` returns the meta-data of the batch of the document processing job identified by the given job identifier.
    - Exceptions: None
- `DocumentType getDocumentType(JobId jobId)`
    - Effect: The `JobFacade` returns the document type of the document corresponding to the job identified by the given job identifier.
    - Exceptions: None
- `String getNameOfSender(JobId jobId)`
    - Effect: The `JobFacade` returns the name of the sender of the document corresponding to the job identified by the given job identifier.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job identified by the given job identifier.
- `PrintParameters getPrintParameters(JobId jobId) throws NoSuchJobException`
    - Effect: The `JobFacade` returns the print parameters belonging to the customer organization who sends the document corresponding to the the given job identifier.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job identified by the given job identifier.
- `void createJobsAndInitiateProcessing(Customerid cuid, TimeStamp deadline, List<rawdataId> rawDataIds, batchId batchId)`
    - Effect: The `JobFacade` creates a job for each of the raw data identifiers it gets as an argument. The job contains the time of deadline for the document processing job and the batch identifier. It stores the jobs and inserts the jobs in the `Scheduler`, together with their deadline and batch identifier.
    - Exceptions: None
- `void markAsReceived(DocumentId documentId) throws NoSuchJobExceptionException`
    - Effect: The `JobFacade` marks the job corresponding to the document identified by the given document identifier as received by the recipient.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to a document identified by the given document identifier.
- `void markAsSent(DocumentId documentId) throws NoSuchJobExceptionException`
    - Effect: The `JobFacade` marks the job corresponding to the document identified by the given document identifier as sent by the eDocs system.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to a document identified by the given document identifier.
- `void markAsFailed(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobFacade` marks the document processing job identified by the given job identifier as failed.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job identified by the given document identifier.

- `Boolean isReceiptTrackingOn(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobFacade` queries the `JobDBShardingManager` for the customer organization identifier belonging to the job identified by `jobId` and uses it to ask the `OtherDB` whether that customer organization has reciept tracking enabled. The `JobFacade` returns true when the customer organization has receipt tracking enabled and returns false otherwise.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to the given job identifier.
- `CustomerID getCustomerId(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobFacade` returns the customer organization identifier of the customer organization corresponding to the job identified by the given job identifier.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to the given job identifier.
- `Boolean associateDocumentToJob(JobId jobId, DocumentId documentId) throws NoSuchJobExceptionE`
  - Effect: The `JobFacade` will store the given document identifier in the job identified by the given job identifier. This is to associate a document processing job with the document it has generated. It returns true when it succeeds.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to the given job identifier.
- `Boolean isPartOfRecurringBatch(JobId jobId) throws NoSuchJobExceptionException`
  - Effect: The `JobFacade` checks whether the job identified by the given job identifier belongs to a recurring batch ant returns true if it does. Otherwise, it returns false.
  - Exceptions:
    - NoSuchJobException: Thrown if the `JobFacade` cannot find a job associated to the given job identifier.

A.24 JobManager

- **Description:** The `JobManager` is responsible creating jobs and storing them. It is responsible for retrieving data connected to a specific job. It can retrieve the raw data or customer organization info for specific jobs. The `JobFacade` is also used for marking jobs as sent and received.
- **Super-component:** None
- **Sub-components:** `JobFacade`, `JobDBShardingManager` and `JobDBShard`

Provided interfaces

- JobMgmt
  - `void setJobStatusAsTemporarilyFailed(List<JobId> statusesOfJobs)`
    - Effect: The `JobManager` marks the job as "temporarily failed" for each of the jobs identified by the given `JobIds`. Used by the `DocumentGenerationManager` for jobs that where assigned to a failed `Generator` instance.
    - Exceptions: None
  - `RawData getRawData(JobId jobId)`
    - Effect: The `JobManager` returns the raw data entry corresponding to the job identified by the given job identifier.
    - Exceptions: None
  - `List<Tuple<DocumentId, JobStatus>> getAllJobStatuses(CustomerId customerId)`
    - Effect: The `JobManager` returns for all the documents ever initiated by the customer organization identified by the given customer organization identifier the document identifier and the job status.

    - Exceptions: None
- `List<Tuple<JobId, RawData>> getRawData(List<JobId> jobIds)`
    - Effect: The `JobManager` returns the raw data entries corresponding to the jobs identified by the given job identifiers.
    - Exceptions: None
- `List<Tuple<DocumentId, JobStatus>> getAllJobStatuses(CustomerId customerId)`
    - Effect: The `JobManager` returns for all the documents ever initiated by the customer organization identified by the given customer organization identifier the document identifier and the job status.
    - Exceptions: None
- `BatchMetaData getBatchMetaData(JobId jobId)`
    - Effect: The `JobManager` returns the meta-data of the batch of the document processing job identified by the given job identifier.
    - Exceptions: None
- `DocumentType getDocumentType(JobId jobId)`
    - Effect: The `JobManager` returns the document type of the document corresponding to the job identified by the given job identifier.
    - Exceptions: None
- `String getNameOfSender(JobId jobId)`
    - Effect: The `JobManager` returns the name of the sender of the document corresponding to the job identified by the given job identifier.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job identified by the given job identifier.
- `PrintParameters getPrintParameters(JobId jobId) throws NoSuchJobException`
    - Effect: The `JobManager` returns the print parameters belonging to the customer organization who sends the document corresponding to the the given job identifier.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobFacade` cannot find a job identified by the given job identifier.
- `void createJobsAndInitiateProcessing(Customerid cuid, TimeStamp deadline, List<rawdataId> rawDataIds, batchId batchId)`
    - Effect: The `JobManager` creates a job for each of the raw data identifiers it gets as an argument. The job contains the time of deadline for the document processing job and the batch identifier. It stores the jobs and inserts the jobs in the `Scheduler`, together with their deadline and batch identifier.
    - Exceptions: None
- `void markAsReceived(DocumentId documentId) throws NoSuchJobExceptionException`
    - Effect: The `JobManager` marks the job corresponding to the document identified by the given document identifier as received by the recipient.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobManager` cannot find a job associated to a document identified by the given document identifier.
- `void markAsSent(DocumentId documentId) throws NoSuchJobExceptionException`
    - Effect: The `JobManager` marks the job corresponding to the document identified by the given document identifier as sent by the eDocs system.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobManager` cannot find a job associated to a document identified by the given document identifier.
- `void markAsFailed(JobId jobId) throws NoSuchJobExceptionException`

    - Effect: The `JobManager` marks the document processing job identified by the given job identifier as failed.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobManager` cannot find a job identified by the given document identifier.
- `Boolean isReceiptTrackingOn(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobManager` queries the `JobDBShardingManager` for the customer organization identifier belonging to the job identified by `jobId` and uses it to ask the `OtherDB` whether that customer organization has receipt tracking enabled. The `JobManager` returns true when the customer organization has receipt tracking enabled and returns false otherwise.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobManager` cannot find a job associated to the given job identifier.
- `CustomerID getCustomerId(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobFacade` returns the customer organization identifier of the customer organization corresponding to the job identified by the given job identifier.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobManager` cannot find a job associated to the given job identifier.
- `Boolean associateDocumentToJob(JobId jobId, DocumentId documentId) throws NoSuchJobExceptionE`
    - Effect: The `JobFacade` will store the given document identifier in the job identified by the given job identifier. This is to associate a document processing job with the document it has generated. It returns true when it succeeds.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobManager` cannot find a job associated to the given job identifier.
- `Boolean isPartOfRecurringBatch(JobId jobId) throws NoSuchJobExceptionException`
    - Effect: The `JobManager` checks whether the job identified by the given job identifier belongs to a recurring batch ant returns true if it does. Otherwise, it returns false.
    - Exceptions:
        - NoSuchJobException: Thrown if the `JobManager` cannot find a job associated to the given job identifier.

A.25 KeyCache

- **Description:** The `KeyCache` caches the keys which are most recently used for document generation. The `Completer` has to fetch a key every time a `Generator` instance requests new jobs, while the key will be the same for all jobs belonging to the same batch. The `KeyCache` avoids that the key storage system becomes a bottleneck for document generations. The keys are cached based on the `CustomerId` of a Customer Organization.
- **Super-component:** `DocumentGenerationManager`.
- **Sub-components:** None

Provided interfaces

- GetKey
    - `Key getKey(CustomerId customerId) throws NoSuchKeyException`
        - Effect: The `KeyCache` looks into its cache for the `Key` belonging to the customer organisation with id `customerId`. If the `Key` is in its cache, it returns it. If the `Key` is not in its cache, it asks `OtherDB` for the `Key` and stores it in its cache, after which it returns that `Key`.
        - Exceptions:
            - NoSuchKeyException: Thrown if there is no key for the given `customerId`.

A.26 LinkMappingDB

- **Description:** The `LinkMappingDB` is responsible for actually storing a mapping between unique links and `DocumentIds`. With this information, it also stores information about where a document can be found, i.e. only in the `DocumentDB` or both in the `PDSDB` and the `DocumentDB`.
- **Super-component:** `LinkMappingFunctionality`
- **Sub-components:** None

Provided interfaces

- LinkMappingDBMgmt
  - `void storeMapping(Link link, DocumentId documentId, Boolean isRegistered, TimeStamp whenGenerated)`
    - Effect: The `LinkMappingDB` stores the mapping between the given `link` and `documentId`. It also stores a `TimeStamp` with a value the time when the link was generated. The `TimeStamp` of the time when the mapping is generated is also stored with the mapping.
    - Exceptions: None
  - `Boolean isRegistered(Link link) throws NoSuchLinkException`
    - Effect: The `LinkMappingDB` returns true if the given link points to a document of a Registered Recipient and false if the given link points to a document of an Unregistered Recipient.
    - Exceptions:
      - NoSuchLinkException: Thrown if the given link cannot be found.
  - `Boolean getdocumentId(Link link) throws NoSuchLinkException`
    - Effect: The `LinkMappingDB` returns the document to which the given link maps.
    - Exceptions:
      - NoSuchLinkException: Thrown if the given link cannot be found.
  - `TimeStamp getMappingTime(Link link) throws NoSuchLinkException`
    - Effect: The `LinkMappingDB` returns the time when the mapping identified by the given link was generated.
    - Exceptions:
      - NoSuchLinkException: Thrown the given link cannot be found.

A.27 LinkMappingManager

- **Description:** The `LinkMappingManager` is responsible for creating unique links which points to a document. It also sends read and write requests to the `LinkMappingDB` to get and store the mappings between the unique links and the documents.
- **Super-component:** `LinkMappingFunctionality`
- **Sub-components:** None

Provided interfaces

- LinkMgmt
  - `Link createMapping(DocumentId docId, Boolean isRegistered)`
    - Effect: The `LinkMappingManager` generates a unique link for the document identified by the the given document identifier and stores the mapping between the document identifier and the link. This way, the recipient of the document can download it by following the link. The boolean `isRegistered` tells whether the recipient is registered or not. If the recipient is registered, the document can be found in the `PSDB`. Otherwise, it can be found in the `DocumentDB`.
    - Exceptions: None
  - `Boolean isRegistered(Link link) throws NoSuchLinkException`

    - Effect: The `LinkMappingManager` returns true if the given link points to a document of a Registered Recipient and false if the given link points to a document of an Unregistered Recipient.
    - Exceptions:
        - NoSuchLinkException: Thrown if the given link cannot be found.
- `DocumentId getDocumentIdUnregisteredRecipient(Link link) throws NoSuchLinkException, LinkExpiredException`
    - Effect: The `LinkMappingManager` returns the document identifier to which the given link points for an Unregistered Recipient. First, a check occurs to see whether the link is older than 30 days. If it is, an exception is thrown. Otherwise,the document identifier is returned.
    - Exceptions:
        - NoSuchLinkException: Thrown if the given link cannot be found.
        - LinkExpiredException: Thrown if the link is older than 30 days.
- `DocumentId getDocumentIdRegisteredRecipient(Link link) throws NoSuchLinkException`
    - Effect: The `LinkMappingManager` returns the document identifier to which the given link points for a Registered Recipient.
    - Exceptions:
        - NoSuchLinkException: Thrown if the given link cannot be found.

A.28 LinkMappingFunctionality

- **Description:** The `LinkMappingFunctionality` is responsible for creating unique links which point to documents. It is also responsible for storing these mappings and ultimately mapping a link to a document.
- **Super-component:** None
- **Sub-components:** `LinkMappingManager` and `LinkMappingFunctionality`

Provided interfaces

- LinkMgmt
    - `Link createMapping(DocumentId docId, Boolean isRegistered)`
        - Effect: The `LinkMappingFunctionality` generates a unique link for the document identified by the the given document identifier and stores the mapping between the document identifier and the link. This way, the recipient of the document can download it by following the link. The boolean `isRegistered` tells whether the recipient is registered or not. If the recipient is registered, the document can be found in the `PSDB`. Otherwise, it can be found in the `DocumentDB`.
        - Exceptions: None
    - `Boolean isRegistered(Link link) throws NoSuchLinkException`
        - Effect: The `LinkMappingFunctionality` returns true if the given link points to a document of a Registered Recipient and false if the given link points to a document of an Unregistered Recipient.
        - Exceptions:
            - NoSuchLinkException: Thrown if the given link cannot be found.
    - `DocumentId getDocumentIdUnregisteredRecipient(Link link) throws NoSuchLinkException, LinkExpiredException`
        - Effect: The `LinkMappingFunctionality` returns the document identifier to which the given link points for an Unregistered Recipient. First, a check occurs to see whether the link is older than 30 days. If it is, an exception is thrown. Otherwise,the document identifier is returned.
        - Exceptions:
            - NoSuchLinkException: Thrown if the given link cannot be found.
            - LinkExpiredException: Thrown if the link is older than 30 days.

- `DocumentId getDocumentIdRegisteredRecipient(Link link) throws NoSuchLinkException`
  - Effect: The `LinkMappingFunctionality` returns the document identifier to which the given link points for a Registered Recipient.
  - Exceptions:
    - NoSuchLinkException: Thrown if the given link cannot be found.

A.29 NotificationHandler

- **Description:** The `NotificationHandler` is responsible for sending notifications to the appropriate parties, e.g. the eDocs operators and the customer administrators.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- NotifyOperator
  - `void notifyOperatorOfPDSDBReplicaFailure(PDSDBReplicaId replicaId, TimeStamp dateTime)`
    - Effect: The `NotificationHandler` will send the given `PDSDBReplicaId` of the failed `PDSDBReplica` with the given time of failure `dateTime` to the eDocs operators. This method is called by a `PDSReplicationManager`.
    - Exceptions: None
  - `void notifyEdocsOperatorOfDocumentGenerationFailure(NotificationMessage msg, TimeStamp whenFailed)`
    - Effect: The `NotificationHandler` will send a textual message `msg` to the eDocs operators, which contains further information about the specific failure. This method is called by the `DocumentGenerationManager`. More specifically, it is called by the `GeneratorManager`.
    - Exceptions: None
  - `void notifyCustomerOrganizationOfInsufficientData(BatchMetaData, NotificationMessage msg, TimeStamp whenFailed)`
    - Effect: The `NotificationHandler` will send an e-mail message to the customer organization identified by the customer organization identifier in the `batchMetaData`, specifying that a document generation job has failed because the system did not have all of the data required to fill in the document template. This method is called by a `Generator` instance.
    - Exceptions: None

  notifyCustomerOrganizationOfZoomitReceiptFailure(cuid, documentId)
  - `void notifyCustomerOrganizationOfZoomitReceiptFailure(CustomerId cuid, DocumentId documentId)`
    - Effect: The `NotificationHandler` will send an e-mail message to the customer organization identified by the customer organization identifier, specifying that the delivery of the document identified by the given document identifier has failed because of a failed receipt at the Zoomit Service.
    - Exceptions: None
  - `void notifyCustomerOrganizationOfEmailDeliveryFailure(CustomerId cuid, JobId jobId)`
    - Effect: The `NotificationHandler` will send an e-mail message to the customer organization identified by the customer organization identifier, specifying that the delivery of the document corresponding to the job identified by the given job identifier has failed.
    - Exceptions: None

A.30 OtherDB

- **Description:** The `OtherDB` is responsible for storing all information that is not required to be stored separately by non-functional requirements. For example, it stores the raw data and data about customer organizations and registered recipients. It also stores the templates for documents and the keys of customer organizations to sign the documents during generation.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- GetKey
  - `Key getKey(CustomerId customerId)`
    - Effect: The `OtherDB` returns the key belonging to the Customer Organization identified by `customerId`.
    - Exceptions:
      - NoSuchKeyException: Thrown if there is no key for the given `customerId`.
  - `void storeKey(CustomerId customerId, Key key)`
    - Effect: The `OtherDB` stores the given key as the new key for the customer organization identified by the given customer organization identifier.
    - Exceptions:
      - NoSuchKeyException: Thrown if there is no key for the given `customerId`.
- TemplateMgmt
  - `Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived)`
    - Effect: The `OtherDB` returns the `Template` belonging to the customer organisation with id `customerId` corresponding to a document of type `documentType` and received at time `whenReceived`.
    - Exceptions:
      - NoSuchTemplateException: Thrown if there is no template for the given arguments.
  - `Boolean storeDocumentTemplate(CustomerId cuId, DocumentType documentType, Template template, TimeStamp whenReceived) throws InvalidDocumentTypeException`
    - Effect: The `OtherDB` stores the given template with the given time stamp for the customer organization identified by the given `CustomerId`.
    - Exceptions:
      - InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
- UserDataMgmt
  - `Credentials getRegisteredRecipientCredentials(RecipientId recipientId) throws NoSuchRecipientException`
    - Effect: The `OtherDB` returns the credentials belonging to the Registered Recipient identified by `recipientId`.
    - Exceptions:
      - NoSuchRecipientException: Thrown if no Registered Recipient with the given credentials exists.
  - `Credentials getCustomerOrganizationCredentials(CustomerId customerId) throws NoSuchCustomerOrganizationException`
    - Effect: The `OtherDB` returns the credentials belonging to the Customer Organization identified by `customerId`.
    - Exceptions:

        - NoSuchCustomerOrganizationException: Thrown if no Customer Organization with the customer organization identifier exists.

- `CustomerId storeCustomerOrganizationDetails(CustomerOrganizationDetails custOrgDetails)`
    - Effect: The `OtherDB` stores the given customer organization details when registering a new customer organization. It returns the customer identifier of the newly created customer organziation account.
    - Exceptions: None
- `void markCustomerOrganizationAsInactive(CustomerId customerId)`
    - Effect: The `OtherDB` marks the account of the customer organization identified by the given customer organization identifer as inactive.
    - Exceptions: None
- `void storePossibleDocumentTypes(CustomerId customerId, List<DocumentType> documentTypes)`
    - Effect: The `OtherDB` stores the given list of document types as the list of document types that the Customer Organization identified by `customerId` can generate.
    - Exceptions: None
- `List<Tuple<DocumentType, TimeStamp>> getPossibleDocumentTypes(CustomerId customerId)`
    - Effect: The `OtherDB` returns the a list of document types that the Customer Organization identified by `customerId` can generate. For each document type, it also returns the time when the last `Template` for that document type was uploaded.
    - Exceptions: None
- `Boolean isAllowedDocumentType(CustomerId cuid, DocumentType docType)`
    - Effect: The `OtherDB` checks in the SLA of the customer organization with customer identifier `cuid` if the document type `docType` is allowed. It returns true if it is allowed, otherwise it returns false.
    - Exceptions: None
- `void indicateWhichRecurringBatch(CustomerId cuid, BatchMetaData batchMetaData)`
    - Effect: The `OtherDB` stores to which of the recurring batches of the customer organisation identified by `cuid` this batch corresponds. The `batchMetaData` contains a description of which recurring batch this batch is.
    - Exceptions: None
- `int getNumberOfDocumentsMaximallyAllowed(CustomerId cuid, BatchMetaData batchMetaData)`
    - Effect: The `OtherDB` returns the number of documents that is maximally allowed for the customer organization identified by `cuid` for this recurring batch. The `batchMetaData` contains a description of which recurring batch this batch is.
    - Exceptions: None
- `Boolean checkIfRegistered(EmailAddress emailAddress)`
    - Effect: The `OtherDB` returns true if the given e-mail address belongs to a Registered Recipient. Otherwise, it returns false.
    - Exceptions: None
- `RecipientId getRecipientId(EmailAddress emailAddress)`
    - Effect: If the system has a Registered Recipient with the given e-mail address, the `OtherDB` returns its recipient identifier.
    - Exceptions:
        - NoSuchRecipientException: Thrown if the `OtherDB` does not contain the information of a Registered Recipient with the given e-mail address.
- `Boolean isReceiptTrackingOn(CustomerId cuid)`
    - Effect: The `OtherDB` checks if the customer organization with customer identifier `cuid` has receipt tracking turned on. It returns true if it has receipt tracking turned on, otherwise it returns false.
    - Exceptions: None

- `DocumentType getDocumentType(BatchId batchId)`
  - Effect: The `OtherDB` returns the document type of the batch identified by the given batch identifier.
  - Exceptions: None
- `Boolean isRecurringBatch(BatchId batchId)`
  - Effect: The `OtherDB` returns true if the bacth identified by the given batch is a recurring batch. Otherwise, it returns false.
  - Exceptions: None
- `String getNameOfSender(CustomerId cuId)`
  - Effect: The `OtherDB` returns the name of the sender of the document, i.e. the customer organization identified by the given customer organization identifier.
  - Exceptions: None
- `RecipientId storeRegisteredRecipient(RecipientDetails recipientDetails, Credentials credentials)`
  - Effect: The `OtherDB` creates a new Registered Recipient by storing his `recipientDetails` and `credentials`. It returns the `RecipientId` it created for the new Registered Recipient.
  - Exceptions: None
- `void removeRegisteredRecipient(RecipientId recipientId)`
  - Effect: The `OtherDB` removes the Registered Recipient's information from its memory.
  - Exceptions: None
- `Emailaddress getEmailAddress(CustomerId customerId)`
  - Effect: The `OtherDB` returns the e-mail address corresponding to the given customer organization identifier.
  - Exceptions: None
- `PrintParameters getPrintParameters(CustomerId customerId)`
  - Effect: The `JobFacade` returns the print parameters belonging to the customer organization identified by the given customer organization identifier.
  - Exceptions: None

- BatchDataMgmt
  - `BatchId storeBatchMetaData(BatchMetaData batchMetaData)`
    - Effect: The `OtherDB` stores the given `batchMetaData` and returns the batch identifier with which this meta-data can be queried.
    - Exceptions: None
  - `BatchMetaData getBatchMetaData(BatchId batchId)`
    - Effect: The `OtherDB` returns the meta-data if the batch identified by the given `batchId`.
    - Exceptions: None

- RawDataMgmt
  - `List<RawDataId> storeRawData(List<RawData> rawDataEntries)`
    - Effect: The `OtherDB` stores the given raw data entries. It returns the identifiers of the stored raw data entries.
    - Exceptions: None
  - `List<RawData> getRawData(List<RawDataId> rawDataIds)`
    - Effect: The `OtherDB` returns the raw data entries corresponding to the given raw data identifiers.
    - Exceptions: None
  - `RawData getRawData(RawDataId rawDataId)`
    - Effect: The `OtherDB` returns the raw data entry corresponding to the given raw data identifier.
    - Exceptions: None

A.31 PDSDB

- **Description:** The `PDSDB` component is responsible for storing the database of documents in the personal document stores. That database is separated from all other persistent data so that its failure *"does not affect the availability of other types of persistent data"*, as required by *Av2*.
- **Super-component:** None
- **Sub-components:** `PDSDBReplica`, `PDSLongTermDocumentManager`, `PDSReplicationManager`

Provided interfaces

- DocumentMgmt
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: The `PDSDB` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `List<Document> getAllDocumentMetaDataOf(RecipientId recipientId)`
    - Effect: The `PDSDB` will fetch and return all the meta-data of the documents belonging to the Registered Recipient identified by `recipientId`.
    - Exceptions: None
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `PDSDB` will store the given document `doc` together with the provided meta-data `md`.
    - Exceptions: None
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: The `PDSDB` will store the the given list of documents, together with their provided meta-data0
    - Exceptions: None
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
    - Effect: The `PDSDB` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions:
      - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None
  - `void removeAllDocuments(RecipientId recipientId)`
    - Effect: The `PDSDB` removes all the documents belonging to the Registered Recipient identified by the given recipient identifier.
    - Exceptions: None

A.32 PDSDBReplica

- **Description:** The `PDSDBReplica` is responsible for actually storing the documents.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: The `PDSDBReplica` will store the documents and their meta-data.
    - Exceptions: None
  - `void storeDocument(DocumentId documentId, Document document, DocumentMetaData md)`
    - Effect: The `PDSDBReplica` stores the given document with its `DocumentId` and meta-data.
    - Exceptions: None
  - `void removeAllDocuments(RecipientId recipientId)`
    - Effect: The `PDSDBReplica` removes all the documents belonging to the Registered Recipient identified by the given recipient identifier.
    - Exceptions: None
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: The `PDSDB` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
    - Effect: The `PDSDBReplica` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions: None
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None
- Ping
  - `Echo ping()`
    - Effect: The `PDSDBReplica` will respond to the ping request by sending an echo response. This is used by the `PDSReplicationManager` to check whether the `PDSDBReplica` is available.
    - Exceptions: None

A.33 PDSFacade

- **Description:** The `PDSFacade` is responsible for handling all read requests from Registered recipients to the `PDSDB`. It checks whether the documents requested by a Registered Recipient actually belong to him or her.
  It is also responsible for handling queries when Registered Recipients search for documents in their personal document store. It fetches the meta-data of the documents of the searching Registered Recipients, filters through that meta-data and returns an overview of the document meta-data and document identifiers matching the search criteria.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- PDSDBDocMgmt
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`

    - Effect: The `PDSFacade` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions:
        - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `Tuple<Document, DocumentMetaData> getDocument(DocumentId documentId, RecipientId recipientId) throws PDSUnavailableException, DocumentUnavailableException`
    - Effect: The `DocumentFacade` first checks whether the document identified by the given document identifier actually belongs to the registered recipient. If it does, it fetches the document from the `PDSDB` and returns it. Otherwise, it throws an exception.
    - Exceptions:
        - PDSUnavailableException: Thrown if the personal document store is unavailable.
        - DocumentUnavailableException: Thrown if the given document is not available for the given user. An example is when the document identified by the `documentId` exists, but the document does not belong to the Registered Recipient. Another example is when the document identifier does not exist.
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None

A.34 PDSLongTermDocumentManager

- **Description:** The `PDSLongTermDocumentManager` is responsible for managing the different storage clusters. Each cluster consists of a `PDSReplicationManager` and one or more `PDSDBReplica` instances. In the architecture, two clusters are defined. The `PDSLongTermDocumentManager` reads to and write from clusters, and periodically transfers documents from the one cluster to the other.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- DocumentMgmt
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: The `PDSDB` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `List<Document> getAllDocumentMetaDataOf(RecipientId recipientId)`
    - Effect: The `PDSLongTermDocumentManager` will fetch and return all the meta-data of the documents belonging to the Registered Recipient identified by `recipientId`.
    - Exceptions: None
  - `void removeAllDocuments(RecipientId recipientId)`
    - Effect: The `PDSLongTermDocumentManager` removes all the documents belonging to the Registered Recipient identified by the given recipient identifier.
    - Exceptions: None
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `PDSDB` will store the given document`doc` together with the provided meta-data `md`.
    - Exceptions: None
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: The `PDSLongTermDocumentManager` will store the documents and their meta-data.
    - Exceptions: None

- `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
  - Effect: The `PDSLongTermDocumentManager` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
  - Exceptions:
    - PDSUnavailableException: Thrown if the personal document store is unavailable.
- `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
  - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
  - Exceptions: None

A.35 PDSReplicationManager

- **Description:** The `PDSReplicationManager` is responsible for managing the `PDSDBReplicas`. The `PDSReplicationManager` passes read requests to one `PDSDBReplica` and writes to all `PDSDBReplicas`. It monitors their availabilty using the ping/echo.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `Tuple<Document, MetaData> getDocument(DocumentId documentId)`
    - Effect: The `PDSReplicationManager` will fetch and return the document corresponding to `DocumentId` id.
    - Exceptions: None
  - `void removeAllDocuments(RecipientId recipientId)`
    - Effect: The `PDSReplicationManager` removes all the documents belonging to the Registered Recipient identified by the given recipient identifier.
    - Exceptions: None
  - `List<Tuple<DocumentId, DocumentMetaData>> getAllDocumentMetaData(RecipientId recipientId) throws PDSUnavailableException`
    - Effect: The `PDSReplicationManager` fetches and returns the meta-data of all the documents of the Registered Recipient identified by `recipientId`.
    - Exceptions:
      - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `void storeDocument(DocumentId id, Document doc, DocumentMetaData md)`
    - Effect: The `PDSReplicationManager` will store the given document `doc` together with the provided meta-data `md`.
    - Exceptions: None
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: The `PDSReplicationManager` will store the given list of documents and their meta-data.
    - Exceptions: None
  - `Boolean isDocumentOfRecipient(DocumentId documentId, RecipientId recipientId)`
    - Effect: The `PDSDB` returns true when the document identified by the given document identifier belongs to the Registered Recipient identified by the given recipient identifier. Otherwise, it returns false.
    - Exceptions: None

A.36 Print&PostalServiceChannel

- **Description:** The `Print&PostalServiceChannel` is responsible for the printing the document and sending it by mail. It is external to the eDocs system and represents the servers of a print &postal service.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- DeliverMail
  - `void sendDocument(PostalAddress postalAddress, PrintParameters printParameters, Document document)`
    - Effect: The `PrintChannel` wll print the given document using the given print parameters and send it to the given postal address.
    - Exceptions: None

A.37 Print&PostalServiceFacade

- **Description:** The `Print&PostalServiceFacade` is responsible for delivering a document to the `Print&PostalChanne` so it can be printed and sent by mail. It also marks jobs as sent using the `JobManager`.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

Provided interfaces

- Deliver
  - `void sendDocument(PostalAddress postalAddress, PrintParameters printParameters, Document document)`
    - Effect: The `Print&PostalServiceFacade` sends the given document, the given postal address and the given print parameters to the `PrintChannel`. It then marks the document as sent using the `JobManager`.
    - Exceptions: None

A.38 RawDataHandler

- **Description:** The `RawDataHandler` is responsible for verifying the raw data and its entries. It forwards the validated raw data to the `JobManager` to create jobs.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- InputRawDara
  - `List<RawData> initiateDocumentProcessing(CustomerId coid, DocumentType docType, RawDataPacket rawDataPacket, DocumentPriority priority, Boolean isRecurring, TimeStamp isReceived) throws InvalidDocumentTypeException, InvalidRawDataPackageException, ToManyRawDataEntriesException`
    - Effect: The `RawDataHandler` verifies the received raw data. If it is correct, it stores the raw data and batch meta-data in the `OtherDB`. Next, it calculates the deadline of the document processing. It generates a `BatchMetaData` object containing information about the batch and stores it in the `OtherDB`. It also stores the valid raw data entries in the `OtherDB`. The `RawDataHandler` then informs the `JobFacade` to create jobs and to initiate document processing for the valid raw data entries. It returns a list of the invalid raw data entries.

        * Exceptions:
            · InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
            · InvalidRawDataPackageException: Thrown if the raw data package contains an error (e.g. an invalid Excel file or incorrectly formatted XML).
            · ToManyRawDataEntriesException: Thrown if the batch is a recurring batch and contains more raw data entries than maximally allowed for this batch.

A.39 RecipientClient

- **Description:** The `RecipientClient` is external to the eDocs system and represents a client device of an unregistered or registered recipient of eDocs that communicates with the eDocs System.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- RecipientMgmt
    - `Tuple<RecipientId, SessionId> getSessionDetails() throws NotAuthenticatedException`
        * Effect: The `RecipientClient` returns the recipient identifier and the session identifier of the current session if the recipient is a Registered Recipient and currently logged in. Otherwise, an exception is thrown.
        * Exceptions:
            · NoSessionException: Thrown if the recipient does not currently have a session with the eDocs system.
    - `Tuple<RecipientId, SessionId, Boolean> getUnregistrationConfirmation()`
        * Effect: The `RecipientClient` returns its recipient identifier, its session identifier and a boolean indicating whether the recipient confirms he or she wants to unregister or not.
        * Exceptions:
            · NoSessionException: Thrown if the recipient does not currently have a session with the eDocs system.

A.40 RecipientFacade

- **Description:** The `RecipientFacade` is responsible for the interaction of Registered and Unregistered Recipients with the eDocs system. It provides methods for authentication, for consulting the personal document store, for downloading documents, ... Another responsibility of the `UserFacade` is marking documents as received, since this is the last internal component through which the document is passed before the requesting recipient actually receives it and failure of another component in the document lookup process can no longer prevent this from happening.
- **Super-component:** `Userfunctionality`
- **Sub-components:** None

Provided interfaces

- RecipientIncoming
    - `SessionId login(Credentials credentials ) throws InvalidCredentialsException`
        * Effect: The `RecipientFacade` forwards the given `credentials` to the `AuthenticationHandler`, which verifies them and returns a new session identifier if correct. This session identifier can be used in future requests to the `RecipientFacade`.
        * Exceptions:
            · InvalidCredentialsException: Thrown if the `AuthenticationHandler` indicated that the given credentials where incorrect.

- `Boolean logout(SessionId sessionId)`
  - Effect: The `RecipientFacade` removes the session corresponding to the `sessionId` using the `AuthenticationHandler`. As a result, this session cannot be used anymore to access the system without logging in again. If no session corresponds to the `sessionId`, it does not exist, nothing is changed but no exception is thrown.
  - Exceptions: None
- `Tuple<RecipientId, SessionId> registerRecipient(RecipientDetails recipientDetails, Credential credentials) throws InvalidRecipientDetailsException, EmailAddressInUseException`
  - Effect: The `RecipientFacade` registers the recipient using his or her details and logs him or her in. It returns the session identifier of the created session between the newly registered recipient, plus the recipient identifier of the recipient.
  - Exceptions:
    - InvalidRecipientDetailsException: Thrown if the provided details are not valid, e.g. there are missing details. The exception lists the missing details to the Unregistered Recipient.
    - EmailAddressInUseException: Thrown if the given e-mail address in `recipientDetails` is already registered.
- `void unregister(SessionId sessionId, RecipientId recipientId) throws NotAuthenticatedExceptio`
  - Effect: The `RegistrationManager` asks the `RecipientClient` for confirmation of the unregistration after verifying the session identifier. If the Registered Recipient confirms and all the documents of the recipient have been read, his or her account gets deleted and his or her documents are removed from the `PDSDB`. Finally, the `RegistrationManager` logs the recipient out.
  - Exceptions:
    - NotAuthenticatedException: Thrown if the given session identifier is invalid.
- `PDSOverview getPDSOverview(SessionId sessionId, RecipientId recipientId) throws NotAuthenticatedException, PDSUnavailableException`
  - Effect: The `RecipientFacade` first verifies the given session identifier `sessionId` using the `AuthenticationHandler`. The `RecipientFacade` then requests all the document meta-data of the documents of the recipient identified by `recipientId` from the `PDSFacade`. It generates a document overview , which is returned to the caller.
  - Exceptions:
    - NotAuthenticatedException: Thrown if the given session identifier is invalid.
    - PDSUnavailableException: Thrown if the personal document store is unavailable.
- `Tuple<Document, DocumentMetaData> lookupDocument(RecipientId recipientId, SessionId sessionId, DocumentId documentId) throws NotAuthenticatedException, DocumentUnavailableExcept`
  - Effect: The `RecipientFacade` first verifies the given session identifier `sessionId` using the `AuthenticationHandler`. The `RecipientFacade` then gets the document identified by `documentId` and its meta-data using the `PDSFacade`, which checks whether the document belongs to the Registered Recipient, and returns them.
  - Exceptions:
    - NotAuthenticatedException: Thrown if the given session identifier is invalid.
    - DocumentUnavailableException: Thrown if the given document is not available for the given user. An example is when the document identified by the `documentId` exists, but the document does not belong to the Registered Recipient. Another example is when the document identifier does not exist.
- `Tuple<Document, DocumentMetaData> getDocument(Link link) throws NotAuthenticatedException, DocumentUnavailableException, LinkExpiredException`
  - Effect: The `RecipientFacade` returns the document and meta-data to which the unique link points. It first checks whether the link points to a document of a registered or unregistered recipient. If the document belongs to an Unregistered Recipient, a check occurs to see if the link is older than 30 days. If it is, an exception is thrown. If it is not, the document of the Unregistered Recipient gets marked as received and the `RecipientFacade` returns the document.

If the document belongs to an Registered Recipient, the `RecipientFacade` asks the Registered Recipient for its session identifier to check whether he or she is logged in and whether the document belongs to him or her. If those conditions are met, the document of the Registered Recipient gets marked as received and the `RecipientFacade` returns the document.

  - Exceptions:
    - NotAuthenticatedException: Thrown if the given session identifier is invalid. Also thrown if the link points to the document of a Registered Recipient and the recipient following the link currently has no session with the eDocs system.
    - DocumentUnavailableException: Thrown if the given document is not available for the given user. An example is when the document identified by the `documentId` exists, but the document does not belong to the Registered Recipient. Another example is when the document identifier does not exist.
    - LinkExpiredException: Thrown if the link is older than 30 calendar days and points to a document of an Unregistered Recipient.

A.41 RegistrationManager

- **Description:** The `RegistrationManager` is responsible for the registration or unregistered recipients and customer organizations. For the registration of unregistered recipients, the `RegistrationManager` gets called by the `RecipientFacade`, as recipients can register themselves. Customer organizations get registered by an eDocs operator, so the `EDocsAdminClient` calls those methods.
- **Super-component:** `UserFunctionality`
- **Sub-components:** None

Provided interfaces

- RegistrationMgmt
  - `void createRegisteredRecipient(RecipientDetails recipientDetails, Credentials credentials) throws InvalidRecipientDetailsException, EmailAddressInUseException`
    - Effect: The `RegistrationManager` creates a new Registered Recipient in the eDocs system. It first verifies whether `recipientDetails` contains all necessary details of the recipient. If it does, it then verifies if the eDocs system already has a recipient registered with the e-mail address in the given `recipientDetails`. If there is no such Registered Recipient, the `RegistrationManager` stores the Registered Recipient in the `OtherDB`. It uses the `DocumentStorageManager` to store the documents that were previously sent to the given e-mail address in the personal document store of the new Registered Recipient.
    - Exceptions:
      - InvalidRecipientDetailsException: Thrown if the provided details are not valied, e.g. there are missing details. The exception lists the missing details to the Unregistered Recipient.
      - EmailAddressInUseException: Thrown if the given e-mail address in **recipientDetails** is already registered.
  - `void unregisterRecipient(RecipientId recipientId)`
    - Effect: The `RegistrationManager` first checks whether all the document of the recipient identified by the given recipient identifier have been read. If they have been read, it will remove the account of the recipient from the `OtherDB` and remove his or her documents from the `PDSDB`.
    - Exceptions: None
  - `void registerCustomerOrganization(CustomerOrganizationDetails custOrgDetails, Credential credentials) throws InvalidCustomerOrganizationDetailsException, EmailAddressInUseException`
    - Effect: The `RegistrationManager` registers the Customer Organization using the given details.
    - Exceptions:
      - InvalidCustomerOrganizationDetailsException: Thrown if the provided details are not valid, e.g. there are missing details. The exception lists the missing details to the eDocs Administrator.

· EmailAddressInUseException: Thrown if the given e-mail address in **custOrgDetails** is already registered.

- `void unregister(RecipientId recipientId)`
  - Effect: The `RegistrationManager` will mark the account of the customer organizationas inactive.
  - Exceptions: None

A.42 Scheduler

- **Description:** The `Scheduler` receives the new jobs initiated by a Customer Organization and adds them to a queue of all jobs that have not been processed yet. To lower the size of this queue, the Scheduler is only given the information it needs, i.e., the id of the batch, its deadline and the ids of the individual jobs. The raw data of each job and the meta-data of the batch is stored in `OtherDB` and fetched by the `Completer` when needed.
  The `Scheduler` also indicates to the `GenerationManager` the number of required `Generator` instances through its GetStatistics interface.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetNextJobs
  - `Tuple<BatchId, List<JobId>> getNextJobs()`
    - Effect: The `Scheduler` returns the `JobIds` of the group of jobs that belong to the batch identified by `BatchId` that should be generated next. This method is called by the `GeneratorManager` when a `Generator` instance requires a new group of jobs.
    - Exceptions: None
  - `Tuple<BatchId, List<JobId>> jobsCompletedAndGiveMeMore(List<JobId>)`
    - Effect: The `Scheduler` gets notified that the document processing jobs belonging to the list of `JobIds` are completed. It returns the a list of `JobIds` belonging to a batch identified by `BatchId`. The returned list of `JobIds` identify document processing jobs which are not yet started.
    - Exceptions: None
- InsertJobs
  - `void insertJobs(BatchId batchId, TimeStamp deadline, List<JobId> jobIds )`
    - Effect: The `Scheduler` adds the jobs identified by their `JobIds` to its queue of all jobs that have not been processed yet. To lower the size of this queue, the `Scheduler` is only given the information it needs, i.e., the id of the batch, its deadline and the ids of the individual jobs. This method provides new jobs synchronously to the `Scheduler`, which it schedules synchronously. This means that when the method call returns, the given jobs are scheduled.
    - Exceptions: None
- GetStatistics
  - `int getNumberOfFutureJobs()`
    - Effect: The `Scheduler` returns the amount of documents that should be generated in the near future. The `GeneratorManager` queries this method at regular intervals and adjusts the number of `Generator` instances accordingly.
    - Exceptions: None

A.43 SessionDB

- **Description:** The `SessionDB` stores the session identifiers for currently active sessions.
- **Super-component:** `UserFunctionality`
- **Sub-components:** None.

Provided interfaces

- SessionMgmt
  - `RecipientId getRecipientId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `SessionDB` fetches and returns the Registered Recipient's identifier corresponding to the `sessionId` from the `sessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers, or if the session belongs to a customer organization.
  - `CustomerId getCustomerId(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `SessionDB` fetches and returns the Customer Organization's identifier corresponding to the `sessionId` from the `sessionDB`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier, or if the session belongs to a registered recipient.
  - `SessionId openSession(RecipientId recipientId)`
    - Effect: The `SessionDB` generates a new session identifier for the given `recipientId` and stores this as an active session.
    - Exceptions: None
  - `void closeSession(SessionId sessionId) throws NoSuchSessionException`
    - Effect: The `SessionDB` closes the active session associated with the given `sessionId`.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifier.
  - `Map<SessionAttributeKey, SessionAttributeValue> isvalidSession(SessionId sesionId) throws NoSuchsSessionException`
    - Effect: The `SessionDB` verifies whether a session with the given id exists in the `SessionDB` and if so, returns all its associated attributes.
    - Exceptions:
      - NoSuchSessionException: Thrown if no session exists with the given identifiers.

A.44 TemplateCache

- **Description:** The `TemplateCache` caches the templates which are most recently used for document generation. The `Completer` has to fetch a templateevery time a `Generator` instance requests new jobs, while the template will be the same for all jobs belonging to the same batch. The `TemplateCache` avoids that the template storage system becomes a bottleneck for document generations. The templates are cached based on the `CustomerId` of a Customer Organization, the type of the document and the date and time at which the batch was provided by the Customer Organization (in order to account for template updates).
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetTemplate
  - `Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived)`
    - Effect: The `TemplateCache` looks into its cache for the `Template` belonging to the customer organisation with id `customerId` corresponding to a document of type `documentType` and received at time `whenReceived`. If the `Template` is in its cache, it returns it. If the `Template` is not in its cache, it asks `OtherDB` for the `Template` and stores it in its cache, after which it returns that `Template`.
    - Exceptions:
      - NoSuchTemplateException: Thrown if there is no template for the given arguments.

A.45 UserFunctionality

- **Description:** The `UserFunctionality` is responsible for the interaction of registered recipients, unregistered recipients, customer organizations and eDocs operators with the eDocs system. It provides methods to register, to login and to logout, to consult the personal document store and download documents, to consult the status of document processing jobs, ...
- **Super-component:** None
- **Sub-components:** `RecipientFacade`, `CustomerOrganizationClient`, `EDocsadminfacade`, `RegistrationManager`, `AuthenticationHandler` and `SessionDB`

Provided interfaces

- RecipientIncoming
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    - Effect: The `UserFunctionality` returns a new session identifier if the given `Credentials` are correct. This session identifier can be used in future requests to the `UserFunctionality`.
    - Exceptions:
      - InvalidCredentialsException: Thrown if the `UserFunctionality` indicated that the given credentials where incorrect.
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `UserFunctionality` removes the session corresponding to the `sessionId`. As a result, this session cannot be used anymore to access the system without logging in again. If no session corresponds to the `sessionId`, it does not exist, nothing is changed but no exception is thrown.
    - Exceptions: None
  - `Tuple<RecipientId, SessionId> registerRecipient(RecipientDetails recipientDetails, Credential credentials) throws InvalidRecipientDetailsException, EmailAddressInUseException`
    - Effect: The `UserFunctionality` registers the recipient using his or her details and logs him or her in. It returns the session identifier of the created session between the newly registered recipient, plus the recipient identifier of the recipient.
    - Exceptions:
      - InvalidRecipientDetailsException: Thrown if the provided details are not valid, e.g. there are missing details. The exception lists the missing details to the Unregistered Recipient.
      - EmailAddressInUseException: Thrown if the given e-mail address in **recipientDetails** is already registered.
  - `PDSOverview getPDSOverview(SessionId, RecipientId recipientId) throws NotAuthenticatedException, PDSUnavailableException`
    - Effect: The `UserFunctionality` first verifies the given `sessionId`. The `UserFunctionality` then requests all the document meta data of the documents of the recipient identified by `recipientId` from the `PDSFacade`. It generates a document overview , which is returned to the caller.

    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - PDSUnavailableException: Thrown if the personal document store is unavailable.
  - `Tuple<Document, DocumentMetaData> lookupDocument(RecipientId recipientId, SessionId sessionId, DocumentId documentId) throws NotAuthenticatedException, DocumentUnavailableException`
    - Effect: The `UserFunctionality` first verifies the given `sessionId`. The `UserFunctionality` then gets the document identified by `documentId` and its meta-data using the `PDSFacade`, which checks whether the document belongs to the Registered Recipient, and returns them.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
        - DocumentUnavailableException: Thrown if the given document is not available for the given user. An example is when the document identified by the `documentId` exists, but the document does not belong to the Registered Recipient. Another example is when the document identifier does not exist.
  - `Tuple<Document, DocumentMetaData> getDocument(Link link) throws NotAuthenticatedException, DocumentUnavailableException, LinkExpiredException`
    - Effect: The `UserFunctionality` returns the document and meta-data to which the unique link points. It first checks whether the link points to a document of a registered or unregistered recipient. If the document belongs to an Unregistered Recipient, a check occurs to see if the link is older than 30 days. If it is, an exception is thrown. If it is not, the document of the Unregistered Recipient gets marked as received and the `UserFunctionality` returns the document.
      If the document belongs to an Registered Recipient, the `UserFunctionality` asks the Registered Recipient for its session identifier to check whether he or she is logged in and whether the document belongs to him or her. If those conditions are met, the document of the Registered Recipient gets marked as received and the `UserFunctionality` returns the document.
    - Exceptions:
        - NotAuthenticatedException: Thrown if the given session identifier is invalid. Also thrown if the link points to the document of a Registered Recipient and the recipient following the link currently has no session with the eDocs system.
        - DocumentUnavailableException: Thrown if the given document is not available for the given user. An example is when the document identified by the `documentId` exists, but the document does not belong to the Registered Recipient. Another example is when the document identifier does not exist.
        - LinkExpiredException: Thrown if the link is older than 30 calendar days and points to a document of an Unregistered Recipient.

- CustomerOrganizationIncoming
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `UserFunctionality` will delete the session with the given id. If no such session exists, nothing is changed and no exception is thrown.
    - Exceptions: None
  - `SessionId login(Credentials credentials) throws InvalidCredentialsException`
    - Effect: The `UserFunctionality` verifies the `credentials` using the `OtherDB`. If they are correct, the `UserFunctionality` creates a new session, stores the id of the user (i.e. the Registered Recipient id or Customer Organization id) as an attribute in this session and returns the id of the new session. The id of the user is present in the given credentials.
    - Exceptions:
        - InvalidCredentialsException: Thrown if the given credentials are invalid.
  - `List<Tuple<DocumentType,TimeStamp>> getPossibleDocumentTypes(SessionId sessionId, CustomerId cuId) throws NotAuthenticatedException`

    - Effect: The `UserFunctionality` will return a list of the document types that the customer organization is allowed to generate. It also indicates the current template for each document type by returning for each document type the time when the last template was uploaded.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `Boolean updateDocumentTemplate(SessionId sessionId, CustomerId cuId, DocumentType documentType, Template template) throws NotAuthenticatedException, InvalidDocumentTypeExcepti`
    - Effect: The `UserFunctionality` will update the current template for the given `documentType` to the given `template` for the customer organization identified by `cuId`. Returns true when it succeeds.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
      - InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
  - `List<RawData> initiateDocumentProcessing(CustomerId cuid, SessionId sessionId, DocumentType docType, RawDataPacket rawDataPacket, DocumentPriority priority, Boolean isRecurring) throws NotAuthenticatedException, InvalidDocumentTypeException, InvalidRawDataPackageException, ToManyRawDataEntriesException`
    - Effect: The `UserFunctionality` gets an indication from a customer organization with customer identifier `cuid` to start a batch of document processing jobs. The documents to be generated should be of document type `docType`. The boolean `isRecurring` indicates whether the batch is recurring or non-recurring. The information necessary to generate the documents is given in a `RawDataPacket` containing all the info about the raw data entries of the batch. After verification of the session identifier, the `UserFunctionality` generates a `TimeStamp` of the time when this method is called and forwards it together with most of its arguments to the `RawDataHandler`.
    - Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.
      - InvalidDocumentTypeException: Thrown if the given document type is invalid or not allowed for the given customer organization.
      - InvalidRawDataPackageException: Thrown if the raw data package contains an error (e.g. an invalid Excel file or incorrectly formatted XML).
      - ToManyRawDataEntriesException: Thrown if the batch is a recurring batch and contains more raw data entries than maximally allowed for this batch.

- EdocsAdminIncoming
  - `SessionId login(Credentials credentials ) throws InvalidCredentialsException`
    - Effect: The `UserFunctionality` verifies the given credentials and returns a new session identifier if correct. This session identifier can be used in future requests to the `UserFunctionality`.
    - Exceptions:
      - InvalidCredentialsException: Thrown if the `UserFunctionality` indicated that the given credentials where incorrect.
  - `Boolean logout(SessionId sessionId)`
    - Effect: The `UserFunctionality` removes the session corresponding to the `sessionId`. As a result, this session cannot be used anymore to access the system without logging in again. If no session corresponds to the `sessionId`, it does not exist, nothing is changed but no exception is thrown.
    - Exceptions: None
  - `void registerCustomerOrganization(SessionId edocsSessionid, CustomerOrganizationDetails custOrgDetails, Credential credentials) throws InvalidCustomerOrganizationDetailsException, EmailAddressInUseException, NotAuthenticatedException`
    - Effect: The `UserFunctionality` registers the Customer Organization using the given details.
    - Exceptions:

        - InvalidCustomerOrganizationDetailsException: Thrown if the provided details are not valid, e.g. there are missing details. The exception lists the missing details to the eDocs Administrator.
        - EmailAddressInUseException: Thrown if the given e-mail address in **custOrgDetails** is already registered.
        - NotAuthenticatedException: Thrown if the given session identifier is invalid.
  - `void unregister(SessionId edocsSessionid, RecipientId recipientId) throws NotAuthenticatedException`
    * Effect: The `UserFunctionality` asks the `EDocsAdminClient` for confirmation of the unregistration after verifying the session identifier. If the eDocs Administrator confirms, the account of the customer organization gets marked as inactive. Finally, the `UserFunctionality` logs the customer organization out.
    * Exceptions:
      - NotAuthenticatedException: Thrown if the given session identifier is invalid.

A.46 ZoomitChannel

- **Description:** The `ZoomitChannel` is responsible for delivering documents via Zoomit. It is external to the system and represents the servers of Zoomit to which a document can be sent.
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- DeliverZoomit
  - `void sendDocument(ZoomitId zoomitId, DocumentId docId, Document doc, String senderName, Boolean receiptTracking) throws ZoomitReceiptFailureException`
    * Effect: The `ZoomitChannel` sends the given document to the Zoomit user identified by the given Zoomit identifier. The document identifier is also sent, so the Zoomit can tell the System when the document is actually delivered to the addressee, if the customer organization has enabled receipt tracking.
    * Exceptions:
      - ZoomitReceiptFailureException: Thrown when the receipt of the document fails, e.g. because of an incorrect Zoomit account identifier or because this Zoomit user has indicates that he or she does not want to receive documents (of that particular customer organization) via Zoomit anymore.

A.47 ZoomitDeliveryCache

- **Description:** The `ZoomitDeliveryCache` is responsible for caching all documents,their document identifiers and corresponding `ZoomitIds` for which the `ZoomitFacade` did not yet receive an initial delivery confirmation from the `ZoomitChannel` up to a maximum of 2 days of documents.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

Provided interfaces

- ZoomitDeliveryCacheMgmt
  - `void storeDocument(ZoomitId zoomitId, DocumentId documentId, Document doc, String senderName, Boolean receiptTracking)`
    * Effect: The `ZoomitDeliveryCache` stores the given document together with the given information for up to a maximum of 2 days.
    * Exceptions: None

- `void removeDocument(DocumentId documentId)`
  - Effect: The `ZoomitDeliveryCache` removes the document identified by the given document identifier from its memory, together with the data corresponding to the document. If the cache does not contain a document identified by the given document identifier, this method has no effect.
  - Exceptions: None
- `Tuple<ZoomitId, DocumentId, Document, String, Boolean> getNext()`
  - Effect: The `ZoomitDeliveryCache` gets the next document in its memory, together with the data corresponding to the document.
  - Exceptions: None

A.48 ZoomitFacade

- **Description:** The `ZoomitFacade` is responsible for sending documents to Unregistered Recipients through Zoomit. It is also responsible for receiving messages from Zoomit when a document has been received by Zoomit or when a Zoomit user has received his or her document. The `ZoomitFacade` can use the `JobManager` to mark jobs as sent or received.
- **Super-component:** `DeliveryFunctionality`
- **Sub-components:** None

Provided interfaces

- Deliver
  - `void sendDocument(ZoomitId zoomitId, DocumentId docId, Document doc, String senderName, Boolean receiptTracking)`
    - Effect: The `ZoomitFacade` sends the given Zoomit identifier, document identifier, document and name of the sending customer organization to the `ZoomitClient`. It also sends a boolean `receiptTracking`, which is true when the customer organization has receipt tracking turned on. This way, the Zoomit Service knows when it should notify the the eDocs System when the document has actually been delivered to the addressee. If it succeeds, the `ZoomitFacade` will mark the document as sent. But if it fails, i.e., the `ZoomitChannel` throws an exception, then the `ZoomitFacade` will notify the customer organization of this failure.
    - Exceptions: None
- ConfirmZoomit
  - `void markAsDelivered(DocumentId document, TimeStamp whenDelivered)`
    - Effect: The `ZoomitFacade` gets notified that the document identified by the given document identifier is has been delivered by its addressee. The given `TimeStamp` indicates when the document had been delivered.

B Defined data types

List and describe all data types defined in your interface specifications. List them alphabetically for ease of navigation.

- `BatchId`: A piece of data uniquely identifying a batch of document processing jobs in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `BatchMetaData`: A data structure listing the meta-data belonging to a batch of jobs. This includes the `CustomerID` of a Customer Organization, the `DocumentType` of the documents to be generated, the `TimeStamp` of when the batch was received, the **DocumentPriority** of the documents, a `Boolean` specifying whether batch is recurring or not,...

- `CompletePartialBatchData`: A complex data structure listing all data a `Generator` needs to complete document generation jobs that are part of the same batch. It contains an array of `Tuple<JobId, RawData>`. The `JobIds` identify jobs that are all part of the same batch. The `RawData` belongs to these document processing jobs. Also listed in the `BatchMetaData` are the values of the `BatchMetaData`, `Key` and `Template` data types belonging to the batch.

  `CompletePartialBatchData` also contains a `BatchMetaData` entry, a `Key` and a `Template`. *Important to note:* a value of CompletePartialBatchData contains all information necessary to generate **some** jobs of belonging to same batch. It does not have to contain the information of all jobs belonging to same batch.

- `Credentials`: The authentication credentials of a Registered Recipient or Customer Organization. The credentials always contain an identifier of the recipient or customer organization and a proof of his or her identity. The architecture does not specify the specific credentials used, but a possibility is using a username and password.

- `CustomerId`: A piece of data uniquely identifying a Customer Organization in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL ...

- `CustomerOrganizationDetails`: A data structure specifying the details of a Customer Organization. It contains:

  - the name of the customer organization
  - the `PostalAddress` of the customer organization
  - the `EmailAddresses` and initial passwords of the Customer Administrators of the customer organization
  - whether the customer organization opts for receipt tracking for digital documents
  - the `DocumentTypes` that the customer organization is allowed to processs.
  - ... (see *UC18: register customer organization*)

- `DeliveryMethod`: A data structure representing the method with which a document should be delivered. At the moment, it can represent three possible values: e-mail, postal mail and Zoomit. The exact format is not specified by the architecture. During the delivery process, the `ChannelDispatcher` parses one value of this type out of the `RawData` of a document, on which it will decide which delivery method to use for sending the document.

- `Document`: A data file corresponding to a document. The architecture specifies the format of this data type as a PDF-file.

- `DocumentId`: A piece of data uniquely identifying a document in the system.

- `DocumentMetaData`: A data structure representing the meta-data stored with a document. It contains the name of the sender of the document, the `DocumentType`. Depending on where the meta-data is stored, it can contain different values. The meta-data stored with a document in the `DocumentDB` can contain the `EmailAddress` of the recipient, if the delivery method of the document was via e-mail. This way, the system can find all documents that where sent to a specific e-mail address when a recipient registers using that e-mail address. If the meta-data is stored in the `PDSDB`, the meta-data contains the `RecipientId` of the Registered Recipient to who the document belongs.
  Another value that is sometimes stored in the meta-data, is a `TimeStamp` of when the document is received. This `TimeStamp` is added to the meta-data when the document is received by the recipient. It is a final value, meaning it can only be written once. This way, it will not be overwritten when the recipient downloads a document multiple times. It exact moment when this `TimeStamp` is added to the meta-data is dependent om the method of document receipt.

- `DocumentPriority`: A data type representing the priority of document generation jobs. They have values representing the Critical, Diamond, Gold and Silver priorities. The exact format of this data type is not specified by the architecture.

- `DocumentType`: A piece of data describing the type of a document. This architecture does not specify the exact format of this data type, but possibilities are a long integer, a string, a URL ...

- `Echo`: The response to a ping message. This data element does not contain any meaningful data.
- `EmailAddress`: A data structure representing an e-mail address. The exact format is not specified by the architecture, but it could be a String.
- `EmailMessage`: A data structure representing an e-mail message. The exact format of the message is not defined by our architecture. It can contain a unique `Link` with which a document can be downloaded, if applicable. Other possible information it can contain is for example a short description of the received document (i.e., the sender of the document, the type of the document and the date at which the document was sent).
- `Error`: Description of data type.
- `GeneratordId`: A piece of data uniquely identifying a `Generator` in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `JobBatch`: Description of data type.
- `JobId`: A piece of data uniquely identifying a document processing job in the system.
- `Key`: A data structure containing the key of the Customer Organization which is used to sign its documents during the generation process. This architecture does not specify the exact format of this data type, but possibilities are a long integer, a string, a URL etc.
- `JobStatus`: A piece of data representing a job status in the system. Depending on the value, it represents a different document status. Possible document statuses are*initiated*, *generated*, *sent*, *failed*, *delivered*, ...
- `JobStatusOverview`: A data structure representing an overview of the status of all the document processing jobs that a customer organization has ever initiated. For each job belonging to the customer organization, the data structure contains the corresponding document identifier and the status messages of the job, e.g. initiated, generated, failed, sent, received, ...
- `Link`: A data structure representing a unique link. The system uses these links to point to documents. This way, a recipient can download a document by following a link. the exact format is not specified, but a possibility is just a String, since the link is a URL.
- `NotificationMessage`: A textual message which can be used to include extra information about the event of the notification.
- `PDSOverview` The overview of a personal document store that can be shown to a Registered Recipient. Through this overview, the Registered Recipient can consult documents. The architecture does not specify the exact format of such an overview, but a likely possibility is an HTML page.
- `PDSDBReplicaId`: A piece of data uniquely identifying a `PDSDBReplica` in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `PostalAddress`: A data structure containing a postal address. The exact format of this data structure is unspecified.
- `PrintParameters`: A data structure containing the print parameters that the system gives to a Print & Postal Service. Examples of what these parameters can contain are the paper type to be used, whether the documents should be printed single-sided or double-sided, whether the documents should be printed in colour or in black and white ... The exact format of this data structure is not specified by the architecture
- `RawData`: A data structure listing a raw data entry used in a document processing job.
- `RawDataPacket`: A data structure containing all the raw data entries in a batch, given by the customer organization to the `CustomerOrganizationFacade`. The architecture does not specify this data structure, but possibilities are an XML file or Excel file.
- `RawDataId`: A piece of data uniquely identifying a raw data entry in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.

- `RecipientId`: A piece of data uniquely identifying a Registered Recipient in the system. This architecture does not specify the exact format of this identifier, but possibilities are a long integer, a string, a URL etc.
- `RecipientDetails`: A data structure containing the details of a recipient. A recipient has to provide these details at registration. The details contain the first name, the last name, the e-mail address and the postal address of the recipient.
- `SessionId`: A piece of data uniquely identifying a session of a registered recipient of customer organization in the eDocs system. This contains at least the user identifier (i.e. the `CustomerId` for a customer organization or the `RecipientId` for a registered recipient) and the time the session was initiated.
- `SessionAttributeKey`: The key of an attribute attached to a session. This architecture does not specify the exact format of this key. a possible value is a long integer or a flat string.
- `SessionAttributeValue`: The value of an attribute attached to a session. This value van be of any primitive type.
- `TimeStamp`: The representation of a time (i.e. date and time of day) in the system.
- `Template`: A document used as a template for the generation of documents.
- `TemplateId`: A data structure uniquely identifying a template in the system. It lists three values. It contains `CustomerId` which identifies the Customer Organization who the template belongs to. It also contains a `DocumentType`, specifying for which kind of document it is a template for. The last piece of information it contains is a `TimeStamp` specifying when the system received the template.
- `ZoomitId`: A data structure uniquely identifying a Zoomit account in the external Zoomit system. It consists of a number, but the actual information it must be able to contain is not specified by the architecture.