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 develop an architecture for a system for document processing. This part of the project consisted of

2 Overview

2.1 Architectural decisions

Briefly discuss your architectural decisions for each non-functional requirement. Pay attention to the solutions that you employed (in your own terms or using tactics and/or patterns).

ReqX: requirement name Provide a brief discussion of the decisions related to *ReqX*.
Employed tactics and patterns: List all patterns and tactics used to achieve ReqX, if any.

2.2 Discussion

Use this section to discuss your architecture in retrospect. For example, what are the strong points of your architecture? What are the weak points? Is there anything you would have done otherwise with your current experience? Are there any remarks about the architecture that you would give to your customers? Etc.

3 Attribute-driven design documentation

3.1 Decomposition 1: eDocs (X1, Y3, UCa, UCb, UCc)

3.1.1 Module to decompose

In the first run, the eDocs System is decomposed as a whole

3.1.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *X1*: name
- *Y3*: name

The related functional drivers are:

- *UCa*: name
- *UCb*: name
- *UCc*: name

Rationale A short discussion of why these drivers were selected for this decomposition.

3.1.3 Architectural design

Topic Discussion of the solution selected for (a part of) one of the architectural drivers.

Alternatives considered

Alternatives for solution A discussion of the alternative solutions and why that were not selected.

3.1.4 Instantiation and allocation of functionality

Decomposition Main aspects of the resulting decomposition.

ModuleB Per introduced component a paragraph describing its responsibilities.

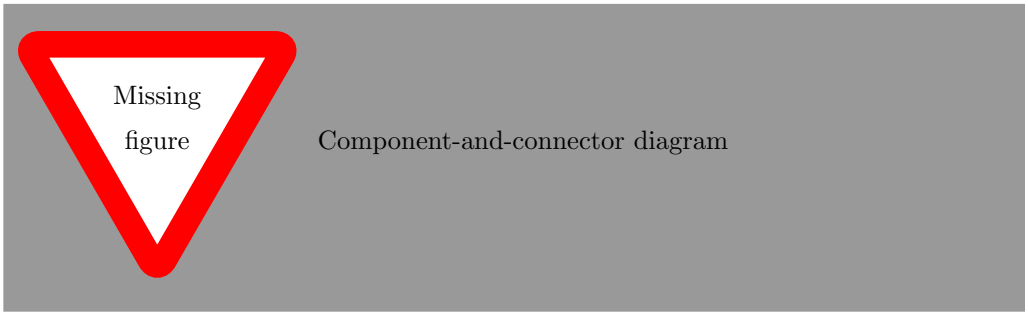


Figure 1: Component-and-connector diagram of this decomposition.

ModuleC Per introduced component a paragraph describing its responsibilities.

Behaviour If needed and explanation of the behaviour of certain aspects of the design so far.

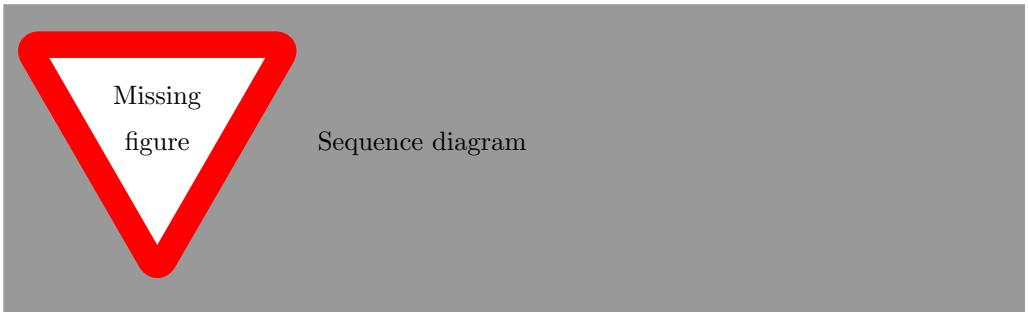


Figure 2: Sequence diagram illustrating a key behavioural aspect.

Deployment Rationale of the allocation of components to physical nodes.

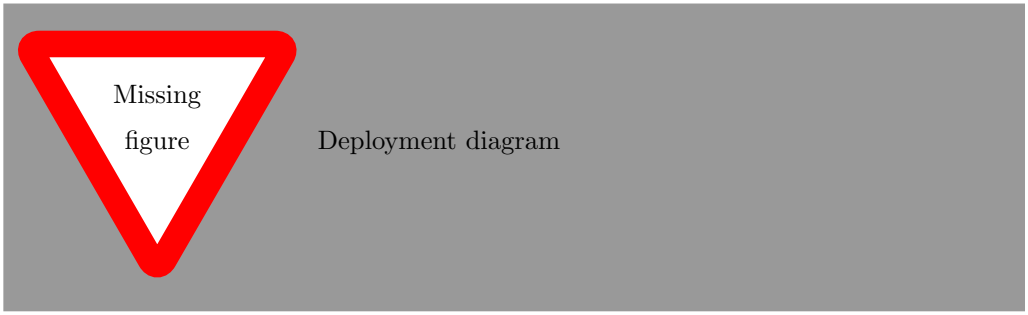


Figure 3: Deployment diagram of this decomposition.

3.1.5 Interfaces for child modules

PDSDB

- DocumentMgmt
  - `void storeDocument(DocumentId id, Document doc, MetaData md)`
    - Effect: The `PDSDB` will store the given document`doc` together with the provided metadata `md`.
    - Exceptions: None
  - `Tuple<Document, MetaData> getDocument(DocumentId id)`
    - Effect: Describe the effect of calling this operation.

    - Exceptions: None
- `void markReceived(DocumentId id)`
    - Effect: Describe the effect of calling this operation.
    - Exceptions: None

3.1.6 Data type definitions

Describe per complex data type used in the interfaces what it represents.

returnType This data element represents X.

ParamType This data element represents Y.

3.1.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

ModuleB

- *Z1*: name
- *UCd*: name

ModuleC

- *UCba*: name
  Description which part of the original use case is the responsibility of this component.

3.2 Decomposition 2: OtherFunctionality(P2, Av2b, UC12, UC13, UC14, UC15)

3.2.1 Module to decompose

In this run we decompose `Otherfunctionality`.

3.2.2 Selected architectural drivers

The non-functional driver for this decomposition is:

- *P2*: Document lookups
- *Av2b*: Temporary storage for PDSDB

The related functional drivers are:

- *UC12*: Consult personal document store
- *UC13*: Search documents in personal document store
- *UC14*: Consult document in personal document store
- *UC15*: Download document via unique link

Rationale P2 and Av2b were chosen because their priorities are amongst the highest ones of all remaining non-functional drivers and the domains on which their focus lies complement the previous decomposition perfectly.

3.2.3 Architectural design

Link mapping In order to support document lookups via links, which are either unique (unregistered recipient) or part of a notification e-mail (registered recipient), the `UserFacade` is concerned with the mapping of each incoming link request to the appropriate document, which is either located in the `PDSDB` (for registered recipients) or in the `DocumentDB` (for unregistered recipients). The mappings themselves are stored in the `MappingDB` component and are fetched by the `UserFacade` when needed.

Dedicated DocumentDB and LinkMappingDB for P2 Since a large number of document lookups and downloads should not affect the performance of other functionality of the system, both link mappings and documents are stored in dedicated databases, each deployed on a different node. This decision ensures that documents can be looked up via the personal document store or a notification in a timely fashion, because it prohibits either of those two components to be a bottleneck in the document lookup process. Since, according to the previous decomposition, the `PDSDB` is deployed on a separate node too, this component's performance is already satisfactory and no changes need to be made to improve it.

Sharding for DocumentDB for P2 In order to improve performance of the `DocumentDB`, we chose to partition this database into multiple shards. This approach was driven by the need for fast response time while keeping in mind the high storage cost for documents. More precisely, a `ShardingManager` is responsible for reading and writing all documents in one of the `DocumentDBShards`, making sure that every document is stored only once. This sharding technique provides roughly the same advantages in response time as active replication, but is significantly less costly when it comes to storage capacity. Note that there must also be a (sub)component monitoring all requests to the shards, in order to throttle excessive requests when necessary. Since this is a rather simple task and the (sub)component must be aware of the details concerning the sharding architecture to efficiently fulfil its purpose, this functionality is delegated to the `ShardingManager` itself. Finally, this `ShardingManager` is also responsible for implicitly pinging all `DocumentDBShards` upon reading and writing in them.

DocumentStorageCache for Av2b This component temporarily stores the IDs of all generated documents that are to be delivered through the personal document store during downtime of the `PDSDB` component up to a maximum of 3 hours. When the latter component turns operational again, the `DocumentStorageManager` retrieves all documents and their corresponding meta data from the `DocumentDB` using the previously mentioned IDs and subsequently stores them in the `PDSDB`. Note that the recipient therefore perceives a maximum total downtime of 3 hours plus the time needed for the `DocumentStorageManager` to transfer all documents and meta data that the cached IDs refer to.
Note that the `DocumentDB` and the `PDSDB` store documents in exactly the same way, both including the document itself, its meta data and its ID, although there is no need for the meta data in the former database. This storage tactic ensures that the `DocumentStorageManager` does not have to fetch or convert any information when transferring documents between both databases, making the transfer as efficient as possible. This does not introduce any overhead to the storage system, since the meta data is but a fraction of the document data. Finally, we would like to stress that 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 converted into document ID writes to the aforementioned cache. Upon revival of the `PDSDB`, this database will send a single heartbeat to the `DocumentStorageManager`, causing this component to begin transferring the missing document data. Once all cached document IDs are processed, subsequent writes to the `PDSDB` will no longer be redirected through the `DocumentStorageCache`.

UserFacade We introduce this component to be able to distinguish between registered recipients and unregistered recipients in the first steps of the document lookup process. The `UserFacade` also maps incoming links to documents that are located either in the `PDSDB` or in the `DocumentDB` by first fetching the correct mapping from the `LinkMappingDB` mentioned above. 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 pipeline can no longer prevent this from happening.

PDSFacade This component handles all read requests that are intended for the personal document store. This extra level of indirection calculates and aggregates all intermediate query results as to relieve the `PDSDB`

of this burden, which is also the reason why both components are best deployed on different nodes. Upon querying the personal document store, this component first collects the recipient's documents' meta data and subsequently performs queries on this collection. This approach introduces no significant overhead, because only those documents that are needed, will be fetched in their entirety. It is also the responsibility of the `PDSFacade` to check the recipient's ID against the recipient ID that can be retrieved from the document meta data and to only pass on those documents for which they are a match.

DocumentStorageManager The storage of documents is handled by the `DocumentStorageManager`, which receives generated documents from the intermediary `OtherFunctionality2` component and stores them in either the `DocumentDB` (for unregistered recipients) or in both the `DocumentDB` and the `PDSDB` (for registered recipients) according to the method that is invoked on it. The necessity for this component follows from the fact that synchronisation is needed between the two storage components mentioned above. Note that the PDSDocMgmt interface in the `PDSDB` component now requires an extra method to save documents.

Alternatives considered

Alternative for DocumentStorageCache We could just as well do without the previously introduced `DocumentStorageCache` by letting the `DocumentStorageManager` actively look for documents in the `DocumentDB` that are not yet, but should be, present in the `PDSDB` upon revival of the `PDSDB`. 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 approach was not chosen.

3.2.4 Instantiation and allocation of functionality

Decomposition Figure 4 shows the components resulting from the decomposition in this run. Extra attention is required concerning the `DocumentDBShard`. This component actually represents multiple instances, each containing a different partition of the document data to accommodate for parallel lookups in different partitions while minimizing storage costs. The responsibilities of the resulting components are as follows:

DocumentDB Responsible for storing all generated documents.

ShardingManager Responsible for managing all reads and writes to the `DocumentDBShards`.

DocumentDBShard Responsible for storing a partition of the documents in the `DocumentDB`

LinkMappingDB Responsible for storing the mappings between links and document locations.

UserFacade We introduce this component to be able to distinguish between registered recipients and unregistered recipients in the first steps of the document lookup process. The `UserFacade` also maps incoming links to documents that are located either in the `PDSDB` or in the `DocumentDB` by first fetching the correct mapping from the `LinkMappingDB` mentioned above. 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 pipeline can no longer prevent this from happening.

PDSFacade This component handles all read requests that are intended for the personal document store. This extra level of indirection calculates and aggregates all intermediate query results as to relieve the `PDSDB` of this burden, which is also the reason why both components are best deployed on different nodes. Upon querying the personal document store, this component first collects the recipient's documents' meta data and subsequently performs queries on this collection. This approach introduces no significant overhead, because only those documents that are needed, will be fetched in their entirety. It is also the responsibility of the `PDSFacade` to check the recipient's ID against the recipient ID that can be retrieved from the document meta data and to only pass on those documents for which they are a match.

DocumentStorageManager The storage of documents is handled by the `DocumentStorageManager`, which receives generated documents from the intermediary `OtherFunctionality2` component and stores them in either the `DocumentDB` (for unregistered recipients) or in both the `DocumentDB` and the `PDSDB` (for registered recipients) according to the method that is invoked on it. The necessity for this component follows from the fact that synchronisation is needed between the two storage components mentioned above. Note that the PDSDocMgmt interface in the `PDSDB` component now requires an extra method to save documents.

DocumentStorageCache Responsible for temporarily storing the IDs of all generated documents that are to be delivered through the personal document store during downtime of the `PDSDB` component.

OtherFunctionality2 Encapsulates requirements that are not tackled in this run and cannot be assigned to other introduced components.

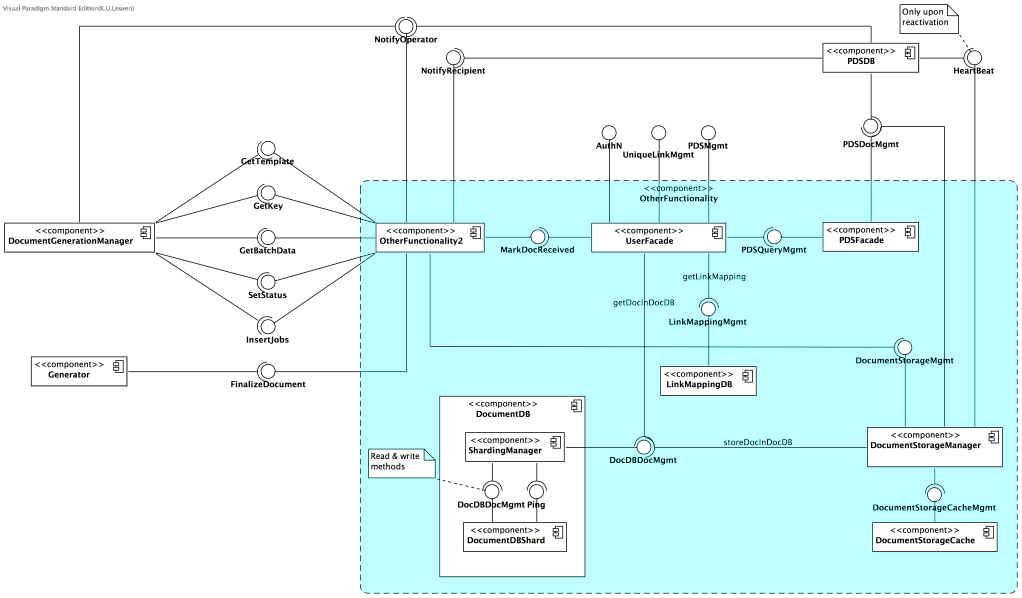


Figure 4: Component-and-connector diagram of the second decomposition.

Behaviour If needed and explanation of the behaviour of certain aspects of the design so far.

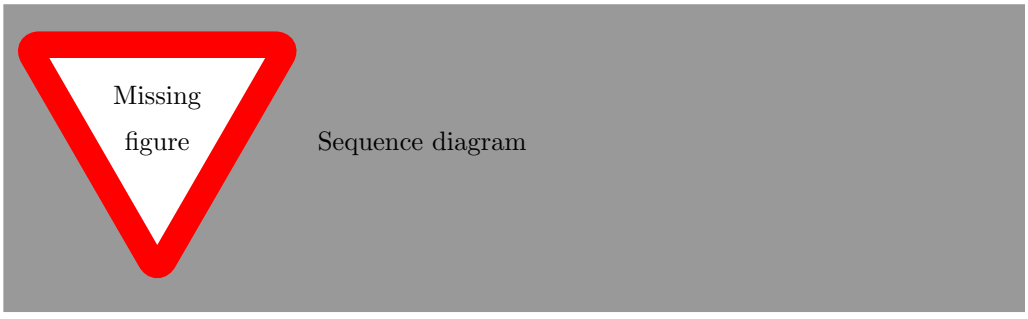


Figure 5: Sequence diagram illustrating a key behavioural aspect.

Deployment Rationale of the allocation of components to physical nodes.

3.2.5 Interfaces for child modules

UserFacade

The `UserFacade` provides three interfaces to the recipient: *AuthN* for authentication, *UniqueLinkMgmt* for document lookups via unique links and *PDSDBMgmt* for access to the personal document store. Links to

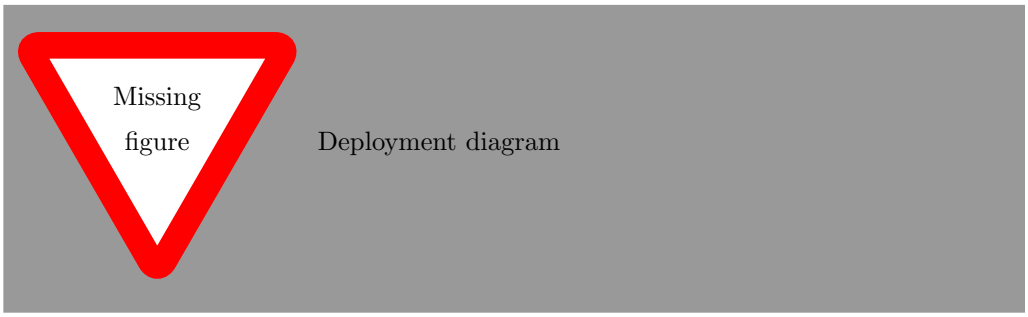


Figure 6: Deployment diagram of this decomposition.

documents in the PDSDB that are part of a notification e-mail are therefore handled by the latter, in which case the `UserFacade` also needs to perform a check to see if the requesting recipient is registered in the eDocs system.

- InterfaceA
  - `returnType operation1(ParamType param1)` throws TypeOfException
    - Effect: Describe the effect of calling this operation.
    - Exceptions:
      - TypeOfException: Describe when this exception is thrown.
  - `returnType operation2()`
    - Effect: Describe the effect of calling this operation.
    - Exceptions: None

3.2.6 Data type definitions

Describe per complex data type used in the interfaces what it represents.

returnType This data element represents X.

ParamType This data element represents Y.

3.2.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

Residual Av2b2 : TIJDIGE notification AAN DE USER

USE CASE 12: residual drivers: het opslaan van de recipient id's in de pdsdb –¿ hoort bij deliveryfunctionality. –¿ voorlopig verondersteld dat recipientID in meta data zit die samen met doc opgeslagen wordt in zowel docDB als PDS Ook het inloggen (authenticatie). USE CASE 13: residual drivers: we veronderstellen dat de DocumentStorageManager metadata bij elk document opslaat, like the name of the sender, the data range in which the document should be received and the document type –¿ hoort bij deliveryfunctionality. The metadata is saved in both the pdsdb en database components, because when a user registers, this metadata has to be copied form the one to the other. Dit is om gemakkelijk de documenten in de pdsdb te kunnen zoeken. USE CASE 14: residual driver: mark as received, puntje 3. –¿ beter te doen bij de deliveryfunctionality USE CASE 15: residual driver: tracking.

ModuleB

- *Z1*: name
- *UCd*: name

ModuleC

- *UCba*: name
  Description which part of the original use case is the responsibility of this component.

3.3 Decomposition 3: UserFacade (M1, UC1, UC2, UC3)

NIET VERWIJDEREN WANT LINKMANAGER STAAT HIER AL ERGENS TUSSEN

3.3.1 Module to decompose

In this run we decompose `ModuleA`.

3.3.2 Selected architectural drivers

The non-functional drivers for this decomposition are:

- *X1*: name
- *Y3*: name

The related functional drivers are:

- *UCa*: name
- *UCb*: name
- *UCc*: name

Rationale A short discussion of why these drivers were selected for this decomposition.

3.3.3 Architectural design

Topic Discussion of the solution selected for (a part of) one of the architectural drivers.

LinkManager VOORLOPIG NIET NODIG IN DEZE DECOMPOSITIE
Motivatie voor LinkManager: Checks expiration date ALS DAT NODIG IS–¿ reason: links naar de pdsdb vervallen niet (zolang de gebruik geregistreerd is) LinkManager maps link to (document ID, place where the document is stored)-pairs –¿ REASON: the unique link has two possible sources: an e-mail to an unregistered recipient or an email to a registered recipient. For an unregistered recipient, the RecipientFacade must look with the documentid for the document in the documentDB. For a registered recipient, the RecipientFacade must look with the documentid for the document in the PDSDB. (Mogelijk een boolean ofzo)
Does NOT do mapping removal after x years –¿ there has to be a notification when the link has expired

Alternatives considered

Alternatives for solution A discussion of the alternative solutions and why that were not selected.

3.3.4 Instantiation and allocation of functionality

Decomposition Main aspects of the resulting decomposition.

ModuleB Per introduced component a paragraph describing its responsibilities.

ModuleC Per introduced component a paragraph describing its responsibilities.

Behaviour If needed and explanation of the behaviour of certain aspects of the design so far.

Deployment Rationale of the allocation of components to physical nodes.

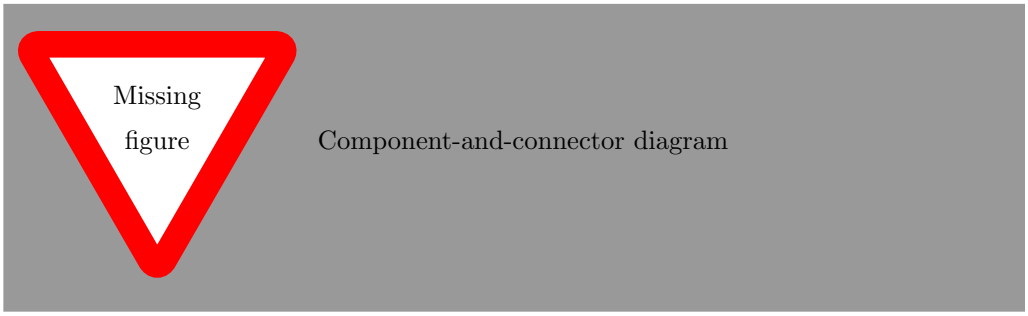


Figure 7: Component-and-connector diagram of this decomposition.

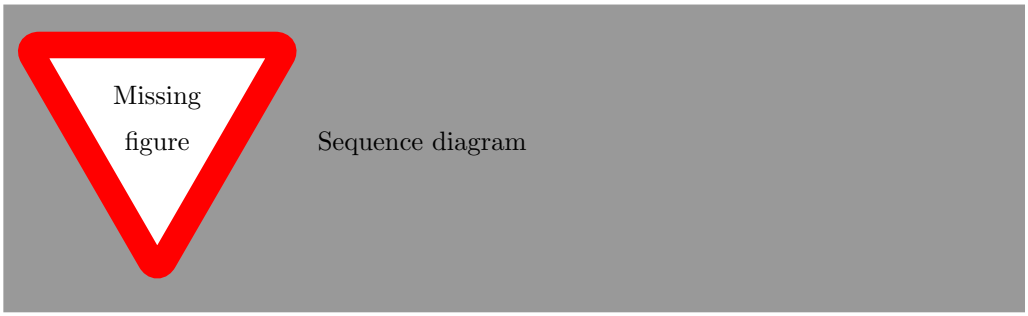


Figure 8: Sequence diagram illustrating a key behavioural aspect.

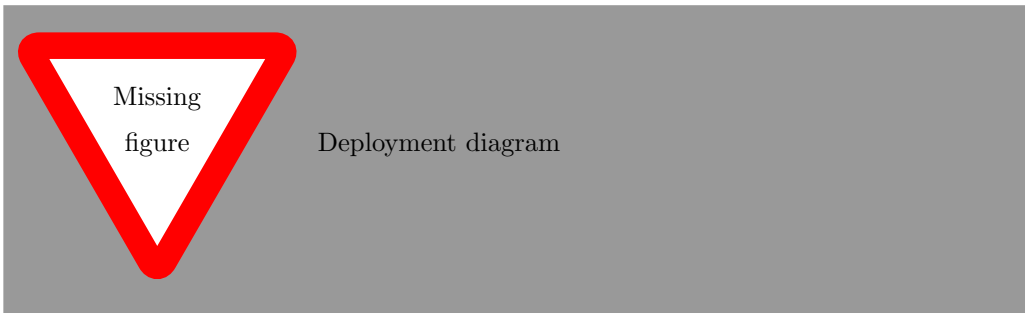


Figure 9: Deployment diagram of this decomposition.

3.3.5 Interfaces for child modules

ModuleB

- InterfaceA
  - `returnType operation1(ParamType param1)` throws TypeOfException
    - Effect: Describe the effect of calling this operation.
    - Exceptions:
      - TypeOfException: Describe when this exception is thrown.
  - `returnType operation2()`
    - Effect: Describe the effect of calling this operation.
    - Exceptions: None

3.3.6 Data type definitions

Describe per complex data type used in the interfaces what it represents.

returnType This data element represents X.

ParamType This data element represents Y.

3.3.7 Verify and refine

This section describes per component which (parts of) the remaining requirements it is responsible for.

ModuleB

- *Z1*: name
- *UCd*: name

ModuleC

- *UCba*: name
  Description which part of the original use case is the responsibility of this component.

4 Client-server view (UML Component diagram)

The context diagram of the client-server view. Discuss which components communicate with external components and what these external components represent.

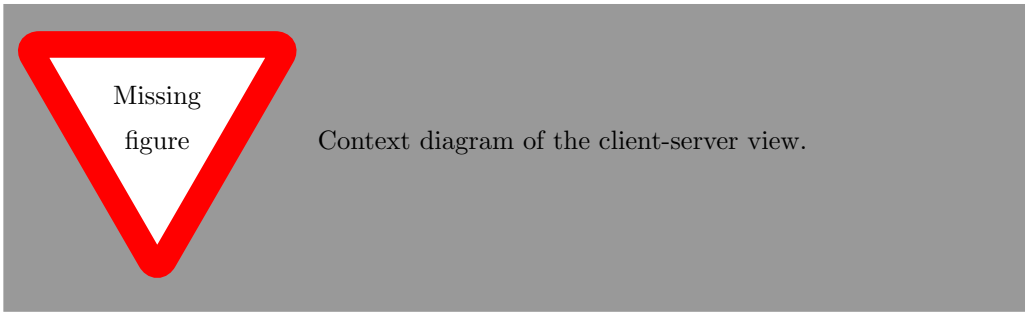


Figure 10: Context diagram for the client-server view.

The primary diagram and accompanying explanation.

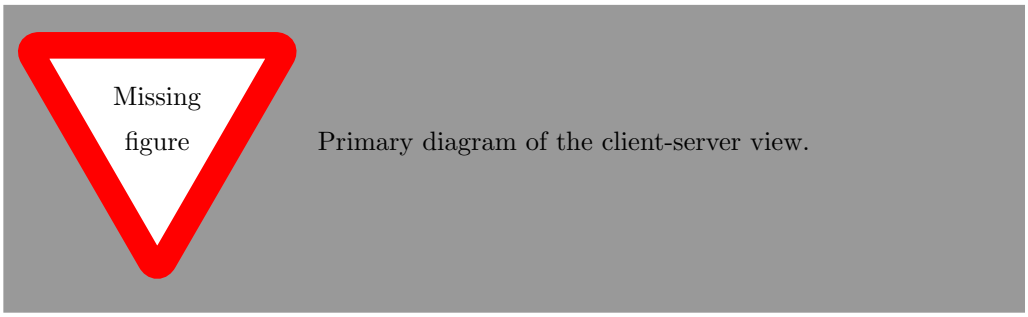


Figure 11: Primary diagram of the client-server view.

4.1 Main architectural decisions

Discuss your architectural decisions for the most important requirements in more detail using the components of the client-server view. Pay attention to the solutions that you employed and the alternatives that you considered. The explanation here must be self-contained and complete. Imagine you had to describe how the architecture supports the core functionality to someone that is looking at the client-server view only. Hide unnecessary details (these should be shown in the decomposition view).

4.1.1 ReqX: requirement name

Describe the design choices related to *ReqX* together with the rationale of why these choices where made.

Alternatives considered

Alternative(s) for choice 1 Explain what alternative(s) you considered for this design choice and why they where not selected.

5 Decomposition view (UML Component diagram)

Discuss the decompositions of the components of the client-server view which you have further decomposed.

5.1 ComponentX

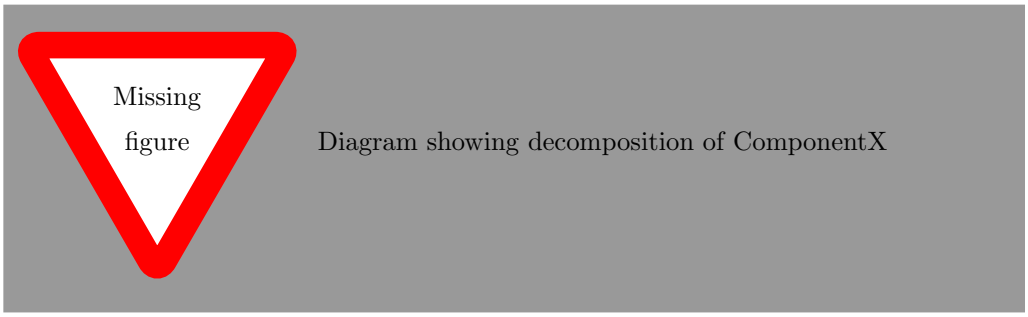
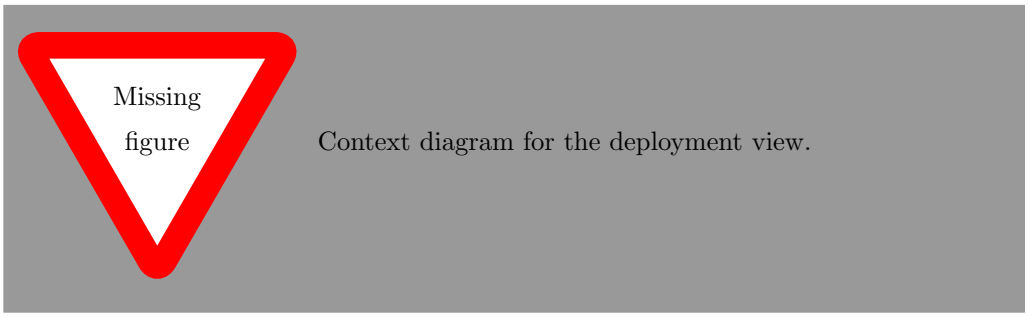


Figure 12: Decomposition of `ComponentX`

Describe the decomposition of `ComponentX` and how this relates to the requirements.

6 Deployment view (UML Deployment diagram)

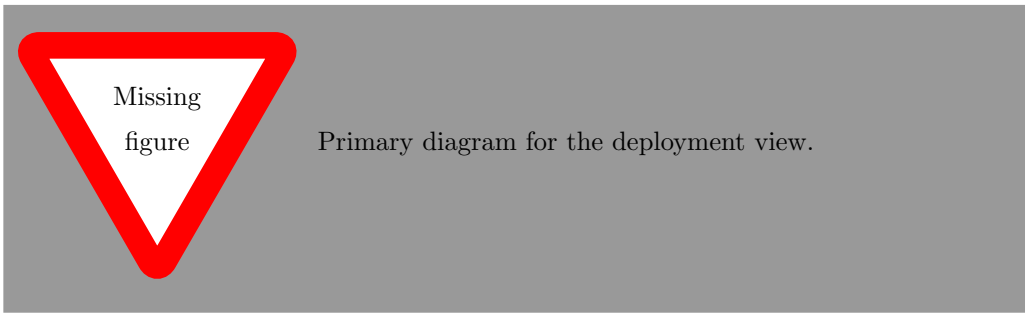
Describe the context diagram for the deployment view. For example, which protocols are used for communication with external systems and why?



Context diagram for the deployment view.

Figure 13: Context diagram for the deployment view.

The primary deployment diagram itself and accompanying explanation. Pay attention to the parts of the deployment diagram which are crucial for achieving certain non-functional requirements. Also discuss any alternative deployments that you considered.



Primary diagram for the deployment view.

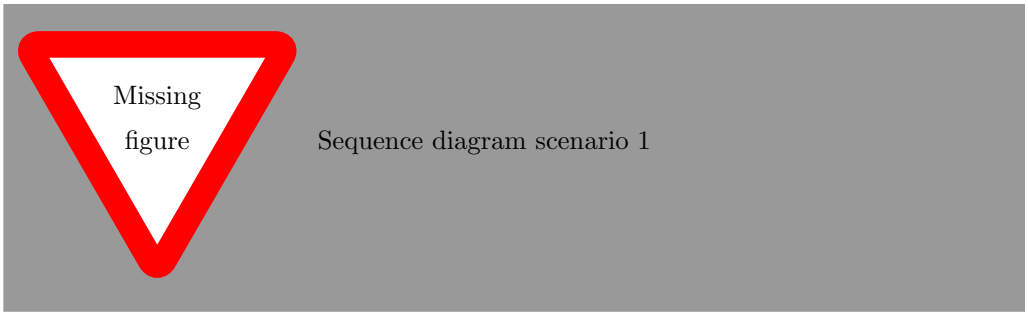
Figure 14: Primary diagram for the deployment view.

7 Scenarios

Illustrate how your architecture fulfills the most important data flows. As a rule of thumb, focus on the scenario of the domain description. Describe the scenario in terms of architectural components using UML Sequence diagrams and further explain the most important interactions in text. Illustrating the scenarios serves as a quick validation of the completeness of your architecture. If you notice at this point that for some reason, certain functionality or qualities are not addressed sufficiently in your architecture, it suffices to document this, together with a rationale of why this is the case according to you. You do not have to further refine you architecture at this point.

7.1 Scenario 1

Shortly describe the scenario shown in this subsection. Show the complete scenario using one or more sequence diagrams.



Sequence diagram scenario 1

Figure 15: The system behavior for the first scenario.

A Element catalog

List all components and describe their responsibilities and provided interfaces. Per interface, list all methods using a Java-like syntax and describe their effect and exceptions if any. List all elements and interfaces alphabetically for ease of navigation.

A.1 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.2 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
  - `returntType1 operation1(ParamType param) throws SomeException`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void operation2(ParamType2 param)`
      - Effect: Describe the effect of the operation
      - 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.3 Generator

- **Description:** A `Generator` generates the documents and forwards them to `OtherFunctionality` to store and deliver them. Its availability is monitored by the `DocumenGenerationManager` with the Ping interface. Document processing jobs are assigned to
- **Super-component:** None
- **Sub-components:** None

Provided interfaces

- AssignJobs
  - `void assignJobs(CompletePartialBatchData batchData)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void operation2(ParamType2 param)`
      - Effect: Describe the effect of the operation
      - 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.4 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.5 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)`
    - 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 `OtherFunctionality` for the `Key` and stores it in its cache, after which it returns that `Key`.
    - Exceptions: None

A.6 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: Describe the effect of the operation
    - Exceptions: None

A.7 PDSDBReplica

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

Provided interfaces

- ExtendedDocumentMgmt
  - `List<Document> getAllDocumentsOf(TimeStamp whenFailed)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `List<Document> getDocumentsSince(TimeStamp whenFailed)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
  - `void storeDocuments(List<Tuple<DocumentId, Document, DocumentMetaData>> documentList)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
  - `void storeDocument(DocumentId documentId, Document document, DocumentMetaDate documentMetaDat`
    - Effect: The `PDSDBReplica` stores the given document with its textttDocumentId and meta-data.
    - Exceptions: None
  - `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: Describe the effect of the operation
    - 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.8 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 `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: Describe the effect of the operation
    - Exceptions: None

A.9 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 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: Describe the effect of the operation
    - Exceptions: None

A.10 OtherFunctionality

- **Description:** Responsibilities of the component.
- **Super-component:** None
- **Sub-components:** the direct sub-components, if any.

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 `OtherFunctionality` 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
    - `void generationError(JobId jobid, Error error)`
      - Effect: Describe the effect of the operation
      - Exceptions: None
- GetBatchData
  - `Tuple<JobId, RawData> getRawData(List<JobId> jobIds)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
  - `BatchMetaData getMetaData(BatchId batchId)`
    - Effect: Describe the effect of the operation
    - Exceptions: None
- GetKey
  - `Key getKey(CustomerId customerId)`
    - Effect: The `OtherFunctionality` returns the key belonging to the Customer Organization identified by `customerId`.
    - Exceptions: None
- GetTemplate
  - `Template getTemplate(CustomerId customerId, DocumentType documentType, TimeStamp whenReceived`
    - Effect: The `OtherFunctionality` returns the `Template` belonging to the customer organisation with id `customerId` corresponding to a document of type `documentType` and received at time `whenReceived`.
    - Exceptions: None
- SetStatus
  - `void setJobStatusAsTemporarilyFailed(List<JobId> statusesOfJobs)`
    - Effect: The `OtherFunctionality` stores the `JobStatus` 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
- NotifyOperator
  - `void notifyOperatorOfPDSDBReplicaFailure(PDSDBReplicaId replicaId, TimeStamp dateTime)`
    - Effect: The `OtherFunctionality` 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 notifyOperatorOfDocumentGenerationFailure(NotificationMessage msg, TimeStamp whenFailed)`

* Effect: The `OtherFunctionality` 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

A.11 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 `OtherFunctionality` 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, List<JobId> jobIds )`
    * Effect: The `Scheduler` adds the jobs identified by their `JobId` 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.12 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 `OtherFunctionality` for the `Template` and stores it in its cache, after which it returns that `Template`.
    - Exceptions: None

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 possibilites are a long integer, a string, a URL etc.
- `BatchMetaData`: A data structure listing the metadata 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, ...
- `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.
- `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 possibilites are a long integer, a string, a URL etc.
- `Document`: A data file corresponding to a document. The architecture specifies the format of this data type as a PDF-file. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `DocumentId`: A piece of data uniquely identifying a document in the system.
- `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 etc.
- `Echo`: The response to a ping message. This data element does not contain any meaningful data.
- `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 possibilites 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 possibilites are a long integer, a string, a URL etc.
- `NotificationMessage`: A textual message which can be used to include extra information about the event of the notification.
- `PDSDBReplicaId`: A piece of data uniquely identifying a `PDSDBReplica` in the system. This architecture does not specify the exact format of this identifier, but possibilites are a long integer, a string, a URL etc.
- `RawData`: A data structure listing the raw data used in a document processing job.
- `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 possibilites are a long integer, a string, a URL etc.
- `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.