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 **3**

2 Overview **3**
2.1 Architectural decisions . . . 3
2.2 Discussion . . . 3

3 Attribute-driven design documentation **3**
3.1 Decomposition 1: eDocs (X1, Y3, UCa, UCb, UCc) . . . 3
3.1.1 Module to decompose . . . 3
3.1.2 Selected architectural drivers . . . 3
3.1.3 Architectural design . . . 3
3.1.4 Instantiation and allocation of functionality . . . 3
3.1.5 Interfaces for child modules . . . 4
3.1.6 Data type definitions . . . 5
3.1.7 Verify and refine . . . 5
3.2 Decomposition 2: OtherFunctionality(P2, UC12, UC13, UC14, UC15) . . . 5
3.2.1 Module to decompose . . . 5
3.2.2 Selected architectural drivers . . . 5
3.2.3 Architectural design . . . 5
3.2.4 Instantiation and allocation of functionality . . . 7
3.2.5 Interfaces for child modules . . . 7
3.2.6 Data type definitions . . . 7
3.2.7 Verify and refine . . . 9
3.3 Decomposition 3: ModuleA (X1, Y3, UCa, UCb, UCc) . . . 9
3.3.1 Module to decompose . . . 9
3.3.2 Selected architectural drivers . . . 9
3.3.3 Architectural design . . . 10
3.3.4 Instantiation and allocation of functionality . . . 10
3.3.5 Interfaces for child modules . . . 11
3.3.6 Data type definitions . . . 11
3.3.7 Verify and refine . . . 11

4 Client-server view (UML Component diagram) **11**
4.1 Main architectural decisions . . . 12
4.1.1 ReqX: requirement name . . . 12

5 Decomposition view (UML Component diagram) **12**
5.1 ComponentX . . . 12

6 Deployment view (UML Deployment diagram) **12**

7 Scenarios **14**
7.1 Scenario 1 . . . 14

A Element catalog **14**
A.1 Completer . . . 14
A.2 DocumentGenerationManager . . . 15
A.3 Generator . . . 15
A.4 GeneratorManager . . . 16
A.5 KeyCache . . . 16
A.6 PDSDB . . . 16
A.7 PDSDBReplica . . . 17
A.8 PDSLongTermDocumentManager . . . 17
A.9 PDSReplicationManager . . . 17
A.10 OtherFunctionality . . . 18
A.11 Scheduler . . . 19
A.12 TemplateCache . . . 19

B Defined data types **19**

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.

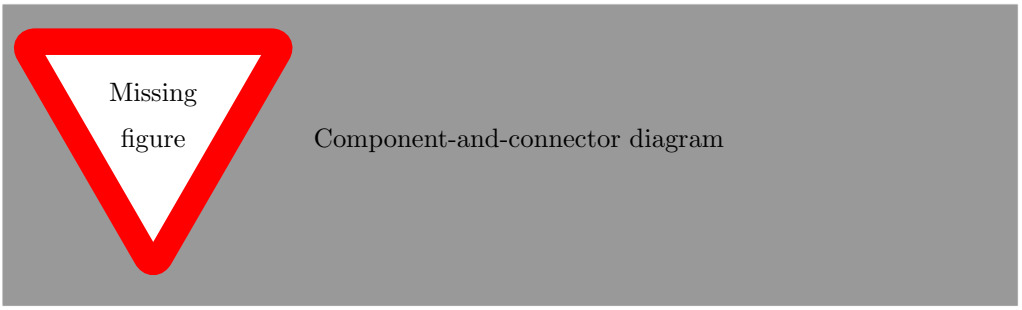

Figure 1: Component-and-connector diagram of this decomposition.

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.

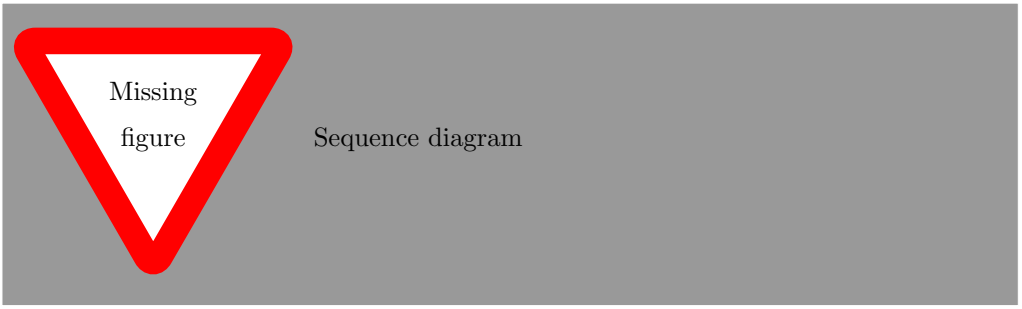

Figure 2: Sequence diagram illustrating a key behavioural aspect.

Deployment Rationale of the allocation of components to physical nodes.

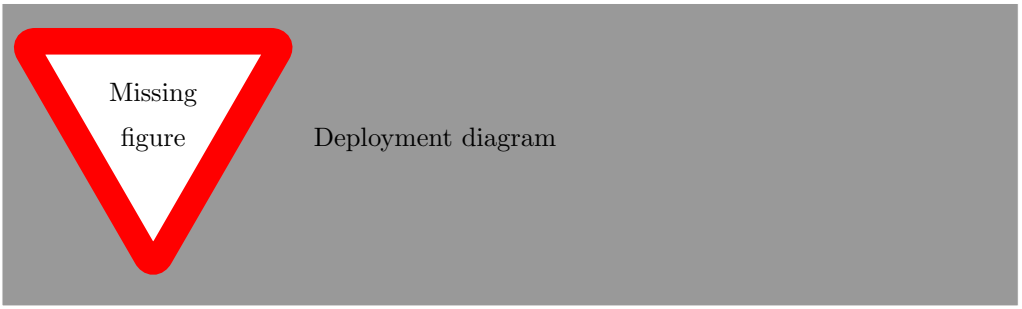

Figure 3: Deployment diagram of this decomposition.

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, 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

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 was chosen because it has one of the highest priorities among all remaining non-functional drivers and the domain on which its focus lies complements 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.

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.

UserFacade We introduce this component to be able to distinguish between registered users and unregistered users 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. 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 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.

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.

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. 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`.

Alternatives considered

Alternatives for solution Een nadeel aan deze methode is wel dat wanneer de PDSDB langer dan drie uur offline is, er documenten in de DocumentDB opgeslagen zitten die niet in de PDSDB gestoken worden wanneer die terug online komt. Deze documenten zijn niet meet opvraagbaar -¿ out of scope Het alternatief is dat wanneer de PDSDB terug online komt, de DocumentStorageManager vergelijkt welke documenten er voor de Registered Recipients in de PDSDB en de DocumentDB opgeslagen zijn. Het voordeel hierbij is dat alle documenten opvraagbaar blijven wanneer de PDSDB terug online komt. Het nadeel is dat dit meer werk inhoudt voor de DocumentStorageManager. Aangezien enkel drie uur vereist zijn, gaan we voor het eerste alternatief.

Merk op: als alternatief hadden we gedacht aan GEEN actieve heartbeat vanuit de PDSDB bij het online komen, maar bij een schrijfpoging naar de PDSDB als impliciete ping wanneer er een document opgeslagen wordt. Het probleem hierbij is dat wanneer er al even geen documenten genereerd zijn maar er toch noch documentreferenties in de cache zitten en op dat moment de PDSDB online komt, een registered recipient die documenten niet kan opvragen.

3.2.4 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.

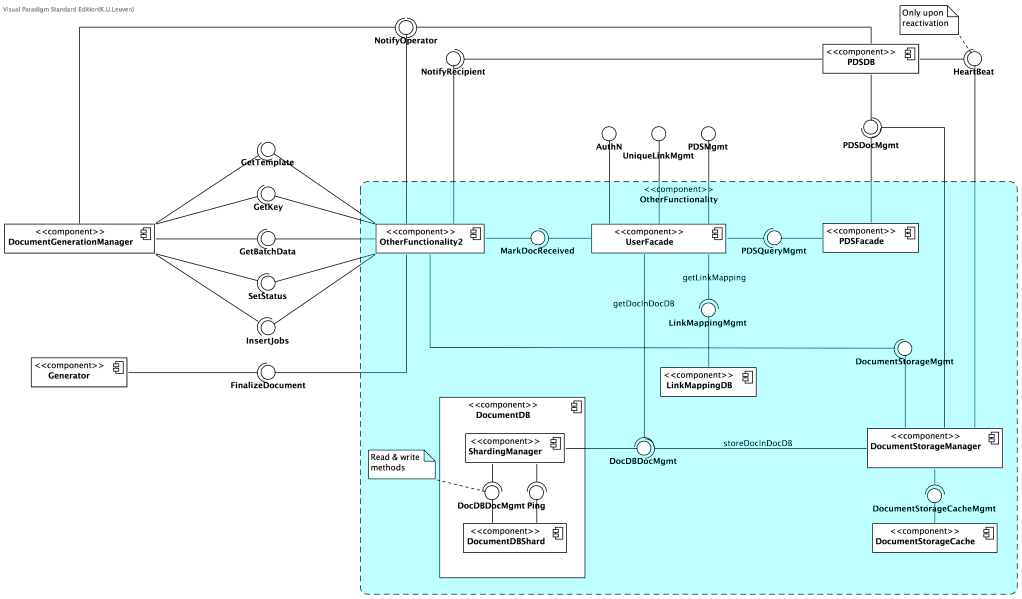

Figure 4: Component-and-connector diagram of the second decomposition.

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

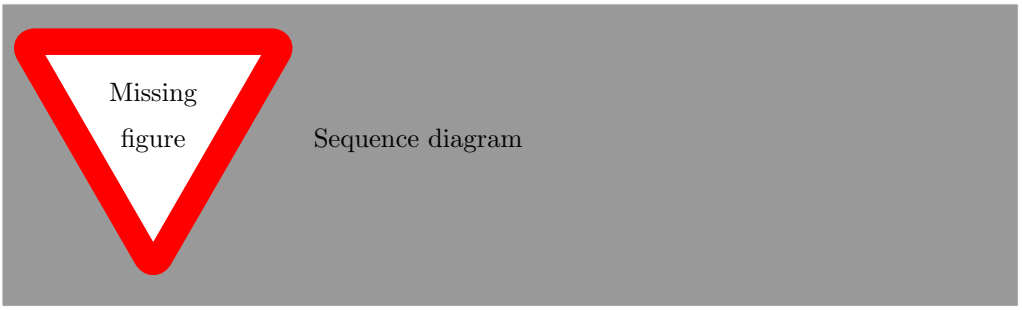

Figure 5: Sequence diagram illustrating a key behavioural aspect.

Deployment Rationale of the allocation of components to physical nodes.

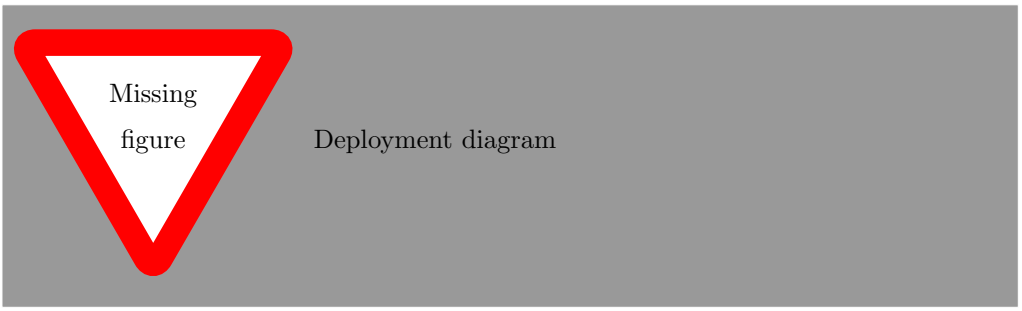

Figure 6: Deployment diagram of this decomposition.

3.2.5 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

3.2.6 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: ModuleA (X1, Y3, UCa, UCb, UCc)

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.

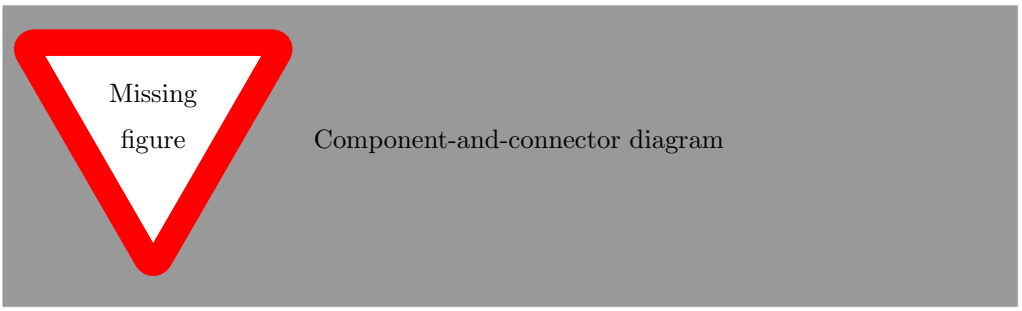

Figure 7: Component-and-connector diagram of this decomposition.

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

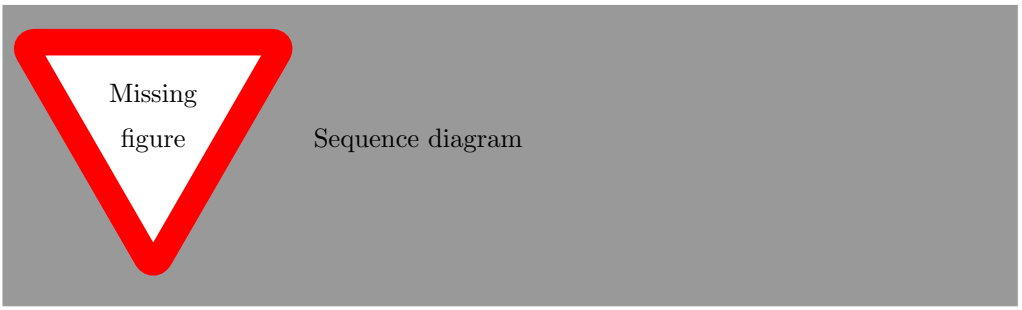

Figure 8: Sequence diagram illustrating a key behavioural aspect.

Deployment Rationale of the allocation of components to physical nodes.

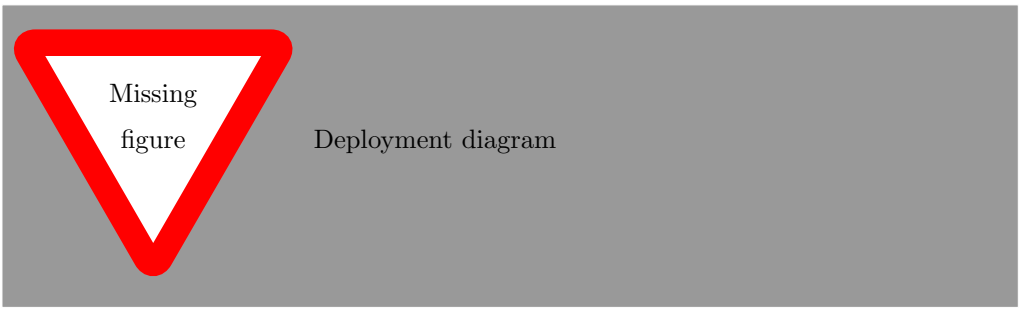

Figure 9: Deployment diagram of this decomposition.

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.

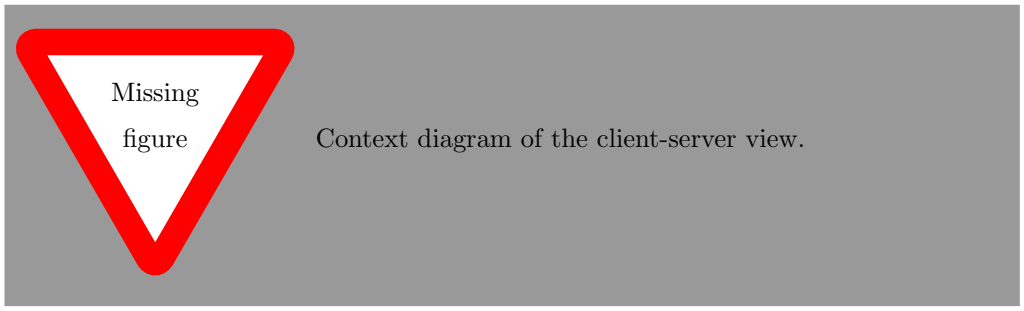

Figure 10: Context diagram for the client-server view.

The primary diagram and accompanying explanation.

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).

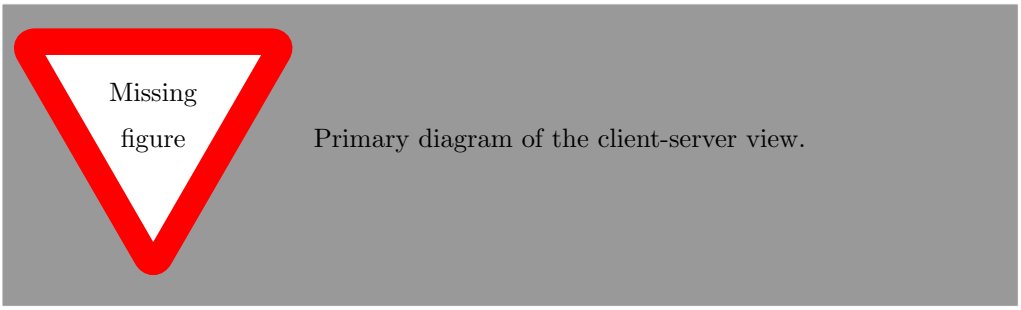

Figure 11: Primary diagram of the client-server view.

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

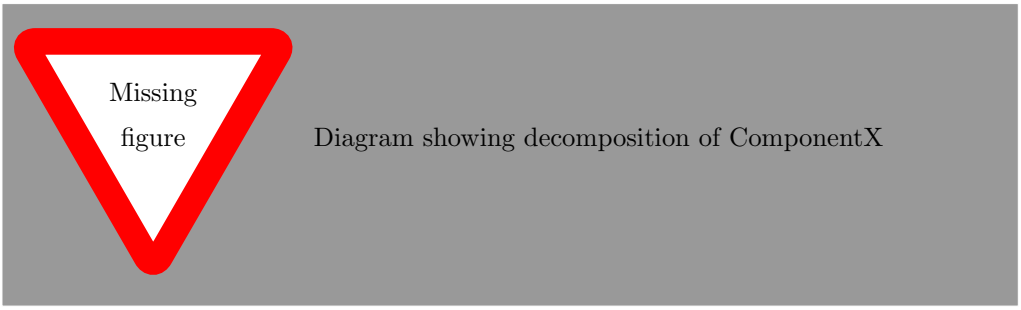

Figure 12: Decomposition of `ComponentX`

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?

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.

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

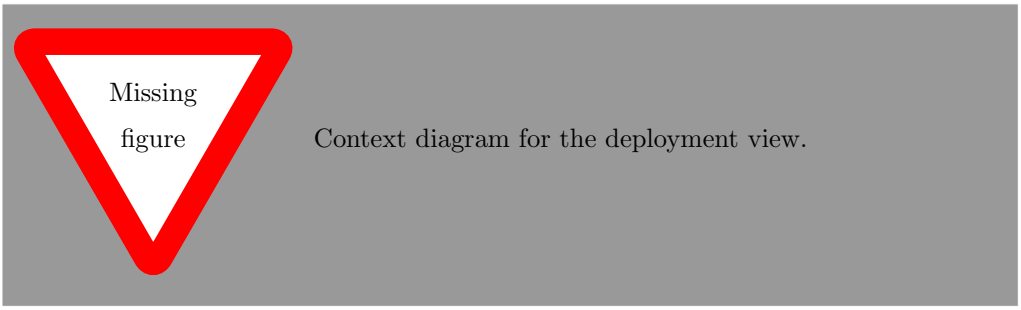

Figure 13: Context diagram for the deployment view.

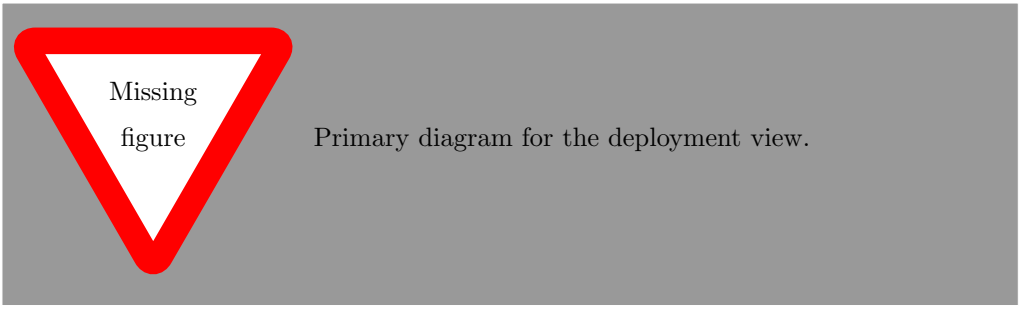

Figure 14: Primary diagram for the deployment view.

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.

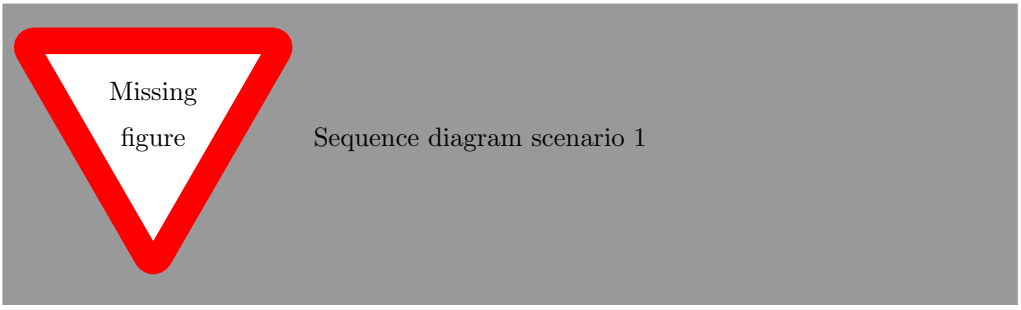

Figure 15: The system behavior for the first scenario.

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:** Responsibilities of the component.

- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- Complete
  - `CompletePartialBatchData getComplete(BatchId batchId, List<JobId> jobIds )`
    - 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

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.
- **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: Describe the effect of the operation
    - Exceptions: None

A.3 Generator

- **Description:** A `Generator` generates the documents and forwards them to `OtherFunctionality` to store and deliver them.
- **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
  - `returntType2 operation3()`
    - Effect: Describe the effect of the operation
    - 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:** Responsibilities of the component.
- **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:** Responsibilities of the component.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetKey
  - `Key getKey(CustomerId customerId)`
    - Effect: Describe the effect of the operation
    - Exceptions: None

A.6 PDSDB

- **Description:** Responsibilities of the component.
- **Super-component:** None
- **Sub-components:** `PDSDBReplica`, `PDSLongTermDocumentManager`, `PDSReplicationManager`

Provided interfaces

- 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

A.7 PDSDBReplica

- **Description:** Responsibilities of the component.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `List<Document> getDocumentsSince(TimeStamp whenFailed)`
    - Effect: Describe the effect of the operation
    - Exceptions:
      - SomeException: Describe when the exception is thrown.
    - `void storeDocuments(ParamType2 param)`
      - 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:** Responsibilities of the component.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- DocumentMgmt
  - `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

A.9 PDSReplicationManager

- **Description:** Responsibilities of the component.
- **Super-component:** `PDSDB`
- **Sub-components:** None

Provided interfaces

- ExtendedDocumentMgmt
  - `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

A.10 OtherFunctionality

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

Provided interfaces

- FinalizeDocument
  - `void storeAndDeliverDocument(JobId jobid, Document doc) throws SomeException`
    - Effect: The `OtherFunctionality` will store the given document `document` and deliver it.
    - 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: Describe the effect of the operation
    * Exceptions: None

- GetTemplate
  - `Template getTemplate(TemplateId templateId)`
    * Effect: Describe the effect of the operation
    * Exceptions: None

- SetStatus
  - `returntType2 operation3()`
    * Effect: Describe the effect of the operation
    * Exceptions: None

- NotifyOperator
  - `void notifyOperatorOfPDSDBReplicaFailure(PDSDBReplicaId replicaId, TimeStamp dateTime)`
    * Effect: Describe the effect of the operation
    * Exceptions: None

A.11 Scheduler

- **Description:** Responsibilities of the component.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetNextJobs
  - `Tuple<BatchId, List<JobId>> getNextJobs()`
    * Effect: Describe the effect of the operation
    * Exceptions:
      · SomeException: Describe when the exception is thrown.
    * `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
  - `returntType2 operation3()`
    * Effect: Describe the effect of the operation
    * Exceptions: None

- GetStatistics
  - `returntType2 operation3()`
    * Effect: Describe the effect of the operation
    * Exceptions: None

A.12 TemplateCache

- **Description:** Responsibilities of the component.
- **Super-component:** `DocumentGenerationManager`
- **Sub-components:** None

Provided interfaces

- GetTemplate
  - `Template getTemplate(TemplateId templateId)`
    - Effect: Describe the effect of the operation
    - 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.
- `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.
- `Document`: Description of data type.
- `DocumentId`:
- `DocumentType`: A piece of data describing the type of a document. This architecture does not specify the exact format of this identifier, 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`:
- `PDSDBReplicaId`:
- `RawData`: A data structure listing the raw data used in a document processing job.
- `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.