**Projekt: Mosti**

Software Architektur Spezifikation

(Software Architecture Document)

[Dokumentstruktur basiert auf RUP „Software Architecture Document“]

# Dokumentinformationen

## Änderungsgeschichte

|  |  |  |  |
| --- | --- | --- | --- |
| *Datum* | *Version* | *Änderung* | *Autor* |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Inhalt

[1 Dokumentinformationen 2](#_Toc448394277)

[1.1 Änderungsgeschichte 2](#_Toc448394278)

[1.2 Inhalt 3](#_Toc448394279)

[2 Einführung (Introduction) 4](#_Toc448394280)

[2.1 Definitionen und Abkürzungen (Definitions, Acronyms, Abbreviations) 4](#_Toc448394281)

[2.2 Referenzen (References) 4](#_Toc448394282)

[2.3 Übersicht (Overview) 4](#_Toc448394283)

[3 Architektonische Darstellung (Architectural Representation) 5](#_Toc448394284)

[4 Architektonische Ziele & Einschränkungen (Architectural Goals and Constraints) 5](#_Toc448394285)

[5 Logische Architektur (Logical View) 6](#_Toc448394286)

[5.1 Übersicht (Overview) 7](#_Toc448394287)

[5.2 Design Pakete (Architecturally Significant Design Packages) 8](#_Toc448394288)

[5.2.1 Package ABC *(z.B. Gui)* 8](#_Toc448394289)

[5.2.1.1 Beschreibung des Package 8](#_Toc448394290)

[5.2.1.2 Diagramme 8](#_Toc448394291)

[5.2.1.3 Schnittstellen 8](#_Toc448394292)

[5.2.1.4 Operationen 8](#_Toc448394293)

[5.2.1.4.1 Interne Operation 1 8](#_Toc448394294)

[5.2.1.4.2 Interne Operation 2 8](#_Toc448394295)

[5.2.2 Package DEF *(z.B. Problem Domain)* 9](#_Toc448394296)

[5.2.2.1 Beschreibung des Package 9](#_Toc448394297)

[5.2.2.2 Diagramme 9](#_Toc448394298)

[5.2.2.3 Schnittstellen 9](#_Toc448394299)

[5.2.2.4 Operationen 9](#_Toc448394300)

[5.2.2.4.1 Interne Operation 1 9](#_Toc448394301)

[5.2.2.4.2 Interne Operation 2 9](#_Toc448394302)

[5.2.3 Package GHI *(z.B. Datenhaltung)* 10](#_Toc448394303)

[5.2.3.1 Beschreibung des Package 10](#_Toc448394304)

[5.2.3.2 Diagramme 10](#_Toc448394305)

[5.2.3.3 Schnittstellen 10](#_Toc448394306)

[5.2.3.4 Operationen 10](#_Toc448394307)

[5.2.3.4.1 Interne Operation 1 10](#_Toc448394308)

[5.2.3.4.2 Interne Operation 2 10](#_Toc448394309)

[6 Physikalische Sicht (Physical View) 11](#_Toc448394310)

[7 Prozesse und Threads (Process View) 11](#_Toc448394311)

[8 Datenspeicherung (Data View) 11](#_Toc448394312)

[9 Größen und Leistung (Size and Performance) 11](#_Toc448394313)

# Einführung (Introduction)

## Definitionen und Abkürzungen (Definitions, Acronyms, Abbreviations)

Vgl. separates Glossary-Dokument im Repository, Dateiname: Glossary.docx

## Referenzen (References)

Buch: Craig Larman: UML 2 und Patterns angewendet (2005)

## Übersicht (Overview)

Im folgenden Teil dieses Dokuments gehen wir auf die architektonische Darstellung bei unserer Software ein und beschreiben, welche Ziele und Einschränkungen damit verbunden sind. Außerdem möchten wir insbesondere auf die logische Architektur (logical view) und die Datenspeicherung (data view) eingehen, da diese Views bei unserer Software mehr zum Tragen kommen als andere. Abschließend wird noch beschrieben, welche Größen und Leistungen Mosti umfassen soll.

# Architektonische Darstellung (Architectural Representation)

Ein System lässt sich durch verschiedene architektonische Sichten darstellen. So gibt es beispielsweise Sichten für Logik, Daten, Sicherheit, Implementierung oder auch für Use Cases.

In diesem Dokument möchten wir die Architektur der Mosti-Software mit folgenden Sichten beschreiben:

Logische Sicht (logical view):

Diese Sicht soll die Struktur der Software widerspiegeln. Dies betrifft hier vor allem die Implementierung. So wird beispielsweise beschrieben, wie Packages und Klassen gegliedert bzw. organisiert werden oder auch wie Klassen anhand bestimmter Funktionen interagieren können. Wir werden bei unserer Software das Drei-Schichten-Modell an, welches später genauer erläutert wird.

Daten-Sicht (data view):

Diese Sicht soll einen Überblick darüber geben, wie wir Daten erhalten und verwalten. Bei Mosti setzen wir das mithilfe einer Datenbank um. Die Software hat dabei jederzeit Zugriff auf die Daten, die in der Datenbank gespeichert sind.

*Anmerkung:*

Eine physikalische Sicht ist bei Mosti derzeit nicht relevant, weil die Software so geplant ist, dass sie von einem Anwender nur auf einem bestimmten Rechner benutzt wird und nicht mit anderen Rechnern oder Systemen kommuniziert.

*<Verbale Beschreibung wichtiger Architekturentscheidungen. Ggf. auch andere Architekturideen kurz beschreiben, die diskutiert und (warum?) verworfen wurden.>*

# Architektonische Ziele & Einschränkungen (Architectural Goals and Constraints)

Wir haben uns für eine Drei-Schichten-Architektur entschieden, welche sich aus den Schichten Präsentation, Logik und Datenerhaltung zusammensetzt. Dabei sollen zwischen den Schichten nur Abhängigkeiten von oben nach unten bestehen. Durch diese Reduktion von Abhängigkeiten können Änderungen in einzelnen Schichten vorgenommen werden, ohne das gesamte System ändern zu müssen. Da wir einen iterativen Software-Entwicklungs-Prozess durchlaufen, können nahezu jederzeit Änderungen in der Implementierung gefordert sein, da sich z. B. nochmals die Anforderungen geändert haben. Es kann auch beispielsweise eine andere Präsentation der Daten gewünscht sein – dann sind nur Änderungen in der sogenannten Präsentations- bzw. GUI-Schicht erforderlich. In den anderen beiden Schichten müssen dabei keinerlei Änderungen vorgenommen werden, da diese sich unterhalb der GUI-Schicht befinden und somit nicht von dieser abhängig sind. Ein weiterer Grund, weshalb wir uns für diese Architektur entschieden haben, ist, dass wir diese bereits in der Vorlesung behandelt haben und diese im Vergleich zu anderen vorgestellten Architekturen am passendsten finden. Auch deswegen, weil wir u. a. Teilfunktionalitäten mit spezifischen GUIs haben, an welchen bestimmte Teammitglieder arbeiten. Durch das Schichtenmodell ist eine bestimmte Struktur bzgl. der Implementierung vorgegeben, wodurch andere Teammitglieder einen besseren Überblick und schnelleres Verständnis für Bereiche, die sie nicht selbst implementiert haben, erlangen sollen.

*<Begründung der Architekturentscheidungen: Welche Anforderungen / Einschränkungen haben zu dieser Architektur geführt, d.h.: warum so und nicht anders, welche Vorteile ergeben sich aus dieser Architektur?>*

# Logische Architektur (Logical View)

Bei der Entwicklung von Mosti verfolgen wir das Drei-Schichten-Modell. Dabei unterscheiden wir folgende Schichten:

* GUI-Schicht
* Domain-Logik-Schicht
* Persistenz-Schicht (oder auch Datenerhaltungsschicht)

Dieses Modell erläutern wir im folgenden Punkt (5.1 Überblick) noch genauer.

Bezüglich der Package-Struktur streben wir generell an, ähnliche Klassen in einem Package zu gliedern (hohe Kohäsion) und andererseits, dass wenige Abhängigkeiten zwischen den Packages bestehen (geringe Kopplung).

*<Beschreibung der logischen Struktur des Projekts>*

## Übersicht (Overview)

Das Drei-Schichten-Modell für unsere Software lässt sich wie folgt darstellen:

**GUI**

**Domain-Logik**

**Persistenz**

Schichten-Architektur

Die Schicht GUI umfasst die grafische Benutzeroberfläche, also wie die Daten dem Benutzer präsentiert werden.

Die Domain-Logik-Schicht ist in dieser Darstellung die Schnittstelle zwischen der GUI-Schicht und der Persistenz-Schicht. Darin werden die spezifischen Anforderungen bzw. Funktionalitäten umgesetzt.

In der Persistenz-Schicht wird die Datenerhaltung geregelt.

Aus dem Diagramm ist zu entnehmen, dass die Abhängigkeit zwischen den Schichten von oben nach unten verläuft. So kann die GUI-Schicht nur auf die Domain-Logik-Schicht zugreifen und die Domain-Logik-Schicht wiederum nur auf die Persistenz-Schicht. Somit kann die GUI-Schicht zum einen nicht auf die Persistenz-Schicht zugreifen. Zum anderen können Domain-Logik-Schicht und Persistenz-Schicht nicht auf die GUI-Schicht zugreifen und machen somit eine unabhängige Entwicklung möglich.

Im Folgenden werden die einzelnen Schichten noch spezifischer dargelegt.

## Design Pakete (Architecturally Significant Design Packages)

In diesem Überpunkt gehen wir nochmals auf die drei einzelnen Schichten ein, die wir hier als Design-Pakete bezeichnen. Hierbei ist zu erwähnen, dass es sich dabei nicht um die von uns erstellten Java-Packages handelt. Da es bei Mosti verschiedene Teilbereiche gibt (z. B. Lagerverwaltung oder Kundenverwaltung), wurden diese jeweils zusammenhängenden Klassen in einem Java-Package zusammengefasst und sind innerhalb dieses Teilbereiches/Java-Packages in die Schichten unterteilt. Diese spezifischen Unterteilungen führen wir im Punkt 5.3 auf.

### Package GUI

#### Beschreibung des Package

Das Package GUI umfasst die graphische Benutzeroberfläche und somit u. a. die Präsentation von Daten, die über die Domain-Logik-Schicht zur Verfügung stehen. Bei Mosti arbeitet der Benutzer selbst ausschließlich über die GUI-Schicht, um Daten zu erfassen, abzurufen oder um bestimmte Funktionen auszuführen. Somit ist es auch wichtig, dass in diesem Package auch Benutzereingaben erkannt und weitergeleitet werden.

#### Diagramme

<Klassendiagramm>

#### Schnittstellen

Die GUI-Schicht hat Zugang zur Domain-Logik-Schicht. Jedoch kann von anderen Schichten nicht auf die GUI-Schicht zugegriffen werden.

#### Operationen

##### Interne Operation 1

<Beschreibung der Operation mit Sequenzdiagramm>

##### Interne Operation 2

<Beschreibung der Operation mit Sequenzdiagramm>

### Package Domain-Logik

#### Beschreibung des Package

Das Package Domain-Logik umfasst die Logik, welche hinter den Funktionalitäten der Software liegt. Somit ist darin geregelt, wie bestimmte Daten verarbeitet werden. Bezogen auf Mosti ist beispielsweise die Funktionalität, welche die Kassenfunktion betrifft, von hoher Bedeutung.

#### Diagramme

<Klassendiagramm>

#### Schnittstellen

Die Domain-Logik-Schicht hat Zugang auf die darunterliegende Persistenz-Schicht. Auf die Domain-Logik-Schicht zugreifen kann hingegen nur die GUI-Schicht.

#### Operationen

##### Interne Operation 1

<Beschreibung der Operation mit Sequenzdiagramm>

##### Interne Operation 2

<Beschreibung der Operation mit Sequenzdiagramm>

### Package Persistenz

#### Beschreibung des Package

Im Package Persistenz wird die Datenerhaltung geregelt. Da wir zur Speicherung von Daten eine Datenbank benutzen, soll dieses Package ermöglichen, dass eine Verbindung zur Datenbank aufgebaut wird, man auf die Daten zugreifen und Datenänderungen tätigen kann.

#### Diagramme

<Klassendiagramm>

#### Schnittstellen

Die Schnittstelle zwischen unserer Software und der Datenbank wird innerhalb der Persistenz-Schicht gewährleistet. Zugang auf die Persistenz-Schicht hat die Domain-Logik-Schicht.

#### Operationen

##### Interne Operation 1

<Beschreibung der Operation mit Sequenzdiagramm>

##### Interne Operation 2

<Beschreibung der Operation mit Sequenzdiagramm>

## Java-Pakete und deren Architektur-Design

### Package Account

#### Beschreibung des Packages

Das Package *account* ermöglicht den Zugang zu den Anderen Packages. Ohne einen Account kann in unserer Mosti Software keiner etwas machen.

#### Diagramme

**TO DO**

#### Schnittstellen

Das Package *account* besitzt Schnittstellen zum Package *terminplanung, kassenfunktion, kundenverwaltung, lagerverwaltung, mitarbeiterverwaltung, verkaufsverwaltung, dienstleistungsverwaltung* und zur *main.*

#### Aufteilung der Klassen in das Drei-Schichten-Modell:

GUI-Schicht: M\_Startseite, Anmeldung,

Logik-Schicht: M\_Account

Persistenz-Schicht: Accountverwaltung

#### Operationen

Accountverwaltung:

* package boolean mitarbeiterSuchen(String,String)

Anmeldung:

* Konstruktor: public Anmeldung()
* Innere Klasse: ActionListener
  + public void actionPerformed(ActionEvent)

M\_Account:

* Konstruktor: public M\_Account(String,String)
* package void anmelden(string)
* package string getBenutzername()
* package void setBenutzername(String)

M\_Startseite:

* Konstruktor: public M\_Startseite(Angebote, Kundeneinkäufe)
* Innere Klasse: ActionListener
  + public void actionPerformed(ActionEvent)
* Innere Klasse: ActionListener
  + public void actionPerformed(ActionEvent)
* Innere Klasse: ActionListener
  + Public void actionPerformed(ActionEvent)
* Innere Klasse: ActionListener
  + Public void actionPerformed(ActionEvent)
* Innere Klasse: ActionListener
  + Public void actionPerformed(ActionEvent)
* Innere Klasse: ActionListener
  + Public void actionPerformed(ActionEvent)
* Innere Klasse: ActionListener
  + Public void actionPerformed(ActionEvent)

### Package terminplanung

#### Beschreibung des Packages

Das Package *terminplanung* umfasst alle Aspekte der Planung und Speicherung von Terminen. Es enthält sowohl Klassen zur Präsentation von Terminen und terminrelevanten Frames, Klassen zum Management des Datenbankzugriffes sowie Klassen zur internen Logik und Verwaltung der Termine.

#### Diagramme

**TO DO**

#### Schnittstellen

Das Package *terminplanung* besitzt Schnittstellen zum Package *kassenfunktion*, von welcher aus die Kundentermine mit den tatsächlichen Termindurchführung verknüpft werden. Eine weitere Schnittstelle existiert zum Package *account*, welches die Startseite der Software beinhaltet und den Benutzer zwischen den einzelnen Modulen navigiert.

#### Aufteilung der Klassen in das Drei-Schichten-Modell:

GUI-Schicht: TagFrame, TerminErstellenDialog, TerminHinzufügenFrame, TerminplanungsFrame

Logik-Schicht: FreieTermineTableModel, Intervall, Konfigurationswerte, KundenNamenCellRenderer, Termin, TermineCellRenderer, TermineTableModel,

Persisitenz-Schicht: TerminDB

#### Operationen

FreieTermineTableModel:

* Konstruktor: public FreieTermineTableModel(ArrayList<Termin>)
* public String getColumnName(int)
* public int getColumnCount()
* public int getRowCount()
* public Object getValueAt(int, int)
* private void berechneLücken()

Intervall:

* package int getEnde()
* package void setEnde(int)
* package int getStart()
* package void setStart()

Konfigurationswerte:

* Konstruktor: public Konfigurationswerte()
* package int getZeitslot()
* package void setZeitslot(int)
* package int getArbeitsbeginn()
* package void setArbeitsbeginn(int)
* package int getArbeitsende()
* package void setArbeitsende(int)
* package int getAufteilung()
* package void setAufteilung(int)
* package int getZeilenanzahlProSeite()
* package void setZeilenanzahlProSeite(int)

KundenNameCellRenderer:

* Konstruktor public KundenNameCellRenderer()
* public Component getTableCellRendererComponent(JTable, Object, boolean, boolean, int, int)

TagFrame:

* Konstruktor TagFrame(Date, int, TerminHinzufügenFrame)
* innere Klasse: MyMouseListener
* public void mousePressed(MouseEvent)
* innerer Klasse: ActionListener
* public void actionperformed(ActionEvent)

Termin:

* package int getKundenId()
* package void setKundenId(int)
* package int getTerminId()
* package void setTerminId(int)

TerminDB:

* Konstruktor: public TerminDB()
* private void init()
* package ArrayList<Termin> termineLaden(Date, int)
* package void termineSpeichern(ArrayList<Termin>, Date)
* package String kundenNamenLaden(int)
* package ArrayList<Integer> adminwerteLaden()
* package ArrayList<Termin> freieTermineSuchen(Date)
* package int kundenIdLaden(String)

TermineCellRenderer:

* Konstruktor: public TermineCellRenderer()
* public Component getTableCellRendererComponent(JTable, Object, boolean, boolean, int, int)
* private String termineNachUhrzeit(int)

TerminErstellenDialog:

* Konstruktor: TerminErstellenDialog(int. Date, ArrayList<Termin>, String)
* innere Klasse: MyOkListener
* public void actionPerformed(ActionEvent)
* innere Klasse: MyKeyListener
* public void keyPressed(KeyEvent)

TermineTableModel:

* Konstruktor: public TermineTableModel(ArrayList<Termin>)
* public String getColumnName(int)
* public int getColumnCount()
* public int getRowCount()
* public Object getValueAt(int,int)
* package ArrayList<Termin> getTermine(int, int)
* package ArrayList<Termin> getAlleTermine()

TerminHinzufügenFrame:

* Konstruktor: public TerminHinzufügenFrame()
* innere Klasse: MyBerechnenHandler
* public void actionPerformed(ActionEvent)
* innere Klasse: MyMouseListener
* public void MousePressed(MouseEvent)
* package int berechneTermindauer(String)
* private int berechneAnzeigeSeite(int)
* package int getTerminlänge()

TerminplanungsFrame:

* Konstruktor: public TerminplanungsFrame()
* innere Klasse: ActionListener
* public void actionPerformed(ActionEvent)

# Physikalische Sicht (Physical View)

**Haben wir sowas??**

*<Aufteilung in Tiers / Partitions, Zuordnung von Modulen oder Prozessen zu einzelnen Rechnern, wenn notwendig (z.B. bei Client-Server-Architektur.>*

# Prozesse und Threads (Process View)

**Haben wir sowas??**

*<Wenn mehrere Prozesse oder Threads eingesetzt werden, wird hier beschrieben, wie diese ablaufen, miteinander funktionieren, Daten austauschen, sich synchronisieren, etc...>*

# Datenspeicherung (Data View)

Bei Mosti werden die erfassten Daten in einer Datenbank gespeichert. Die Verbindung wird mithilfe eines JDBC-Treibers aufgebaut, der innerhalb der Open-Source Implementierung UCanAccess auf eine Microsoft Access-Datenbank zugreift.

Aufbau der Datenbank:

*kunden:*

Enthält die Informationen, die über den Kunden abgespeichert werden.

Tabellenspalten: ID, Vorname, Nachname, Straße, PLZ, Wohnort, Telefonnummer,

*termine:*

Enthält die jährlichen Termine und ihre Belegung mit KundenId

TabellenSpalten: Tag1, Tag2, Tag3, ………, Tag365, Tag366

*mitarbeiter:*

Enthält alle Informationen, die über den Mitarbeiter und seinen Account abgespeichert werden

Tabellenspalten: ID, Vorname, Nachname, Strasse, Hausnummer, PLZ, Stadt, Telefonnummer, Benutzername, Passwort

*adminwerte:*

Enthält Konfigurationswerte, die der Mosterei-Inhaber bei Bedarf ändern kann

Tabellenspalten:Id, AnzeigeAufteilung, Zeitslotlänge, Arbeitsbeginn, Arbeitsende,

*<Beschreibung mit Diagramm der Datenspeicherung [Data Model].*

# Größen und Leistung (Size and Performance)

**Wieviele Kunden und so**

*<Einschränkungen der Applikation bezüglich Speicher, Leistung, etc…. (zum Beispiel: Verwaltung unterstützt maximal 20'000 Einträge)>*

Inhaltverzeichnis anpassen #Maddy