
TECHNISCHE UNIVERSITÄT WIEN

VIENNA UNIVERSITY OF TECHNOLOGY

*Lightweight Leela[1] SOA Middleware*

# Communication Framework

**184.153 Design Methods for Distributed Systems**
**swa028**

*BORKO Michael, 0025876*
*GREIFENEDER Michael,* 0525042
*MOTLIK Florian,* 0425525

Version 1.0

1 http://www.infosys.tuwien.ac.at/staff/zdun/publications/leelaSOAMiddleware.pdf

# Table of Contents

Installation und Deployment.....3
Modellbeschreibung.....3
Modellgenerierung.....3
Templatedefinition.....6
Workflow.....6
Anpassung der Implementation.....7
Technologien.....7
Arbeitsaufteilung.....7
Borko.....7
Hamm.....8
Hummer.....8
Mezensky.....8
Annahmen und Probleme.....8
Ausgelassene Punkte.....9

# Installation und Deployment

Das mitgelieferte Paket kann sehr einfach eingebunden und getestet werden, jedoch sollten folgende Pakete installiert werden: **ant** und **ant-optional**. Dies kann zum Beispiel unter ubuntu sehr einfach mit apt-get durchgeführt werden:

```
apt-get install ant ant-optional
```

Anschließend ist es erforderlich mit **ant** die zusätzlichen Bibliotheken zu laden. Dies wird mit folgendem Befehl erreicht:

```
ant get-deps
```

Sollten teilnehmende Peers an externen Rechnern angesprochen werden, muss die Datei peers.csv entsprechend geändert werden. Sonst können die geforderten Targets mit folgendem Befehl gestartet werden:

```
ant clean && ant test && ant run
```

# Modellbeschreibung

Das zu verwendende Modell basiert auf dem Broker-Muster aus dem Buch Software Architektur[2]. *„Es soll vermieden werden, dass Aspekte verteilter Programmierung über den Code einer verteilten Applikation verstreut sind ... Durch die Verlagerung aller Kommunikationsaufgaben in einen Broker werden die Kommunikationsaufgaben eines verteilten Systems von dessen Applikationslogik getrennt. Der Broker verbirgt und steuert die Kommunikation zwischen den Objekten oder Komponenten des verteilten Systems. Auf der Klientenseite baut der Broker die verteilten Aufrufe zusammen und leitet sie danach an den Server weiter. Auf der Server-Seite nimmt der Broker die Anfrage entgegen und baut daraus einen Aufruf zusammen, den er dann auf einem Server-Objekt durchführt ... Der Broker übernimmt alle Details der verteilten Kommunikation, wie Verbindungsaufbau, Marshalling der Nachricht etc., und verbirgt diese Details – so weit wie möglich – vor dem Klienten und dem verteilten Objekt.*"

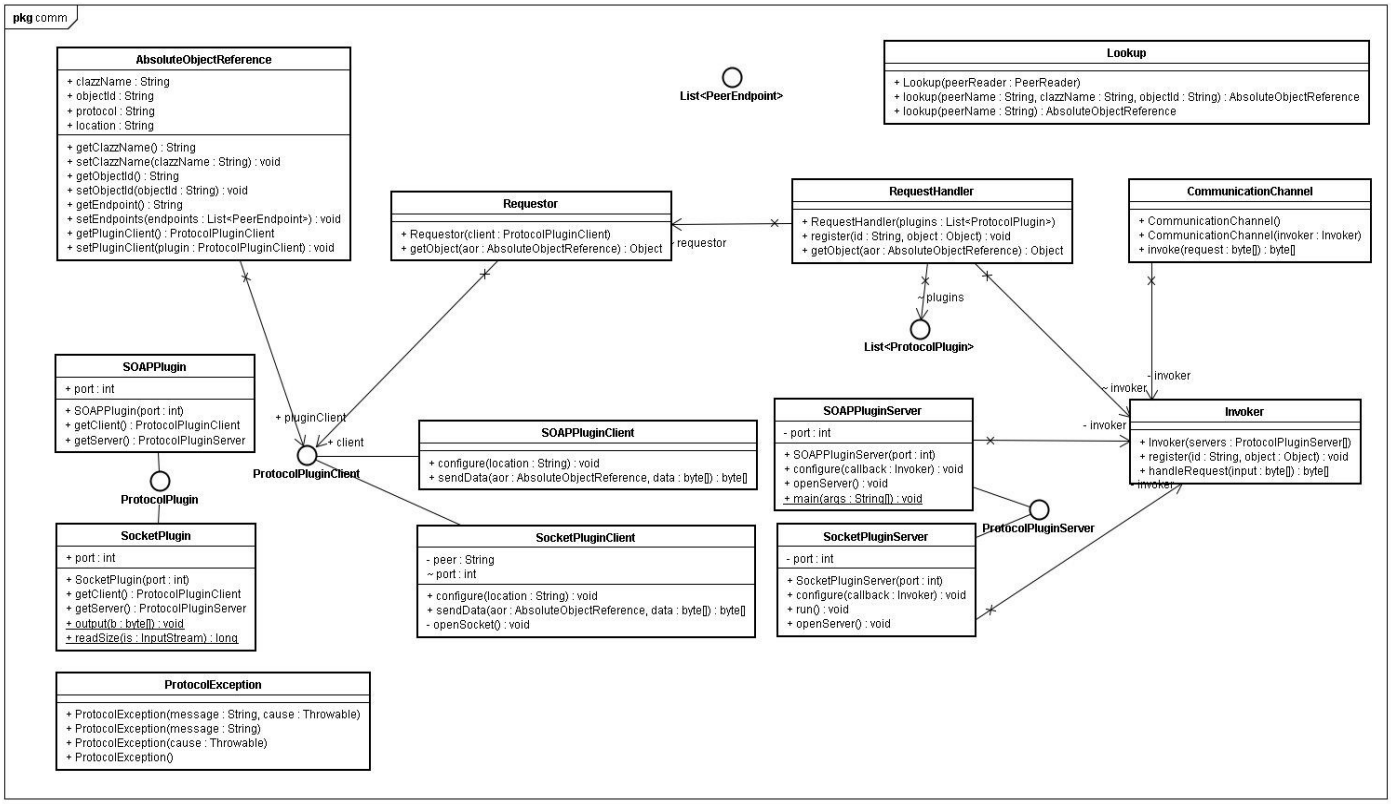
## *CommunicationFramework*

Die Kommunikation schließt nun folgende Klassen und Interfaces ein:

- **AbsoluteObjectReference** hat notwendige Informationen eines Peers, wie Protokoll und Bestimmungsort. Das AOR wird vom Requestor verwendet.
- **Lookup** liefert das AOR eines Peers zurück, das durch dessen Name indentifizierbar ist.

2 Software-Architektur; O.Vogel, I.Arnold, A.Chughtai, E.Ihler, U.Mehlig, Th.Neumann, M.Völter, U.Zdun; Spektrum2005

- **Requestor** bietet ein dynamisches Interface zum Aufruf von Methoden über den RequestHandler an.
- **RequestorHandler** arbeitet als Schnittstelle zwischen dem lokalen Peer und den Anfragen von entfernten Peers. Dabei nutzt der RequestHandler den Invoker für die einzelnen Server Instanzen.
- **Invoker** bietet die Methode `handleRequest(byte[])`, welche eingehende Anfragen abarbeitet.
- **ProtocolPluginServer** wird als Interface in den einzelnen Plugins implementiert und bearbeitet die eingehenden Aufrufe. Die einzelnen Protokolle werden beim Aufruf des Invokers instanziert und konfiguriert.
- **ProtocolPluginClient** ist als Interface in den Protokollen als Schnittstelle nach außen vorgesehen. Durch das AOR wird der richtige Requestor ausgewählt und verwendet um eine Anfrage an einen entfernten Peer zu senden.



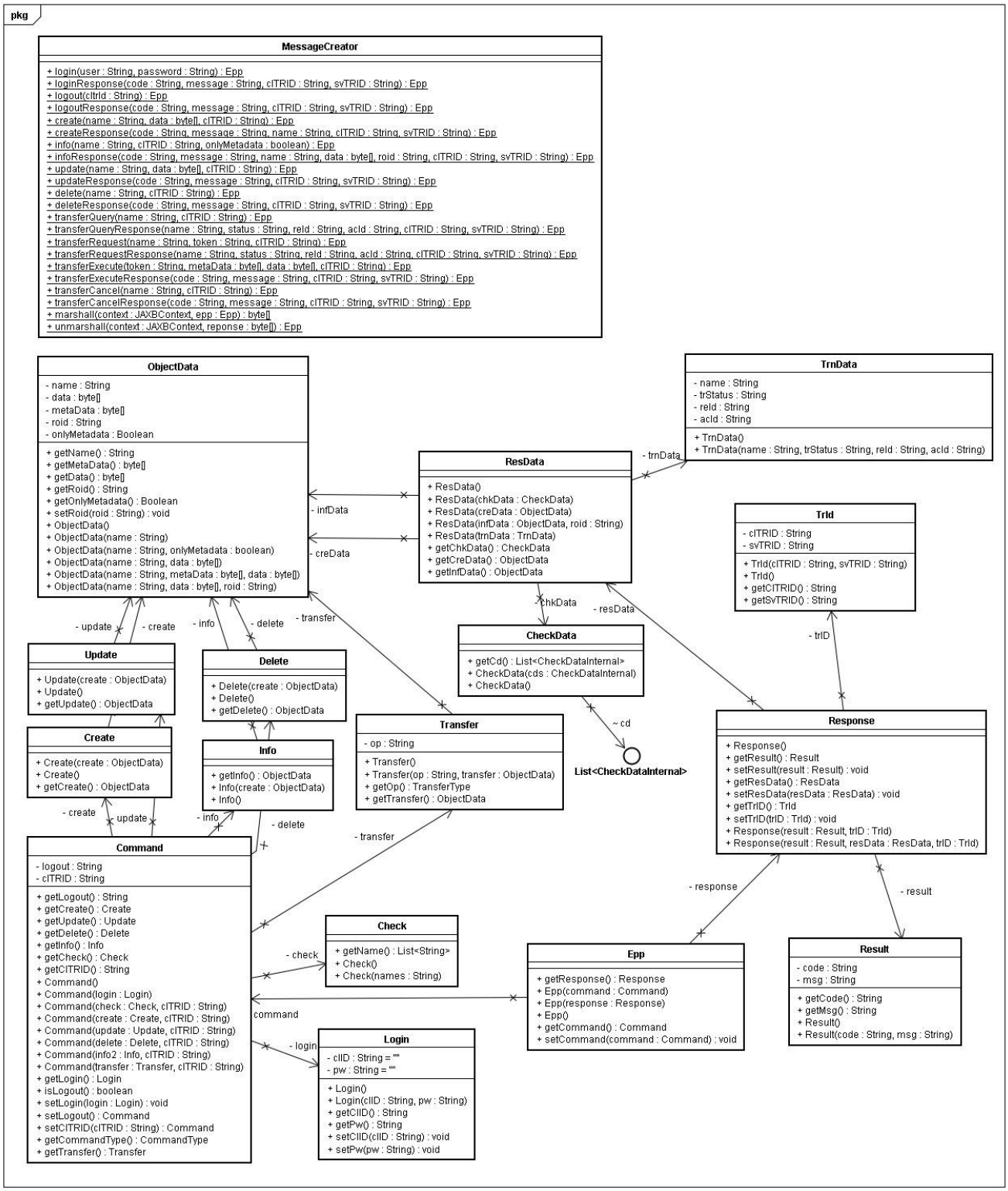

## *Mapping*

Die Interkommunikation erfolgt über das Extensible Provisioning Protocol[3]. Das EPP wird als XML Format übertragen. Die in der RFC[4] festgelegten Erweiterungen mussten nicht implementiert werden, gaben aber ein gutes Grundgerüst ab.

Den Aufruf des Marshallings übernimmt die **EppCommunication** Klasse aus dem Core package. Dabei wird der **MessageCreator** verwendet, der die einzelnen Befehle entsprechend mappt.

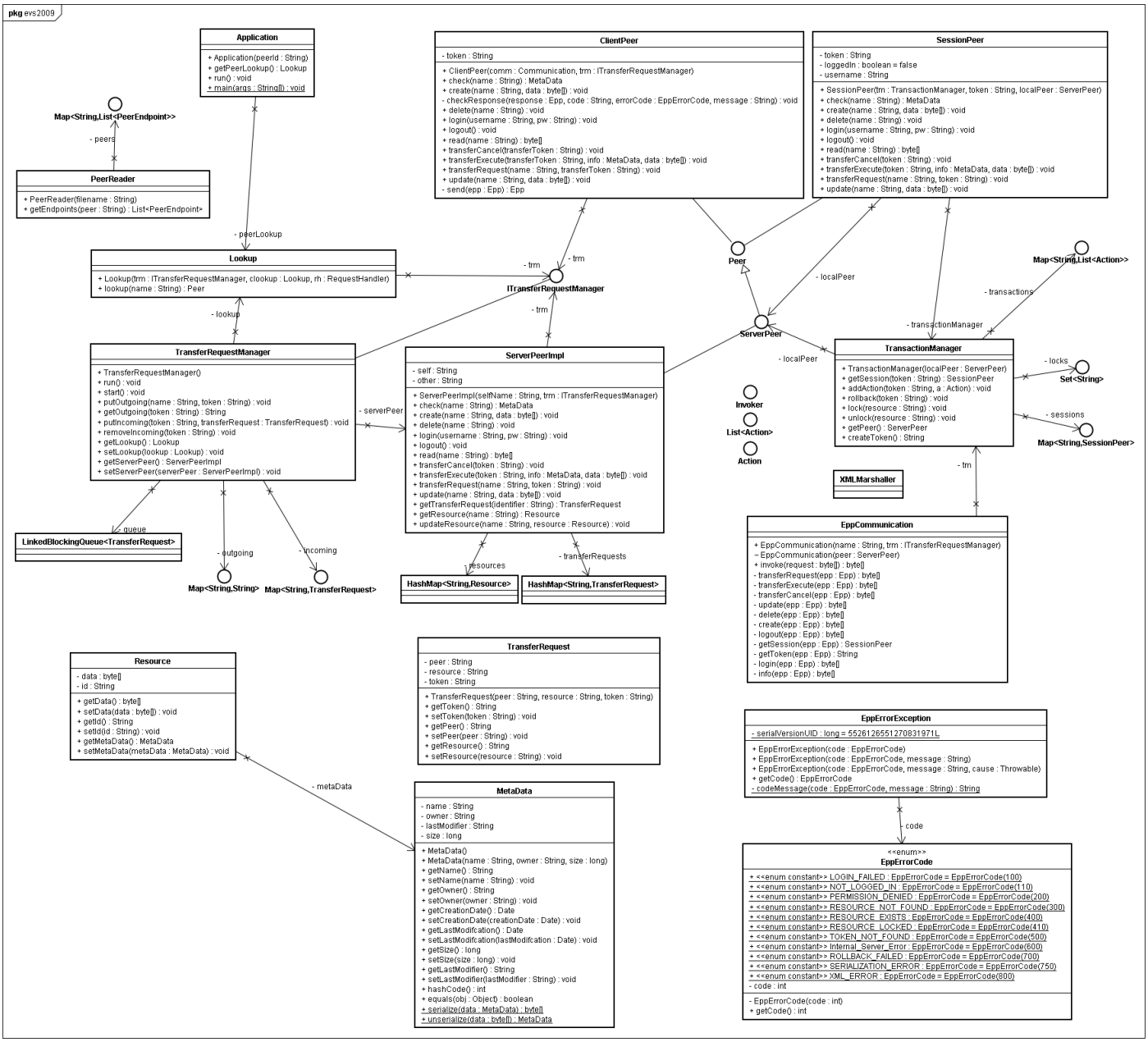
3 http://en.wikipedia.org/wiki/Extensible_Provisioning_Protocol
4 http://tools.ietf.org/html/rfc4930

Die Informationen werden in den jeweiligen Klassen aufgenommen, die so dann leichter verwendet werden können. Dabei kommen die **javax.xml** Klassen zur Anwendung.



## *Application*

Die Applikation wird durch die Auswahl eines Peers (Name) gestartet. Da kein NamingService verwendet wird, sondern eine statische Liste an teilnehmenden Peers (**peers.csv**), welche durch einen **PeerReader** ausgelesen werden kann, muss dieser Name auch vorher definiert werden.




Die Anforderung der ACID-Implementierung wird durch die Instanzen des **TransactionManager** und des **SessionPeer** gelöst. Ein entfernter Peer meint, seine Anfragen sofort an den Peer zu leiten, jedoch fängt diese der SessionPeer ab. Dieser kommuniziert dann mit dem TransactionManager, der Befehle erst nach erfolgreichem Endstatus an die Resources weiterleitet. Die Befehle werden in einer Liste zwischengespeichert.

# Annahmen

- Transfer Request/Execute ist ein Move Befehl
- Transfer Request/Execute sind asynchron. Requests werden in einer Queue gespeichert, vom Server später abgearbeitet und dann asynchron an den client geschickt.
- Transfer Request übersendet einen Token und den Namen vom gewünschten File. Der Server nimmt diesen Request in eine Queue auf. Sollte ein Request von dem selben Requestor da sein, kommt eine Errormessage. Bei TransferExecute schickt der Server dem Client den Token und die Daten (TransferExecute mit gesetztem Attribut op=Execute) zurück. Alle anderen Request für dieses File werden gelöscht. Bei einem TransferCancel wird der Request aus der Queue gelöscht.
- Transfer Execute Fehlermeldungen werden mittels Exception gegeben
- Jeder Peer hat einen Identifier (und müsste sich bei einem Naming-Service für seinen Protokolle registrieren. Das erfolgt bei uns statisch. Jeder Peer kennt alle anderen Peers)
- Der RequestHandler hat eine Zuordnung von Identifiern zu verwendetetem Protokoll/eindeutiger Netzwerkadresse in einem csv gespeichert (<ID>,<Protokoll>::<Location:Protokoll>)
- Deadlocks möglich. Locken auf einzelne Resource, dadurch → lost update, phantom read, dirty read
- Lock wird für gesamte Session gehalten wenn einmal genommen
- Owner einer Resource ist der Peer bei dem die Resource liegt
- Error einer Operation beendet die Session nicht
- Explizites beenden einer Session notwendig (kein Timeout vorerst) und damit auch kein automatischer rollback nötig
- TransactionManager hat eine große Collection mit gelockten Resources. Führt nach Commit die jeweiligen Befehle am Peer aus. Liefert SessionPeer anhand des SessionToken zurück. Bearbeitete Ressourcen sind in der obengenannten Collection dabei. DIRTY_READ für Lesezugriffe.
- SessionPeer - kommuniziert mit TransaktionManager und nicht direkt mit Peer. Hier läuft concurrent code ab.
- SessionToken - Jeder Session wird beim Login ein Token zugewiesen, dass bei jedem Kommando mitübertragen wird.

# Tests

| Testcases | Possible Errors |
|---|---|
| Login | Wrong credentials |
| Logout | Not logged in before |
| Create | Already created |
| Read | Not created |

| | |
|---|---|
| Update | Not created |
| Delete | Not created |
| Check | Not created |
| Transfer Request | Not created |
| Transfer Execute | Item already exists, Wrong Token, Internal Error (e.g. no space left) |
| All | No authorization, Session not closed → Rollback |

### *Login & Logout*

*) Login → Logout → Correct

*) Login (wrong password or username) → Error

*) Logout (not logged in) → Error

### *Create*

*) Login → Create → Read → Logout → Correct

*) Login → Create (already exists) → Error

### *Read*

*) Login → Create → Read → Logout → Correct

*) Login → Read (doesn't exist) → Error

### *Update*

*) Login → Create → Read → Update → Read → Logout → Correct

*) Login → Update (doesn't exist) → Error

### *Delete*

*) Login → Create → Read → Delete → Read (should result in Error) → Logout → Correct

*) Login → Delete (doesn't exist) → Error

### *Check*

*) Login → Create → Check → Logout → Correct

*) Login → Check (doesn't exist) → Error

### *TransferRequest*

*) Login → Create → TransferRequest → Logout → Correct

*) Login → TransferRequest (already exists for this requestor) → Error

*) Login → TransferRequest (doesn't exist) → Error

### *TransferExecute*

*) Login → TransferExecute → Read → Logout

*) Login → TransferExecute (Item already exists) → Error

*) Login → TransferExecute (Wrong Token) → Error

*) Login → TransferExecute (Internal Error) → Error

# Arbeitsaufteilung

### *Borko*

ProtocolPlugin
CommunicationFramework
Dokumentation erstellen

### *Greifeneder*

CommunicationFramework
Application
Testing

### *Motlik*

Application
EPP-Mapping
Testing

# Ausgelassene Punkte

Zur Zeit sind einige Dinge noch im Werden ...