

Physical Mobile Interaction Framework (PMIF) for Mobile Devices

Overview

- Introduction
- PMIF Architecture
- Implementation
- Writing an Mobile Interaction application by using PMIF
 - Installation
 - Communication between the enhanced physical object and the mobile device
 - Communication between the mobile device and the server
 - Server programming
- Eextending PMIF
- Conclusion

Introduction – Physical Mobile Interactions

→ Physical Mobile Interactions

- Typical Use Cases
- Typical technologies supporting this kind of interactions
 - RFID (Radio Frequency Identifier), Visual Codes, NFC (Near Field Communication), Bluetooth



interaction with visual codes



interaction based on RFID tags

Introduction – Problems by development

- existence of different communication technologies
 - Knowledge of details of the communication technologies
 - exact knowledge of rich set of technical APIs
 - Each kind of interactions needs a different approach
 - business code is infused with technical code
 - reduced the clarity of code
- marginal tool support
- insufficient reuse of existing software components

Introduction – goals and requirements

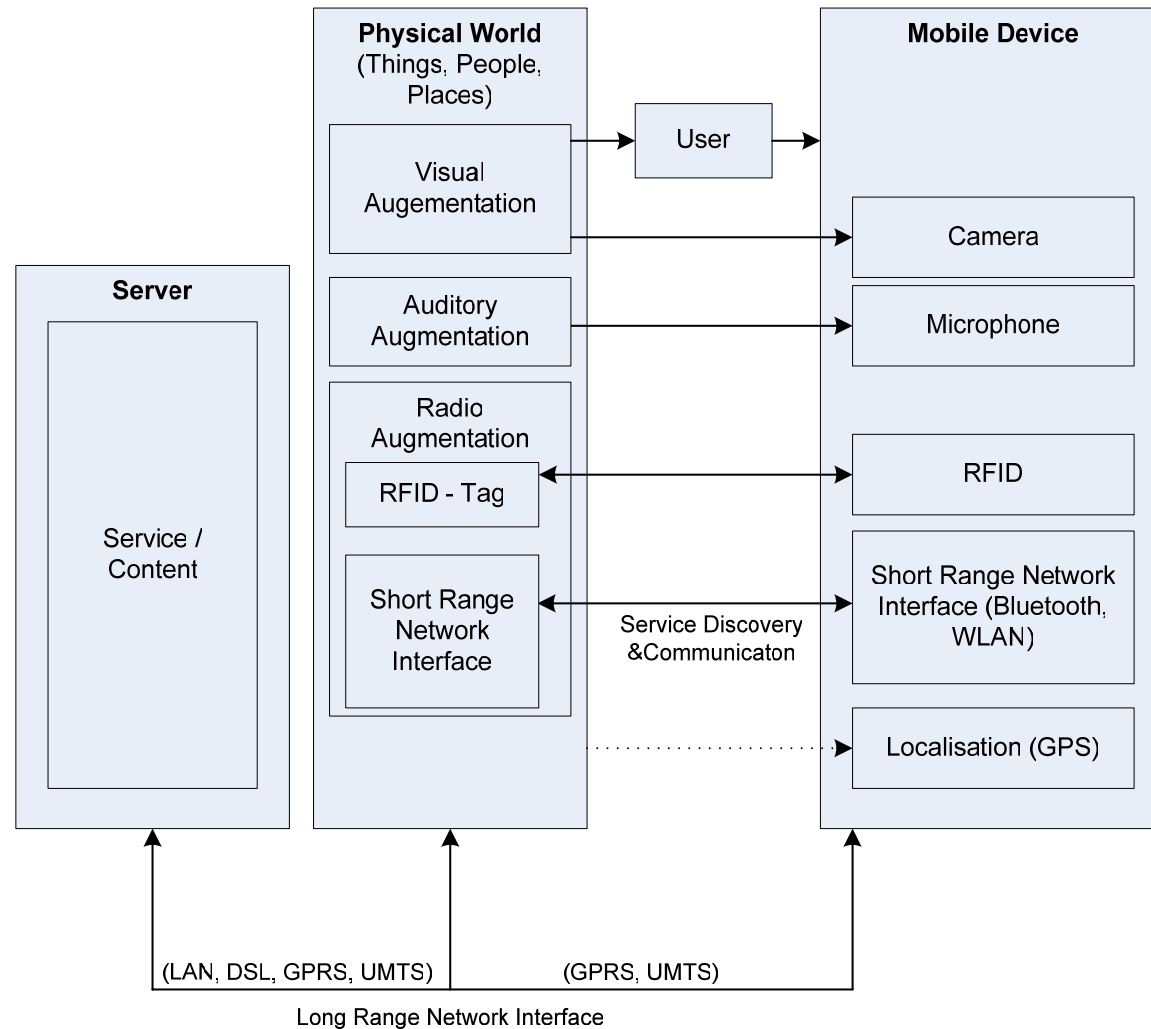
→ Requirements

- Support for the development and implementation
- Support for all relevant interaction techniques
- Provision of abstractions
- Orientation on existing and evolving standards:
 - J2ME, Conatcless Communication API (JSR 257), Bluetooth API (JSR 82), Location API (JSR 179)

→ Additional goals

- The ability to extend the framework with own component
- Separation of concerns
- Reuse of existing software components

Architecture - Overview (1)



Generic Architecture

Architecture - Overview (2)

→ Mobile Device

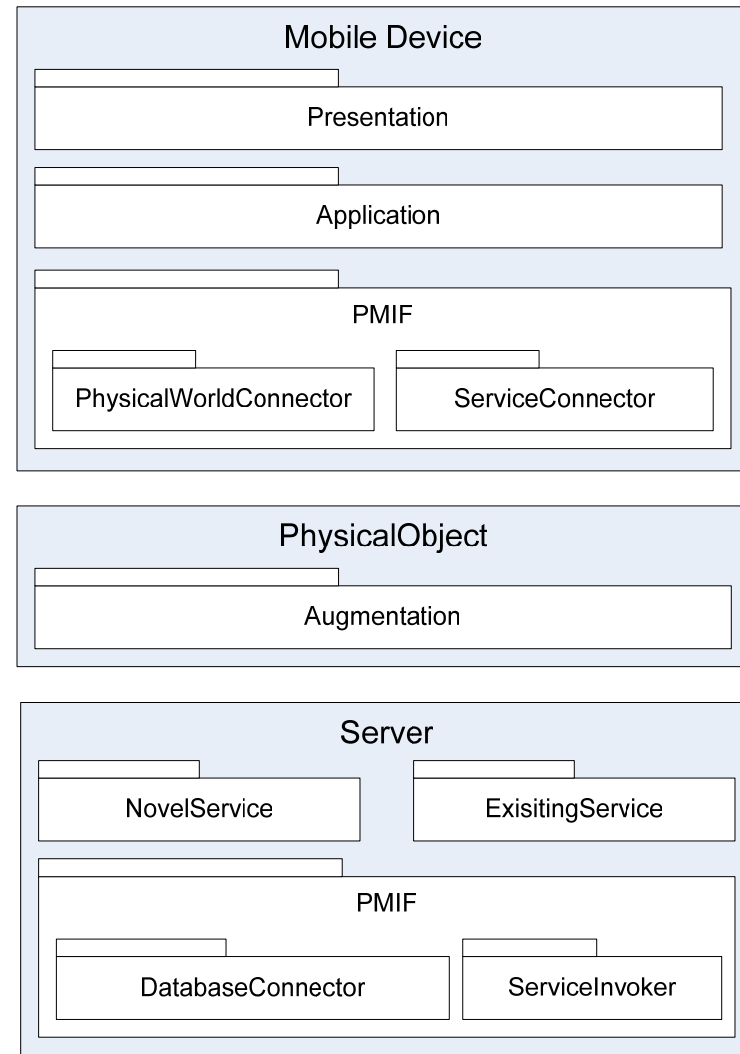
- ServiceConnector
- PhysicalWorldConnector

→ Physical Object

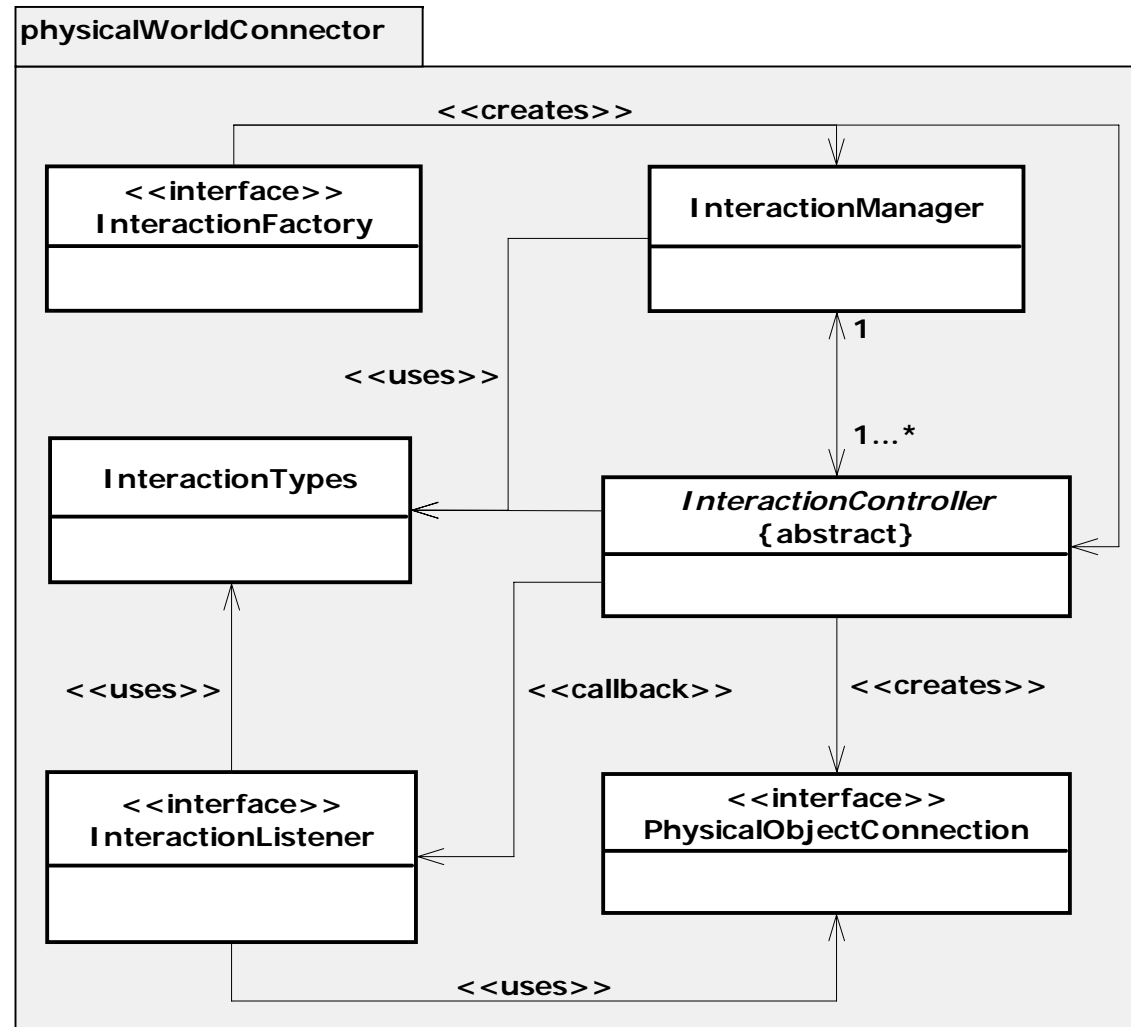
- posters which are augmented by visual markers
- machines which are augmented by RFID tags
- a public display which is augmented by a Bluetooth-based service

→ Server

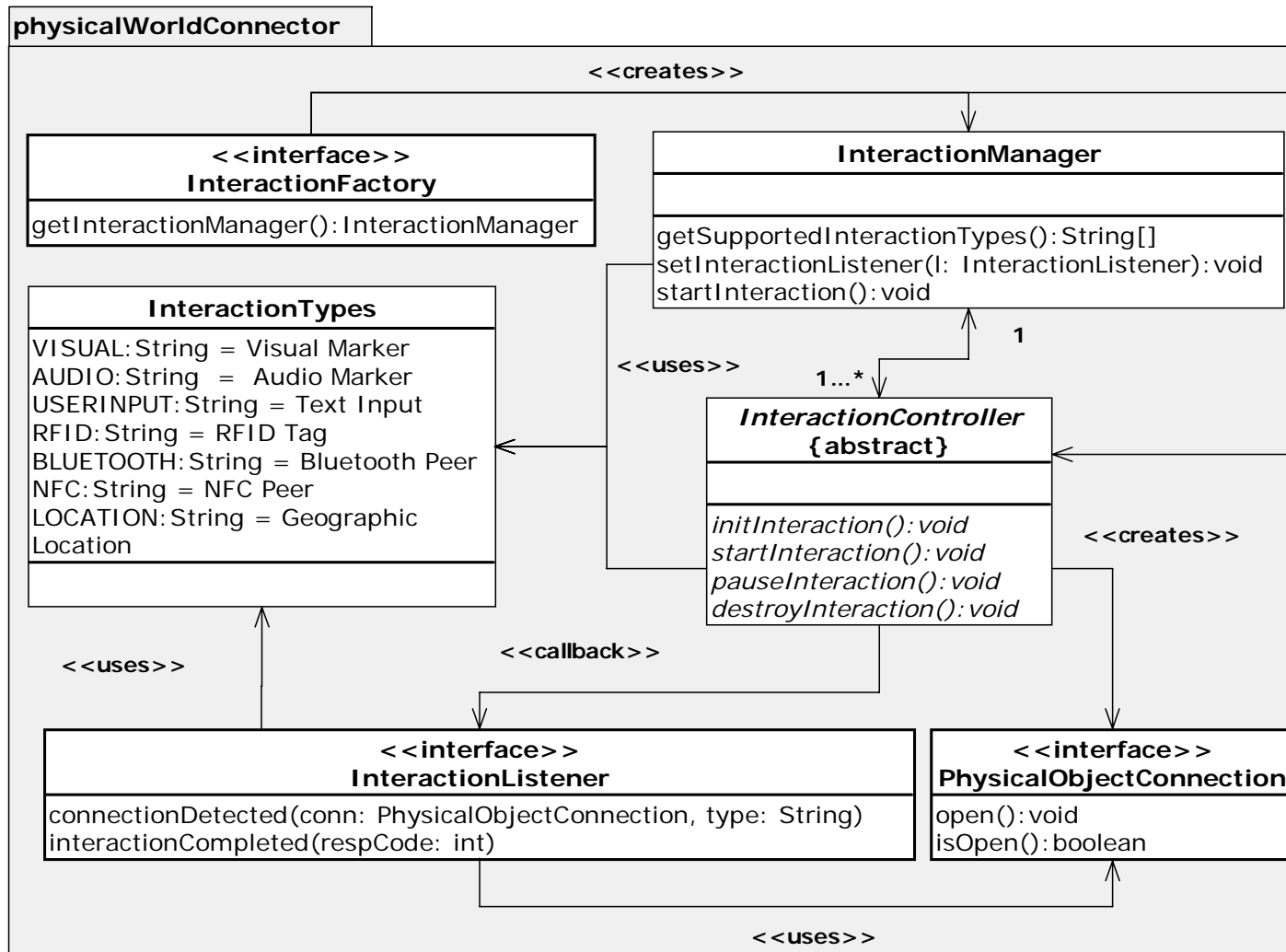
- PhysicalObjectsDatabase
- ServiceConnector



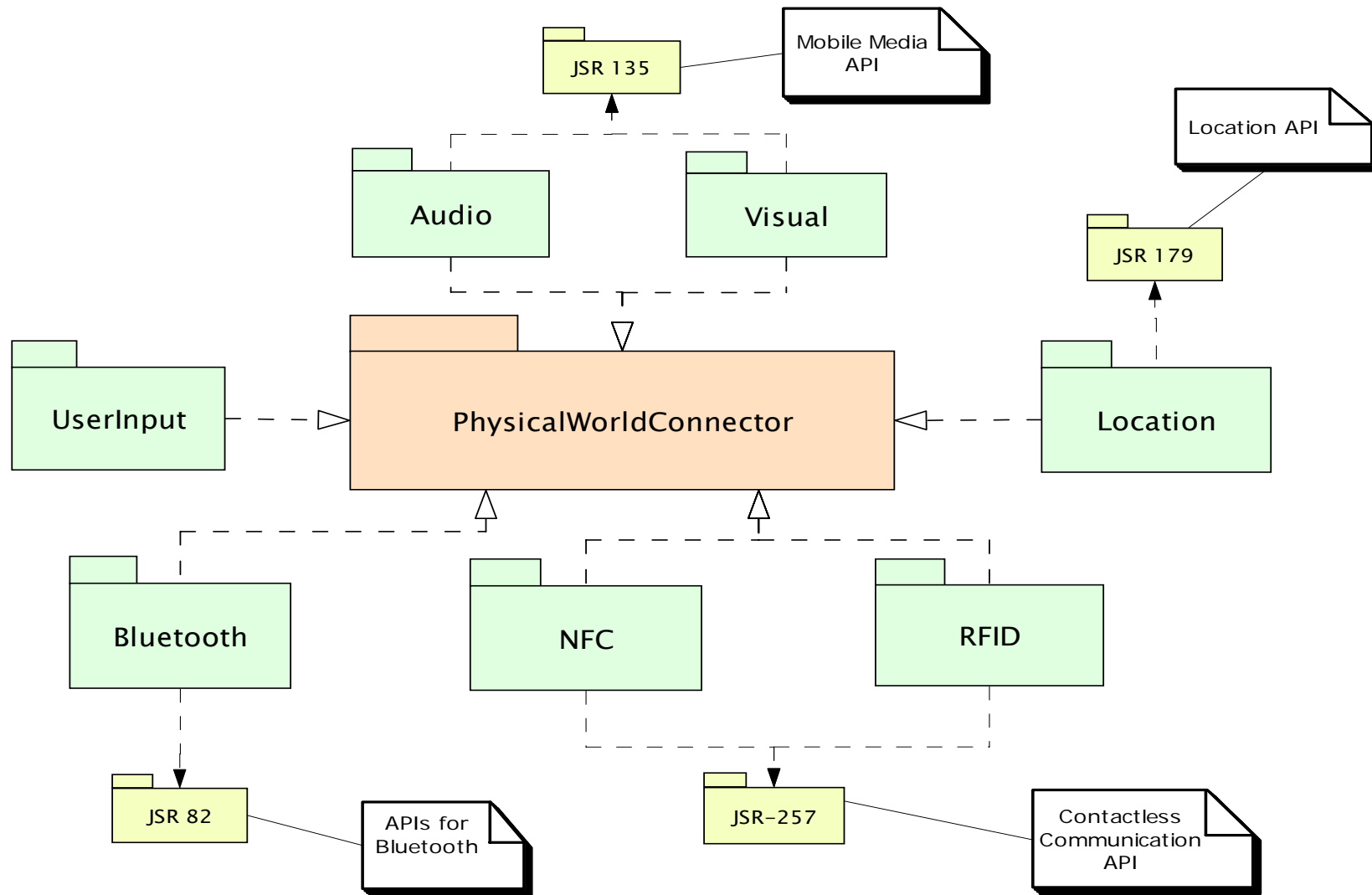
Implementation (1)



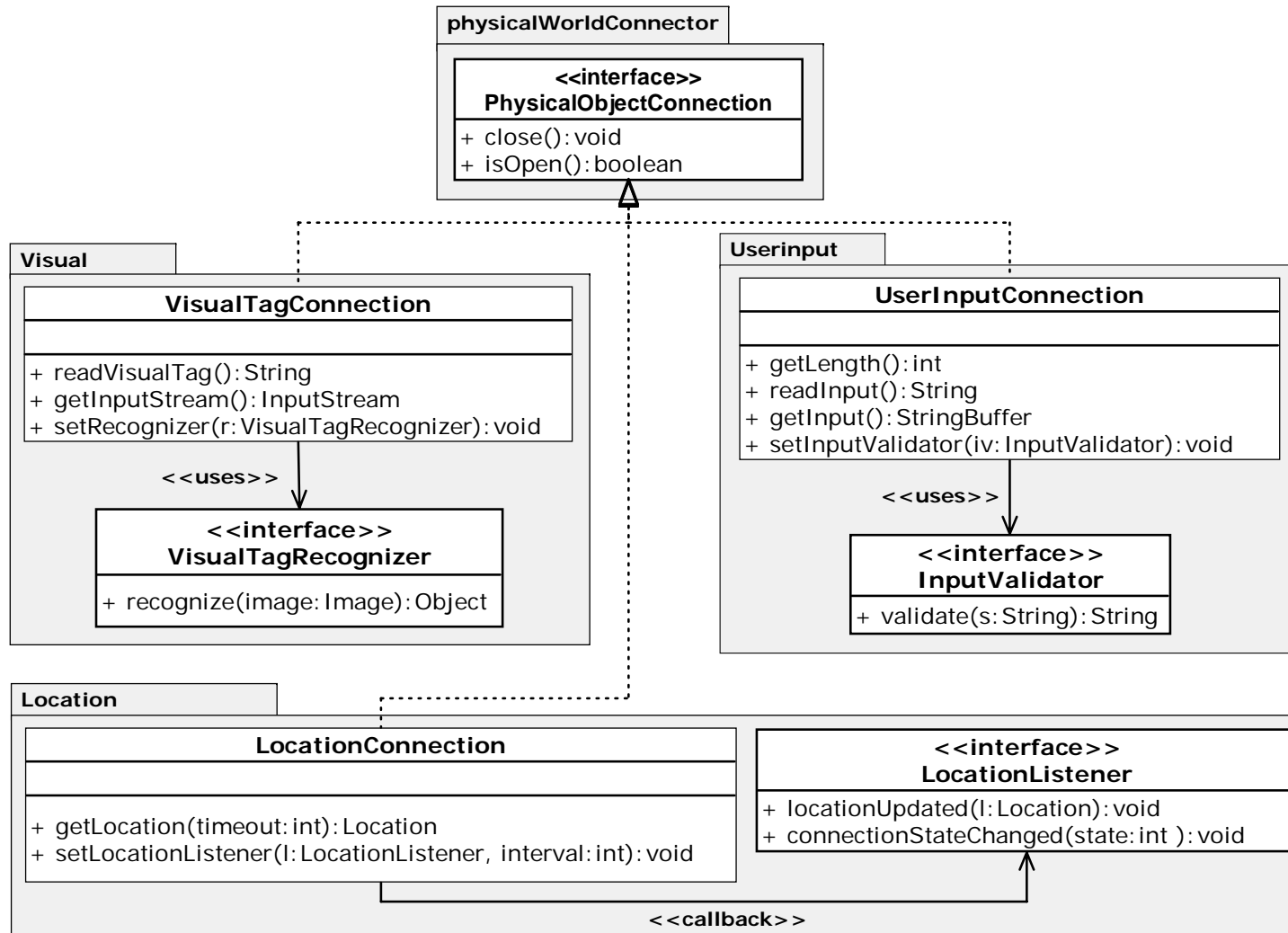
Implementation (2)



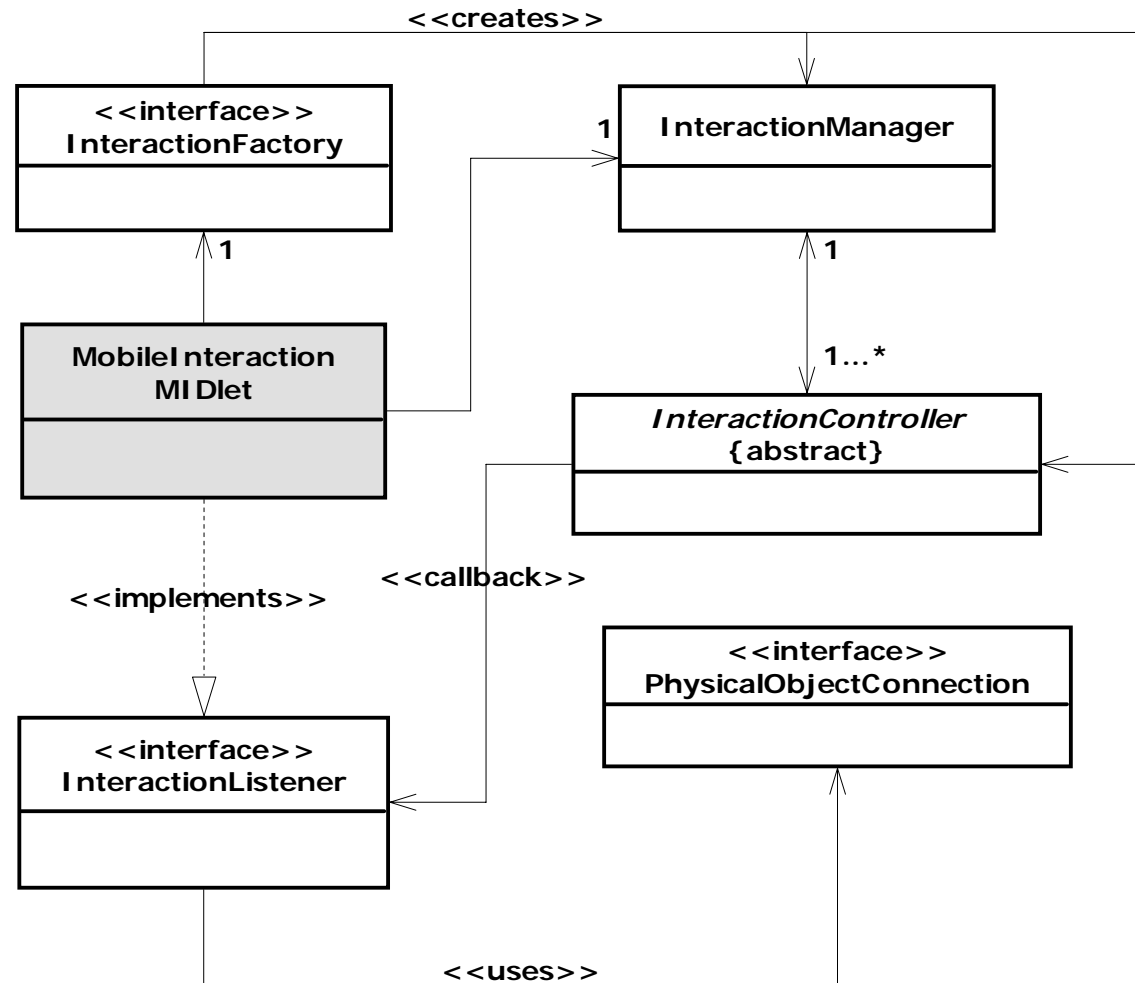
Implementation (3)



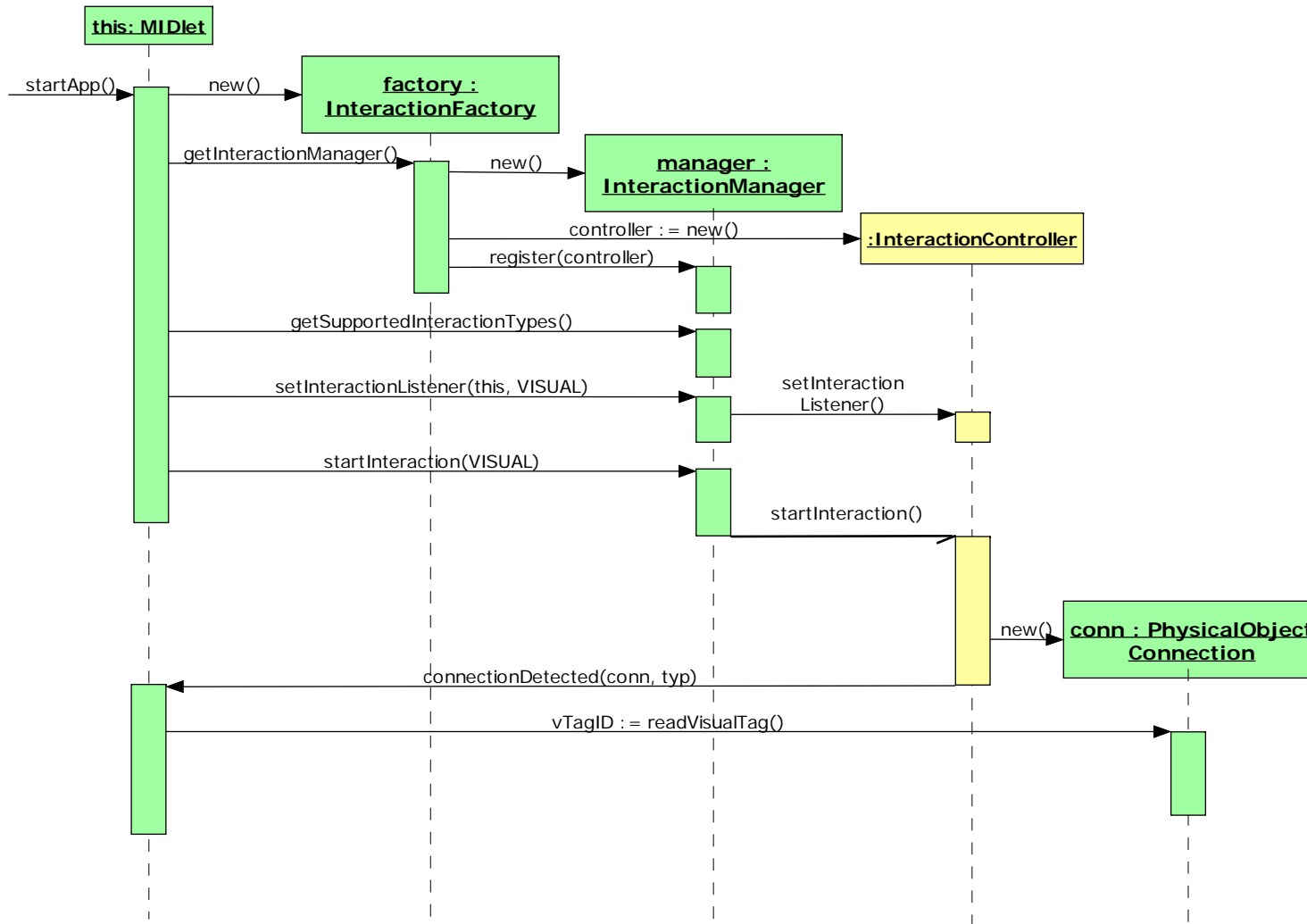
Implementation (4)



Implementation (5)



Implementation (6)



sequence diagram – initialization and start of interaction

Programmieren

→ Installation

- für mobiles Endgerät

- PMIF Projekt + Libraries:

- xmlpull_1_1_3_1.jar
 - kobjects-j2me.jar

- für Server-Programmierung

- WebServiceInvoker Projekt + Libraries:

- pmif-server.jar
 - ksoap2-j2me-full.jar
 - commons-dbutils-1.0.jar

→ Entwicklungsumgebung

- NetBeans 4.1

Programmieren

- Typische Schritte bei Implementierung einer Anwendung
 - Interaktion initialisieren und starten
 - Mit dem Objekt kommunizieren
 - Mit dem Server kommunizieren

Initialisierung und Start einer Interaktion

- DiscoveryManager instanziiieren
 - Konfiguration des Managers
 - Factory-Klasse
- Ein Listener für die Interaktion bei dem Manager anmelden
 - Ein Listener implementieren
 - Implementierungsarten
- Interaktion starten

Initialisierung und Start einer Interaktion

→ DiscoveryManager instanziiieren

■ Direktes Instanziiieren

```
1. manager = InteractionManager.getInstance(this);  
2. manager.register(new VisualMarkerInteractionController(),  
    InteractionTypes.VISUAL);  
3. manager.register(new TextInputInteractionController(),  
    InteractionTypes.USERINPUT);  
4. manager.register(new BluetoothInteractionController(),  
    InteractionTypes.BLUETOOTH);
```

■ Verwendung einer Factory-Klasse

```
1. InteractionFactory factory = new PMIFFactory(this); InteractionManager  
manager = factory.getInteractionManager();
```

Initialisierung und Start einer Interaktion

→ Definition einer Factory-Klasse

```
1. public class PMIFFactory implements InteractionFactory{
2.     MIDlet midlet;
3.     InteractionManager manager;
4.     public PMIFFactory(MIDlet midlet){
5.         this.midlet = midlet;
6.     }
7.     public InteractionManager getInteractionManager() {
8.
9.         manger = PhysicalObjectDiscoverManager.getInstance(midlet);
10.
11.         manager.register(new VisualInteractionController(),
12.             InteractionTypes.VISUAL);
13.         manager.register(new TextInputInteractionController(),
14.             InteractionTypes.USERINPUT);
15.     }
16. }
```

Initialisierung und Start einer Interaktion

→ Listener für einen Interaktionstyp anmelden

```
1. manager.setInteractionListener(interactionListener, InteractionTypes.VISUAL)
```

→ Definition eines Listener (als Innere Klasse)

```
1. public class MobileInteractionMIDlet extends MIDlet {
2.     ...
3.     //InnerClass
4.     class VisualInteractionListener implements InteractionListener {
5.         public void connectionDetected(PhysicalObjectConnection conn, String
6.         interactType){
7.             VisualTagConnection vConn = (VisualTagConnection) conn;
8.             //
9.         }
10.    }
11.    class BluetoothInteractionListener implements InteractionListener {
12.        public void connectionDetected(PhysicalObjectConnection conn, String
13.        interactType){
14.            BluetoothConnection bConn = (BluetoothConnection) conn;
15.            //
16.        }
17.    }
```

Initialisierung und Start einer Interaktion

→ Registrierung eines Listeners

```
1. public class MobileInteractionMIDlet extends MIDlet {
2.     InteractionFactory factory;
3.     InteractionManager manager;
4.     VisualInteractionListener visualListener;
5.     BluetoothInteractionListener bluetoothListener
6.     ...
7.     public void startApp() {
8.         factory = new PMIFFactory(this);
9.         manager = factory.getInteractionManager();
10.
11.         visualListener = new VisualInteractionListener();
12.         bluetoothListener = new BluetoothInteractionListener();
13.
14.         manager.setInteractionListener(visualListener,
15.             InteractionTypes.VISUAL);
16.         manager.setInteractionListener(bluetoothListener,
17.             InteractionTypes.BLUETOOTH)
18.     }
19.     ...
20. }
```

Initialisierung und Start einer Interaktion

→ Definition eines Listener (für alle Interaktionstypen)

```
1.  public class MobileInteractionMIDlet extends MIDlet
      implements InteractionListener {
2.      InteractionFactory factory;
3.      InteractionManager manager;
4.      ...
5.      public void sartApp() {
6.          factory = new PMIFFactory(this);
7.          manager = factory.getDiscoveryManager();
8.
9.          manager.setInteractionListener(this, InteractionTypes.VISUAL);
10.         manager.setInteractionListener(this, InteractionTypes.BLUETOOTH);
11.     }
12.     ...
13.     //Implements InteractionListener
14.     public void connectionDetected(PhysicalObjectConnection conn,
      String interactType){
15.         //
16.     }
17.     public void interactionTerminated(int respCode){
18.         //
19.     }
```

Initialisierung und Start einer Interaktion

→ Start der Interaktion

■ Implizites Starten

```
1. manager.setInteractionListener(this, InteractionTypes.VISUAL);  
2. manager.setInteractionListener(this, InteractionTypes.BLUETOOTH);  
3. manager.startInteraction(); // startet Bluetooth-Interaktion
```

■ Explizites Starten

```
1. manager.setInteractionListener(this, InteractionTypes.VISUAL);  
2. manager.setInteractionListener(this, InteractionTypes.BLUETOOTH);  
3. manager.startInteraction(InteractionTypes.VISUAL); // startet Visual-Interaktion
```

Nach dem Start übernimmt der zuständige InteractionController die Steuerung der Interaktion

Auf Interaktionsereignisse reagieren

- Die Kommunikation zwischen dem InteractionController und der Anwendung geschieht über InteractionListener Interface
- Der Listener stellt zwei Methoden bereit, mit deren Hilfe der Controller die Ereignisse an den Listener melden kann
 - `interactionTerminated()`
 - `connectionDetected()`

Auf Interaktionsereignisse reagieren

→ Meldung der Beendigung einer Interaktion

```
1. //Implements InteactionListener
2. public void interactionTerminated(int respCode){
3.     //
4.     switch (respCode) {
5.         //Abbruch durch den User
6.         case InteractionListener.INTERACTION_TERMINATED:
7.             //
8.             break;
9.         //keine für den Interaktionstyp passende Objekte wurden entdeckt
10.        case InteractionListener.INTERACTION_NO_OBJECTS:
11.            //
12.            break;
13.        //Abbruch im Folge einer Fehler
14.        case InteractionListener.INTERACTION_ERROR:
15.            //
16.            break;
17.    }
18. }
```

Auf Interaktionsereignisse reagieren

→ Meldung einer PhysicalObjectConnection und ihr Casting

```
1. //Listener ist nur für Bluetooth-Interaktionen zuständig
2. public void connectionDetected(PhysicalObjectConnection conn, String
   interactType){
3.     BluetoothConnection bConn;
   //Connection conn ist eine BluetoothConnection
4.     bConn = (BluetoothConnection) conn;
5. }
```

```
1. //Listener ist für alle Interactionstypen zuständig,
2. //anhang des zweiten Parameters kann man entscheiden
3. //wie das PhysicalObjectConnection gecastet werden soll.
4. public void connectionDetected(PhysicalObjectConnection conn, String
   interactType){
5.     if (inteactType == InteractionTypes.VISUAL) {
6.         VisualTagConnection vConn = (VisualTagConnection) conn;
7.         //Connection kann genutzt werden
8.     }else if (inteactType == InteractionTypes.BLUETOOTH) {
9.         BluetoothConnection bConn = (BluetoothConnection) conn;
10.        //Connection kann genutzt werden
11.    }
12. }
```

PhysicalObjectConnection

→ VisualTagConnection

```
1.  VisualTagRecognizer visualTagRecognizer;  
2.  ...  
3.  public void connectionDetected(PhysicalObjectConnection conn, String  
    interactType){  
4.      VisualTagConnection vConn = (VisualTagConnection) conn;  
5.      visualTagRecognizer = new VisualCodeRecognizer();  
6.      vConn.setRecognizer(visualTagRecognizer);  
7.      String tagID = vConn.readVisualTag();  
8.      if (tagID != null) {  
9.          // tagID kann für weitere Schritte genutzt werden  
10.     }  
11.     //Mann kann auch auf die Daten über ein InputStream zugreifen  
12.     InputStream is = vConn.getInputStream();  
13. }
```

■ VisualTagRecognizer Interface

```
1.  public interface VisualTagRecognizer {  
2.      public Object recognize(Image image);  
3.  }
```

PhysicalObjectConnection

→ BluetoothConnection

```
1. public void connectionDetected(PhysicalObjectConnection conn, String  
   interactType){  
2.     BluetoothConnection bConn = (BluetoothConnection) conn;  
   String sID = bConn.readString();  
3. }
```

→ UserInputConnection

```
1. public void connectionDetected(PhysicalObjectConnection conn, String  
   interactType){  
2.     UserInputConnection uConn = (UserInputConnection) conn;  
3.     //einen InputValidator setzen  
4.     uConn.setInputValidator(new NumberValidator());  
5.     String sID = uConn.readInput();  
6. }
```

■ InputValidator Interface

```
1. public interface InputValidator {  
2.     public String validate(String userInput);  
3. }
```

Kommunikation mit dem Server

- für die Kommunikation mit dem Server wird das SOAP Protokoll eingesetzt
- Web Service
 - erlaubt, ihre Methoden über eine Web-Schnittstelle aufzurufen
 - Der Aufruf der Methoden und die Beschreibung basieren auf XML
- SOAP
 - Kommunikation erfolgt mittels SOAP Nachrichten
 - ein selbstbeschreibendes einfaches XML-basiertes Format
 - setzt auf HTTP als Übertragungsprotokoll

Kommunikation mit dem Server

→ Aufbau eine SOAP-Nachricht (Request)

Aufruf einer Methodes in Java:

```
1. InfoWebServiceIF infoService = new InfoWebService();  
2. String strInfo = infoService. getObjectInfo("1");
```

SOAP:

```
1. <v:Envelope xmlns:i="http://www.w3.org/2001/XMLSchema-instance"  
2.     xmlns:d="http://www.w3.org/2001/XMLSchema"  
3.     xmlns:c="http://www.w3.org/2001/12/soap-encoding"  
4.     xmlns:v="http://www.w3.org/2001/12/soap-envelope">  
5.     <v:Header />  
6.     <v:Body>  
7.         <n0:getObjectInfo id="o0" c:root="1" xmlns:n0="medien.ifi.lmu">  
8.             <objID i:type="d:string">1</objID>  
9.         </n0:getObjectInfo>  
10.     </v:Body>  
11. </v:Envelope>
```

Kommunikation mit dem Server

→ Aufbau einer SOAP-Nachricht (Response)

SOAP – Antwort:

```
1. <v:Envelope xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
2.     xmlns:d="http://www.w3.org/2001/XMLSchema"
3.     xmlns:c="http://www.w3.org/2001/12/soap-encoding"
4.     xmlns:v="http://www.w3.org/2001/12/soap-envelope">
5.     <v:Header />
6.     <v:Body>
7.         <n0:getObjectInfoResponse id="o0" c:root="1"
xmlns:n0="medien.ifi.lmu">
8.             <return i:type="d:string">
9.                 SYSTEMS 2005 - Hier finden Sie alle Informationen
f&#252;r Aussteller, Besucher und Journalisten
10.             </return>
11.         </n0:getObjectInfoResponse>
12.     </v:Body>
13. </v:Envelope>
```

Kommunikation mit dem Server

→ **ServiceConnector** Interface

- Zuständig für die Kommunikation mit dem Server, definiert drei Methoden:
- **open()** - öffnet eine Verbindung zum Server
- **invoke()** - führt den Aufruf Web Service Operation durch
- **close()** - schließt die Verbindung

→ **HttpWsConnector** Klasse

- implementiert **ServiceConnector** Interface
- setzt auf HTTP als Übertragungsprotokoll

```
1. String url =  
   "http://devel.medien.ifi.lmu.de:8080/ServiceInvoker?service=InfoWebService";  
2. ServiceConnector service = new HttpWsConnector();  
3. service.open(url);  
4. SoapObject method = new SoapObject("lmu.ifi.media.webservices", "getObjectInfo");  
5. method.addProperty("objectID", "123");  
6. String str = service.invoke(method); //eigentliche Aufruf
```

Kommunikation mit dem Server

→ Eigenes ServiceConnector implementieren

■ Definiere ein **XyzWebService** Interface

```
1. public interface InfoWebServiceIF {
2.     public String getObjectInfo(String objID);
3.     public String getURL(String objID);
4. }
```

■ Definiere eine Klasse die von einem **HTTPWsConnector** abgeleitet ist und das **XyzWebService** Interface implementiert.

```
1. public class InfoWebService implements InfoWebServiceIF extends HttpWsConnector
   {
2.     public InfoWebServiceConnector(){
3.         super();
         open("http://devel.medien.ifi.lmu.de:8080/ServiceInvoker?service=InfoWebSer
   vice");
4.     }
5.     ...
```

Kommunikation mit dem Server

```
6.     public String getObjectInfo(String objID){
7.         SoapObject method = new SoapObject("lmu.ifi.media.webservices",
"getObjectInfo");
        method.addProperty("objectID", objID);
        return invoke(method);
8.     }

9.     public String getURL(String objID) {
10.        SoapObject method = new SoapObject("lmu.ifi.media.webservices", "getURL");
11.        method.addProperty("objectID", objID);
12.        return invoke(method);
13.    }
14. }
```

→ Nutzung der Klasse

```
1. InfoWebServiceIF infoService = new InfoWebService ();
2. public void connectionDetected(PhysicalObjectConnection conn, String
interactType){
3.     UserInputConnection uConn = (UserInputConnection) conn;
4.     //einen InputValidator setzen
5.     uConn.setInputValidator(new NumberValidator());
6.     String sID = uConn.readInput();
7.     String url = infoService.getURL(sID); //eigentliche Aufruf
8. }
```

Serverprogrammierung

- WebServiceInvoker Servlet
 - Nimmt SOAP-Anfragen von den Clients an
 - ruft den Web Service auf
 - Verpackt die Ergebnisse als SOAP-Nachricht und schickt sie an den Client zurück
- DatabaseConnector
 - Connects to the database specified in properties using driver, user and password also specified there.
- DbUtils Package
 - a small set of classes designed to make working with JDBC easier
[<http://jakarta.apache.org/commons/dbutils/>]

Serverprogrammierung

- Eigenes Web Service implementieren, durch definition einer Klasse, die
- **XyzWebService** Interface implementiert
 - von dem **WebServiceInvoker**-Servlet abgeleitet ist

```
1. public class InfoWebService extends WebServiceInvoker implemets InfoWebServiceIF
   {
2.
3. public String getObjectInfo(String objID) {
4.     return (String) infos.get(objID);
5. }
6.
7.     public String getURL(String objID){
8.         return (String) urls.get(objID);
9.     }
10. }
```

Physical Mobile Interaction Framework

- Java-Framework zur Entwicklung von Physical Mobile Interaction Anwendungen
 - Einfache Nutzung, der Entwickler kann sich voll und ganz auf die Anwendungsfunktionalität konzentrieren
 - Keine Beschäftigung mit technischer Details für die spezielle Interaktionstechnik
 - Verschiedene Interaktionstechniken werden einheitlich auf einer höheren Abstraktionsebene behandelt
 - Wiederverwendbarkeit von Komponenten