Development Support for Hardware Oriented Software Applications using Eclipse IDE.

Master Thesis
by
Tarlapally Sri Ram Phani Kumar
October 30, 2006

Submitted in partial fulfillment of the requirements for the degree
Master of Science in Information and Media Technology

supervised by
Prof. Dr. Ralf Möller (TUHH)
Prof. Dr. Ulrich Killat (TUHH)
Dipl.-Inf. Daniel Barisic (Infineon Technologies AG)

Hamburg University of Technology
Software Systems Group (STS)

Declaration

I Tarlapally Sri Ram Phani Kumar (Matr. No. 26679), student of Master of Science in Information and Media Technologies at Hamburg University of Technology, hereby declare that this work bas been prepared by myself, all literal or content based quotations are clearly pointed out, and no other sources or aids than the declared ones have been used.

(Tarlapally Sri Ram Phani Kumar)

Acknowledgments

I take this opportunity to convey my sincere thanks to Prof. Dr. Ralf Möller, Software Systems Group and Prof. Dr. Ulrich Killat, Communication Networks, Hamburg University of Technology for supervising my master thesis.

My sincere thanks to Dipl.-Inf. Daniel Barisic, Infineon Technologies AG, München for supervising my master thesis, and for his guidance and help during the course of work.

I would like to thank Dr. Guido Stromberg, Infineon Technologies AG, München for for his invaluable suggestions and motivation.

My earnest gratitude to my family, for their love and encouragement. It is only with their constant support and cooperation that I have been able to successfully.

Contents

List of Figures

Abstract

Sindrion project, a research project at Infineon Technologies AG, integrates low-cost and low-power wireless embedded devices into a Universal Plug and Play (UPnP) network. Integration of these devices is done by so called proxy applications which are programs executed on an ordinary PC. For each embedded device enhanced by a Sindrion Transceiver a specific set of Java applications are to be developed. The developer needs to deal with diverse issues in the field of distributed components; hence a development support in the form of an IDE is advised.

This thesis deals with concept, design and implementation of mechanisms that provide development support for Sindrion Applications. To achieve this, the development process of Sindrion Applications is analyzed from a developer's point of view, considering the interdependencies between hardware and software components in the domain of distributed embedded devices. Specific parts of this software development process are supported by already existing tools viz. libraries, toolboxes and simulated environments. The thesis develops mechanisms for information flow and data storage by which these independent tools can be combined to form an IDE. Additionally, mechanisms that guide the developer to set-up the environment and give a quick start by initializing the application development process are designed. An initial version of the Sindrion Integrated Development Environment has been implemented based on the Eclipse framework, which shows the capabilities and the feasibility of the overall concept. The resulting Sindrion Integrated Development Environment helps even inexperienced developers to create sophisticated Sindrion applications.

Chapter 1

Introduction

1.1 Motivation

This Master Thesis is focused on providing means to support the development process of applications for interoperable embedded devices in ubiquitous computing environment, where a large number of small devices are spread in the surrounding environment to monitor and provide information which benefits the users.

Among various projects that are situated in the ubiquitous computing area, this thesis deals with a project called Sindrion. The goal of the Sindrion project is to allow the simple and unobtrusive integration of arbitrary embedded devices in a ubiquitous computing environment. In order to realize this vision, the main challenge of Sindrion lies in various aspects of embedded systems design, and design of means to provide interoperability to the number of distributed embedded devices. To this end, interoperation of devices is based on the standardized UPnP middleware that supports device discovery, description and service-oriented control and eventing mechanisms on top of the common TCP/IP internet protocol suite. The devices are networked wirelessly with the help of so-called Sindrion transceivers, which comprise of low-power RF transmitter and receiver modules and a microcontroller with limited resources in terms of computing power and RAM. In order to compensate the computational limitations of the Sindrion transceiver, the expensive calculations necessary for participation in UPnP are sourced out to so-called terminals in the form of dedicated Java applications. These applications are directly provided by the transceiver and are executed on terminals, which basically are PCs, equipped with a special life cycle management software. As the desired UPnP interaction differs for each embedded device, a corresponding set of UPnP descriptions and Java applications have to be implemented. The development of these applications includes implementing, testing and deploying these applications to the transceiver. In this process the developer first needs to develop the UPnP descriptions. Corresponding Java applications that are in accord with the UPnP descriptions are to be implemented. The developer as well needs to some how tie up the Java applications developed with the descriptions and keep them in sync. Once these applications are developed they are to be tested before transferring them to the Sindrion Transceiver. During this whole process the developer needs to keep track of the resources for particular application as they may need to be changed or updated in case of errors, and this makes this development process a complex task. In this Master Thesis an Integrated Development Environment which assists the developer during the complete development process and therefore ease the development of Sindrion applications is designed.

1.2 The Sindrion Concept

The role of middleware in a distributed computing system is to eases the task of designing, programming and managing distributed application by abstracting the complexity and heterogeneity of the underlying distributed environment. This is achieved by providing a simple, consistent and distributed software layer, or 'platform' that lies between the operating system and the applications, which provides uniform, standard, high-level interfaces to the application developers and integrators. Sindrion aims at implementing a system that integrates smalls, embedded devices in high-level distributed system like UPnP (middleware) network by using so-called proxy based applications. These proxy applications act on behalf of devices in the UPnP network and therefore relieve these devices from computational load necessary to interact from in the network. Requests from the UPnP network to the devices will be translated from the proxy into one or multiple messages using a simple, easily parsable protocol. The decision to adopt UPnP as the dedicated middleware to allow interoperation of embedded devices has a great impact on the Sindrion concept. Hence we first need to understand the underlying concepts and features of UPnP (Universal Plug and Play) in order to understand the area of operations of this Thesis.

1.2.1 UPnP

Universal Plug and Play (UPnP) [1] is a service-based middleware that provides almost self configuring peer-to-peer connectivity between networked devices. It is based on the definition of standards for inter-device communication which are completely independent of the physical network layer, the operating system and the programming language. UPnP uses existing standards for communication and data exchange that are situated above the TCP/UDP layer like HTTP and SOAP [2]. Furthermore, it supports Zero-configuration and automatic discovery. Hence dynamically joining the network, acquiring an IP address, publishing its capabilities, and retrieving information about the presence and capabilities of other devices, belong to the device's abilities. The devices functionality is provided as services. The so-called UPnP forum established UPnP as an open standard, thus providing no drivers or APIs. Nevertheless, implementations of UPnP APIs have been conducted by different vendors, e.g. Intel [4] or Infineon.

The participants of a UPnP network are divided into two groups:

- *UPnP Device*: A UPnP device provides functionality to the UPnP network. This functionality is encapsulated into the device's *services*. The device properties and its services are conveyed to the network and can then be utilized by other UPnP participants. For more elaborate UPnP devices, a hierarchical device structure can be implemented which allows the arrangement of a number of subdevices. These subdevices encapsulate certain aspects of the device under a single *root device*. For instance, a UPnP device of hi-fi system could consist of hi-fi system root device and a CD player subdevice.
- *UPnP Control Point (CP)*: A control point basically uses the functionality provided by one or more UPnP devices. Furthermore it can discover devices or services in the network.

The separation of devices and control points is only done on a logical basis. From this follows that it is possible for a single physical device to embed either UPnP devices, control points or a combination of both.

A unique operational sequence has been defined to allow the interaction of distributed UPnP devices and control points. Let us now take a look at this sequence in order to create better understanding of the features of UPnP and the requirements that arise for devices which intend to participate in the network.

Operational Sequence

As we can see in Figure 1.1, the main features comprised in UPnP are **Addressing**, **Discovery**, **Description**, **Control**, **Eventing** and **Presentation**. These features are not isolated but organized in a layered structure, creating dependencies between the features. For example **Addressing** is needed to perform **Discovery**.

0. **Addressing** UPnP is based on IP, thus creating the need to assign an IP address to each device in the network. A UPnP device retrieves its IP address either via DHCP [6] or via Auto-IP [7].

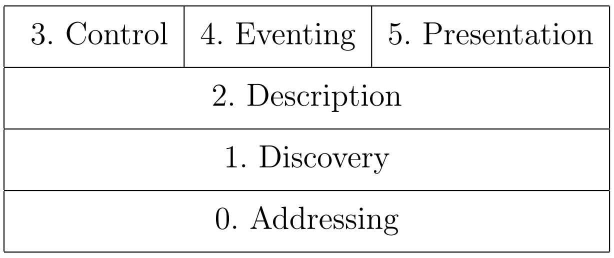


Figure 1.1: UPnP Layer Model

1. **Discovery** UPnP allows devices to flexibly join and leave the network. As there is no central management component in a UPnP network, the devices need to announce their existence and their capabilities by themselves in the discovery phase.

   During *Discovery*, a UPnP device informs the UPnP network about its presence and the services it offer. Provision of this information can be done actively, e.g. on start-up of a device, and as a response to look up message of a UPnP control point:

   - **Advertisement of Properties** Each UPnP device broadcasts its properties to the UPnP network on start-up. For this purpose, a certain multicast address has been specified. The messages which are sent to this address are encoded in the XML based *Simple Service Discovery Protocol* (SSDP) [5]. There are four different types of advertisement messages which are sent according to the structure of the device viz. advertisement from the *root devices* to get themselves identified by control points, announcement of the *Universally Unique Identifier* (UUID) for assigning message packets to respective devices, announcement of *device type* and announcement of *service type.*

     Since SSDP is UDP based, each message is sent several times in order to minimize the risk of message loss. A leasing concept has been specified for advertisements, hence each advertisement is only valid for a certain timespan and has to be renewed by a device in order to verify its existence. When a UPnP device is about to leave the network, it has to send this with a "bye-bye" message. There by all control points in the network are informed that a device and its services are no longer available.

   - **Search for Devices and Services** Additionally, control points can search actively for devices and services. Again these search messages are sent via SSDP. A control point can search for all devices, and services or root devices or a certain device with a specific UUID or service type or device type. Each UPnP device with matching criteria answers to the request with the corresponding advertisement.

2. **Description** The basic idea of UPnP is to allow ad-hoc interaction even between control points and devices which have been unaware of their mutual existence by providing adequate description of its capabilities. This description is given in XML files and contains information about the device's properties and functionalities. In order to save bandwidth, only simple information about the device and its services are published during the discovery phase. Additionally, the URLs of the XML description files are published. Every control point that is interested in the announced device can use these URLs to download the corresponding descriptions. For this purpose, each device incorporates a webserver by which the files are made accessible.

   A complete set of descriptions consists of one **Device Description** that contains general and vendor specific information about the root device, its subdevices and a list of contained services, and a separate **Service Description** for each service, which describes the available actions and the state variables available in the context of this service. Developers need to formulate these UPnP description according to the required behavior of the device and adequate support from a development environment can assist them to perform these tasks without in-depth knowledge of UPnP.

   Device Description In addition to a number of vendor specific information, the description contains four prominent entries.

   - The *device type* is normally searched for by compatible control points, e.g. a light switch would search for all light bulb devices.
   - The *device UUID* unambiguously identifies the device in a UPnP network, even if the device's IP address changes. This enables control points to identify a device, even when its location in the network changes or it repeatedly shuts down and restarts.
   - The *service list* contains information about the services offered by a device. Service type, service identifier and addresses of the service descriptions as well as the corresponding URLs for eventing and controlling are defined.
   - In case of a hierarchical device structure, the device description contains a *device list* composed of a number of subdevices. For each subdevice a complete device description with the same structure as the root device description is specified.

Service Description The service description contains a list of *actions* and *state variables* that can be accessed by a control point. An action is described by its *name* and a number of *parameters*. For each parameter a *name* and its *direction* (in/out) is specified. The *data type* of a parameter is only declared implicitly by the association of a specific *state variable* of the service. A *state variable* comprises a *name*, a *data type* and whether it is *evented* or not (see 4. Eventing).

With the device description and the corresponding service descriptions, a UPnP device is unambiguously defined. Control points only depend on these descriptions in order to determine if and how they can make use of this device.

3. **Control** Up to now, a UPnP control point is able to find a device and retrieve information about its capabilities. The next step is to utilize the device by invoking the UPnP actions, specified during the description phase. The transmission format for such an *action request* is the standardized *Simple Object Access Protocol* (SOAP) [2], which transmits messages in XML format.

   For each request, a TCP connection is created that is used to first send the action request and then transmit its response. The response can either indicate the correct execution of the action or contain an error message. For this purpose, the UPnP specification defines a set of general error messages which can be extended by the vendor with device specific error messages.

4. **Eventing** In addition to the remote execution of tasks, another important feature of UPnP is eventing of the state variables. In the *Control* phase a control point invokes actions on the device, whereas during *Eventing* a device notifies the control point of changes of certain state variables. UPnP uses the General Event Notification Architecture (GENA) [14] for this purpose.

   Within a UPnP service description, a state variable is marked with a "sendEvents" attribute, which specifies if the state variable generally supports eventing. Furthermore, a control point that is interested in changes of the state variable values needs to register itself as an observer of the specific service. When a state variable change occurs, the device will inform all registered control points.

5. **Presentation** The last feature that UPnP offers is the provision of a "Presentation Page". Each device can specify an HTML page which can be retrieved by all control points. The link to this presentation page is provided in the device description which is announced to the UPnP

network. The page e.g. may comprise vendor specific information, information about the device's state and may even provide mechanisms for controlling the device.

Evaluation

To summarize, UPnP allows devices to dynamically join a network to advertise their presence and their capabilities. It also provides ways to access remote functionalities and to be informed of remote state changes. Unfortunately, the implementation of UPnP on a peer-to-peer basis without any centralized management components comes at the cost of higher processing demands for each UPnP participant. As the exchange of messages is done via the XML-based SOAP protocol, each UPnP participant must be able to interpret and flexibly create XML messages. That is why current implementations of UPnP devices require advanced computing capabilities. The Sindrion approach aims at enabling even small, battery powered, embedded devices to join UPnP networks.

1.2.2 Sindrion

The basis of the Sindrion system is to set up a wireless link between peripheral devices and dedicated computing terminals. The objective of this connection is to source out complex data processing from the peripherals to the terminals. To this end, peripheral devices contain small smart transceivers[1], the so-called Sindrion Transceivers, which are attached to embedded sensors or actuators. Typical peripheral devices are environmental sensors, small actuators like switches, or home appliances. They contain very limited or no computing power, and the embedded sensors and actuators can be controlled by simple proprietary analog or digital control lines. These are connected to the input- and output ports (I/O ports) of the Sindrion Transceivers.

The terminal is equipped with an RF transceiver which is compatible with that included in the Sindrion transceiver. UPnP and device specific protocol processing are both done in the terminal, which features a virtually unlimited amount of processing power and memory compared to the Sindrion transceiver.

Fig. 1.2 shows the fundamental structure establishing the communication between the terminal and a previously unknown Sindrion transceiver. The procedure is as follows:

1. **Discovery**: the two end devices find each other in the discovery phase. To meet this end, the UPnP (see Fig. 1.2) discovery protocol is used . [1]

[1]The term *smart transceiver* refers to an RF transceiver with an integrated micro controller.

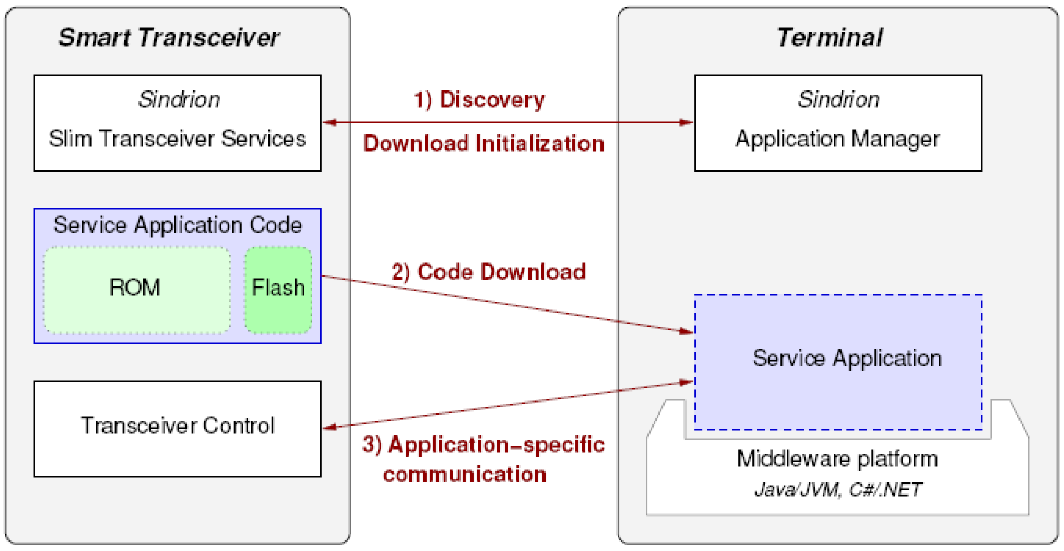


Figure 1.2: Schematic Overview of the Sindrion System

2. **Code Download**: if the terminal does not yet contain the control application for the Sindrion transceiver, the application code is downloaded by the terminal (see Fig. 1.2). Preferably, this code is written for a middleware platform such as the Java Virtual Machine. This guarantees platform independence and allows seamless integration into various terminals. Furthermore, high-level programming languages facilitate application development. The downloaded application runs at the terminal and is fully equipped with UPnP functionalities. Thereby, the service application connects via UPnP interfaces to the environment. Application-specific communication is established through the service application.

3. **Application-Specific Communication**: for the following communication between the downloaded service application on the terminal and its counterpart - the transceiver control unit - on the Sindrion transceiver a proprietary protocol called Sindrion *control protocol* is used. This protocol is transceiver hardware specific and does not depend on the application. It offers the capability to configure the transceiver and control its hardware interfaces. The service application controls the transceiver behavior according to the current application via this service protocol.

With the mechanisms described above, the peripheral devices are integrated into the UPnP network. The topology that is created by the Sindrion approach is visualized in Fig. 1.3, wherein the embedded devices (like e.g. the Sindrion Transceiver in a refrigerator) can now participate in the UPnP network because of the Sindrion enhancement.

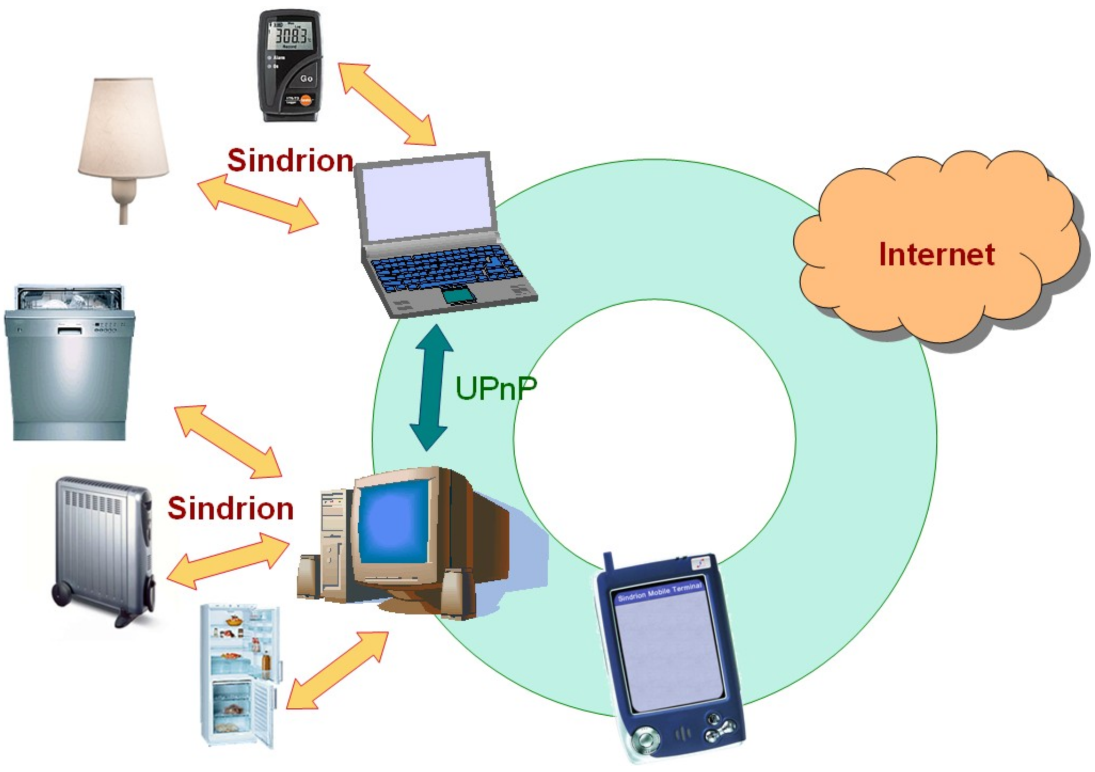


Figure 1.3: Integration of Sindrion Enhanced Devices in UPnP

In the previous sections we gained knowledge about UPnP and Sindrion. By out-sourcing the complex data processing to dedicated terminals Sindrion overcomes the lack of computational power of the embedded devices. The Sindrion applications on the terminals perform this complex data processing and are of special interest as we aim at designing support for the development of these applications. We will now take a closer look at these functional components that are required to realize the Sindrion concept. The code downloaded from the Sindrion Transceiver to a dedicated terminal included three major packages (see Figure 1.4):

1. *The Sindrion Proxy* is the central component as it provided the connected device specific functionality via semantic UPnP functionality to the network. For example, when a temperature sensor is connected to the analog line of the Sindrion Transceiver to measure the current temperature, the Sindrion Proxy interrogates the temperature sensor connected to the Sindrion Transceiver via the Sindrion Control Protocol and retrieves the data coming in at the specified analog line. The user of this temperature sensor need not know how to use the Sindrion Control Protocol, instead the Sindrion Proxy that implements a UPnP device offers a UPnP action called "GetTemperature" which internally maps the functionality provided by UPnP to Control Protocol commands. Thus, the

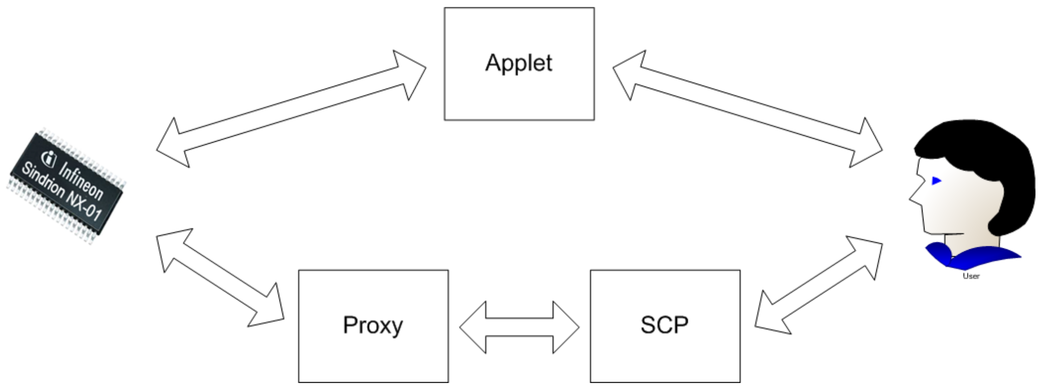


Figure 1.4: Interaction between User and Sindrion Transceiver via Sindrion Software Components

user can use comfortable semantic UPnP action and this gets translated into Sindrion specific commands by the Sindrion Proxy.

2. *The Specific Control Point* is a dedicated UPnP control point used to control the Sindrion Proxy. Its main task is to provide a Graphical User Interface (GUI) to simplify Sindrion Proxy usage. With the help of Specific Control Point the user of the temperature sensor instead of using some generic UPnP control software, can use a dedicated GUI that is stored along with the proxy on the transceiver. The manufacturer of the temperature sensor device bundles a GUI with his device which is outside the scope of the UPnP device but is important for the usability of the device and serves as a means of branding.

3. *The Sindrion Applet* can be embedded into the presentation page of the UPnP device which helps to use the functionality of the device without using a dedicated software, as a browser is sufficient for this task. On the other hand this applet has to combine the functionality of the Sindrion Proxy and the Specific Control Point as it has to provide the Sindrion Control Protocol based controller for the transceiver with a suitable GUI. The programming model provided with Sindrion intrinsically supports merging parts of the UPnP proxy and of the Specific Control Point to create a standard applet without additional programming effort.

Although the general Sindrion concept does not specify a programming language, Java is one reasonable possibility. Sindrion already comes withe a set of Java-based libraries, e.g. UPnP stack, the Sindrion Programming Model [2], or the named Control Protocol libraries. On the whole Sindrion transfers the

[2]The Sindrion Programming Model is a Java library containing interfaces and classes for sophisticated Sindrion software development. [9]

benefits of a Service-oriented architecture into the domain of embedded distributed computing.

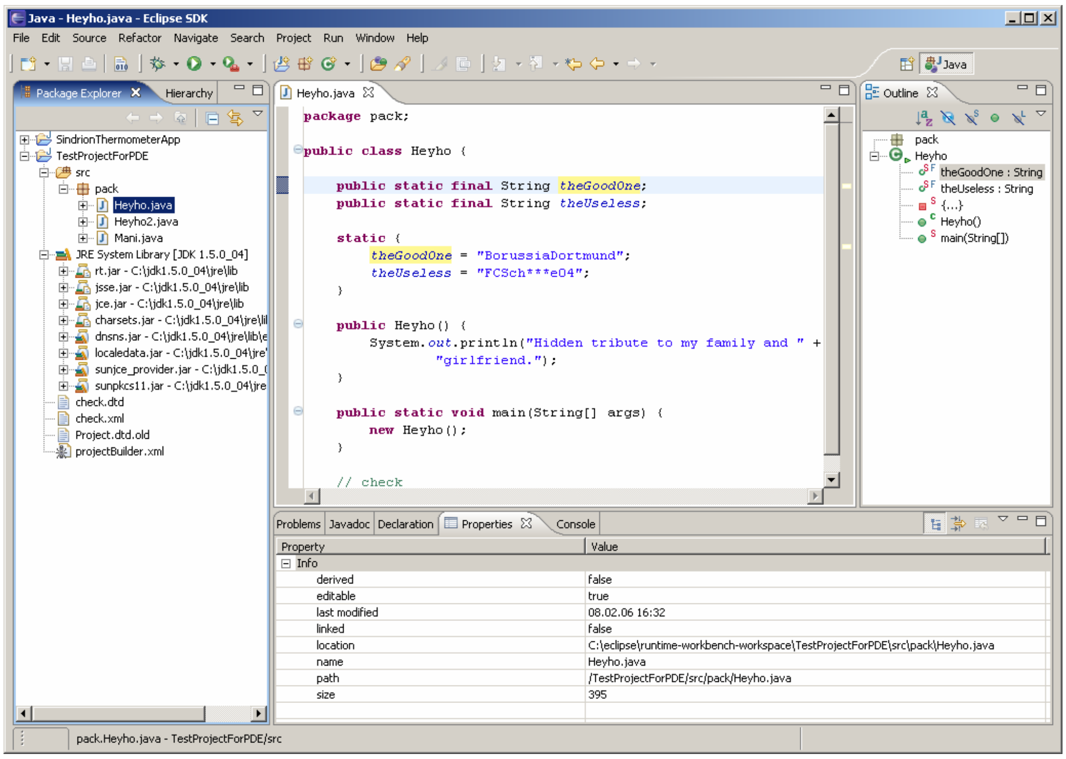
The analysis of Sindrion Application development process done in Chapter 2 helps us to ascertain the development support required. Next section gives and overview of the Eclipse platform on which the development support will be constructed.

1.3 Eclipse IDE

Eclipse, an open source community provides an extensible development platform designed for building integrated development environments (IDEs), application frameworks and arbitrary tools and software. The Eclipse Software Development Kit (SDK), is a combination of the efforts of several Eclipse projects, including Platform, Java Development Tools (JDT), and the Plug-in Development Environment (PDE). [15] Eclipse Software Development Kit is a Java integrated development environment and is used for building products based on the Eclipse Platform. This vendor-independent platform consists of a core and diverse plug-ins [16], where the core provides services for managing the plug-ins and plug-ins provide the actual functionality. This is realized by using Rich Client Platform (RCP) functionality based on the Open Services Gateway Initiative (OSGi) [17] standard framework, which strongly binds the core and the plug-ins to the Java programming language.

Eclipse represents a reliable, proven, scalable technology upon which everyone can contribute own parts enriching existing components or add all new functionality. Moreover, many companies like IBM and Texas Instruments develop commercial plug-in products for Eclipse platform. Therefore, Eclipse can be called a de-facto standard in the domain of integrated development environments that are broadly applied. The Eclipse platform written in Java also comes with plug-ins particularly facilitating Java code development, which are bundled in the so called Java Development Tooling (JDT).

The JDT contributes Java specific behavior to the generic platform, by adding Java specific integrated tools like editors, a compiler and debugger and a GUI. This way, a user is supported in every step of the Java code development. As an example, Fig. 1.5 shows the Eclipse workbench displaying the “Java Perspective”. In the center you can see the editor, in which the user types the code. On the left side, the “Package Explorer” displays resources like source files and libraries. On the right side, you can see the Java specific “Outline view”, which displays logical units of the source file like fields and methods, for a hierarchical overview. The JDT is much more powerful, it sup-


Java - Heyho.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project Run Window Help
Java
Package Explorer
Hierarchy
SindrionThermometerApp
TestProjectForPDE
src
pack
Heyho.java
Heyho2.java
Mani.java
JRE System Library [JDK 1.5.0_04]
rt.jar - C:\jdk1.5.0_04\jre\lib
jsse.jar - C:\jdk1.5.0_04\jre\lib
jce.jar - C:\jdk1.5.0_04\jre\lib
check.dtd
check.xml
Project.dtd.old
projectBuilder.xml
Heyho.java
Outline
pack
Heyho
theGoodOne : String
theUseless : String
{...}
Heyho()
main(String[])
Problems Javadoc Declaration Properties Console
pack.Heyho.java - TestProjectForPDE/src

```java
package pack;

public class Heyho {

    public static final String theGoodOne;
    public static final String theUseless;

    static {
        theGoodOne = "BorussiaDortmund";
        theUseless = "FCSch***e04";
    }

    public Heyho() {
        System.out.println("Hidden tribute to my family and " +
                "girlfriend.");
    }

    public static void main(String[] args) {
        new Heyho();
    }

    // check
```

Property	Value
Info	
derived	false
editable	true
last modified	08.02.06 16:32
linked	false
location	C:\eclipse\runtime-workbench-workspace\TestProjectForPDE\src\pack\Heyho.java
name	Heyho.java
path	/TestProjectForPDE/src/pack/Heyho.java
size	395

Figure 1.5: The Eclipse Workbench

ports code completion[3], an elaborate debugger and even the programmatical usage of all these features. This means, the user can design plug-ins for developing Java-based code by extending existing functionalities. JDT is only an example of the variety of built-in and extensible functionalities of the Eclipse platform.

The Plug-in Development Environment (PDE) provided by Eclipse is a set of tools designed to assist the Eclipse developer in developing, testing, debugging, building, and deploying Eclipse plug-ins while working inside the Eclipse workbench. Writing an Eclipse plug-in is straight forward, but not exactly trivial. The task entails creating a manifest file, writing Java source code, compiling the code into a library, testing it and packaging the plug-in into a form that is suitable for deployment. This task can be quite intricate, depending on the complexity of the plug-in and the developer's Eclipse expertise. PDE integrates itself in the workbench by providing platform contributions, such as editors, wizards, views and a launcher, which users can easily access from any perspective without interrupting their work flow. [13]

[3]If the user types in a class name, the "code assist" function shows e.g. available methods in a list for code completion.

Eclipse comes with powerful tools to develop Java software in particular but is not restricted to this programming language. It additionally permits a user to augment the platform as desired and it perfectly serves as a basis for the development environment, which will be derived throughout this thesis. The basic technologies and concepts used in this thesis have been introduced till now and the next section gives a short outlook on the scope and the structure of the thesis.

1.4 Scope of Thesis

This chapter gave an overview of the concepts used in the Sindrion project which illustrate the context of this master thesis. The goal of this thesis is build a basis for an environment which supports the development of Sindrion Applications. For this, a profound analysis of the current development process without any support from development environment is done which helps us to comprehend the difficulties that arise during the development of Sindrion Applications. Analysis of existing autonomous tools which support the development process, show that they do not bring in the desired benefit to the user and require mechanisms that help to integrate these tools. Design considerations formulated based on the analysis serve to define the problem. Considering these specifications efforts are made to design the initialization process that gives a quick start to the development process. Mechanisms are devised using which the plug-ins are able to interact and therefore the basis for Sindrion Integrated Development Environment is created. Finally, as a proof of the derived concepts, a reference implementation of the previously designed components is conducted.

Chapter 2

Analysis of Software Development Process

A profound analysis of the Sindrion Application development process is done in this chapter. The common difficulties faced by the developer, during the development process are categorized and outlined to point out the intrinsic dependencies and restrictions. Also the necessity to constructing the development environment is shown. This comprehensive analysis leads to implications for the development support required, which will be stated at the end of the chapter.

Sindrion Applications have some beneficial characteristics like programming language independence, software re-use and distributed use of open interfaces. There are also several issues to be dealt with, like the ones coining from the field of embedded software as the developed code finally has to be deployed to the target device.

Considering the support for Java software development has reached a high level of maturity, e.g. implemented in IDEs like Eclipse's JDT (see Section 1.3), we focus mainly on the requirements specifically for Sindrion Software Development. The degree of support needs to model the high expectations of typical Java developers that are e.g. used to the comfortable JDT from Eclipse IDE. The analysis aims at an ideal Sindrion design flow, although only a part of these characteristics can be implemented in this work.

The development process of Sindrion Applications mainly deals with creation of the UPnP description files and compatible Java source code for the applications. The main issues the developer has to deal with in the process of creating these applications are mentioned below:

Project set-up

Since the Sindrion applications are written in Java programming language the first step would be to create the Java project for the application and set-up the base for further development.

UPnP Specifics

The basis for UPnP development is the UPnP description files which describe the implemented device and its services. When modifying the functionality of the device, these description files have to be modified as well. There are two problems in this process: The developer needs detailed knowledge of XML language and of the UPnP description files as the files must be created according to dedicated specification conditions.

Editing Description Files

Implementing the actual functionality of the service or composition of services the developer wants to provide is also an important step. In this process, a *description file* (e.g. a WSDL[1] XML file for web services or the device and service description for UPnP (1.2.1)) of the service has to be *modified* according to the developed code and vice versa. Editing these files by hand can lead to errors that are typically expected to occur at runtime and shows misbehavior (i.e. no runtime exception) thus being hard to test and to debug. For example, a developer changes the name of a UPnP action in the UPnP service description file by hand. Now, the UPnP stack source code refers to a non-existing action as no control mechanism exists and the developer will not be informed about this problem. He will not notice the error until the code is executed.

The development of the `control instance` is tightly connected to the former created functional service. It is best practice for the developer of the service to deliver the control instance by providing a suitable GUI as well.

Software Partitioning

The development of Sindrion related software is supported by already existing libraries that provide UPnP support, implement the control protocol or even help building the Sindrion software components (Proxy, Specific Control Point, applet), by offering a library based programming model. The programmer also

[1]The Web Services Description Language (WSDL) [3] is used to define the interface of a Web service.

might rely on additional libraries, whose usage is critical in several steps of the development process. The output of the project is more than one deliverable namely proxy, Specific Control Point and standard applet which makes the use of the libraries more complex than in the case of a typical Java project. It must be made sure that all these output binaries are bundled with the necessary libraries. Effort to reduce the size of these binaries also should be taken into account as they are transferred through the network, so their size is directly related to actual or virtual communication cost (time, latency, etc.).

Additional Communication

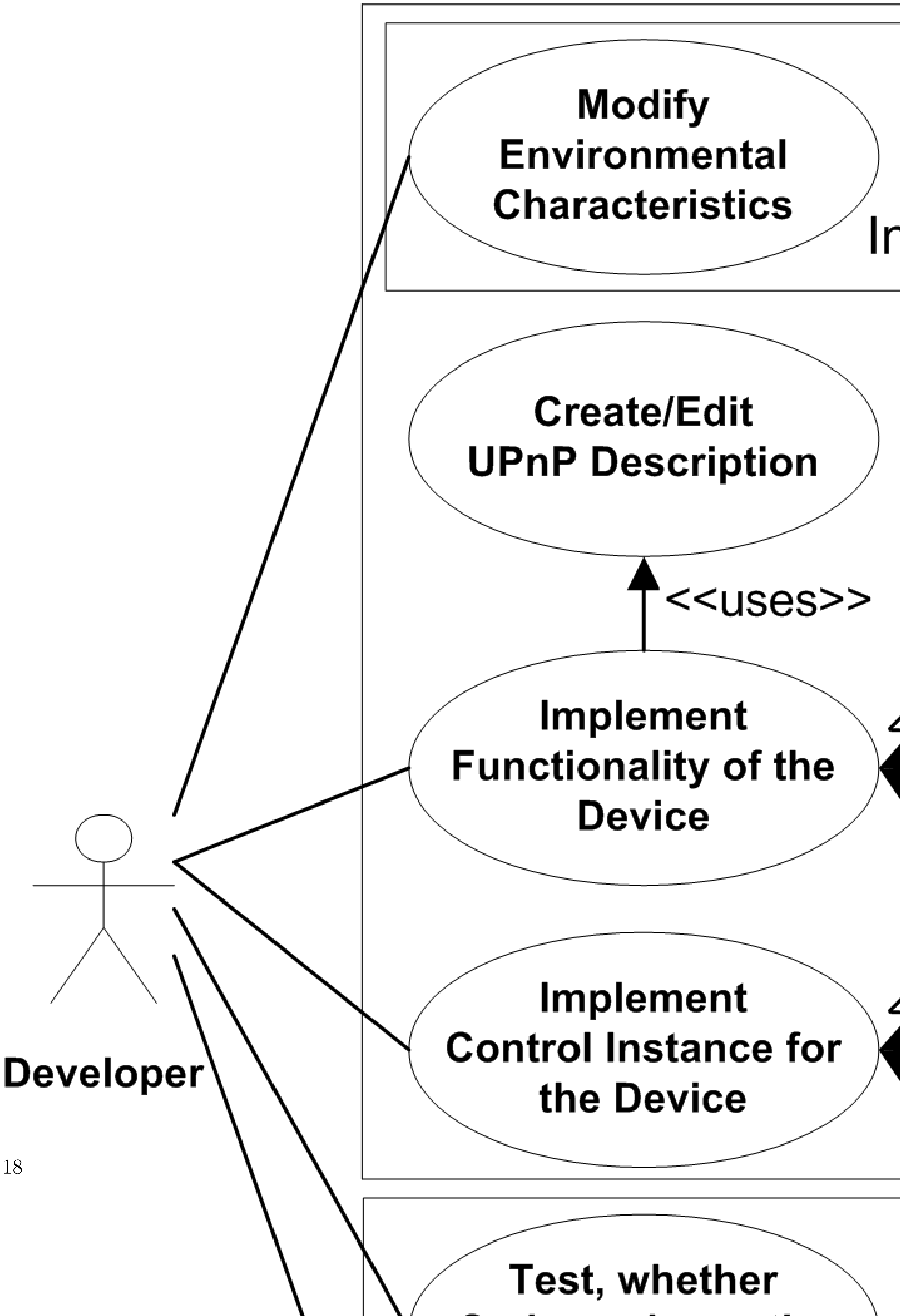
As embedded devices are integrated into the UPnP network, this 'higher layer side' of the communication has to be ensured as well. For example, a Specific Control Point must be able to find and communicate with the proxy it is designed.

The steps performed can be grouped together to four major phases: Initialization, Implementation, Testing and Deployment. Fig. 2.1 describes the use case diagram for Sindrion Development Process.

The functionality of the device (and its services) is provided by the Sindrion Proxy and the Sindrion Applet. The developer has to make sure his code is runnable on the target environment, i.e. the dedicated hardware platform of the Sindrion Transceiver. The regulations concerning the service implementation and its control instance arise as the Sindrion Proxy implements the main functionality of the SOA, the Specific Control Point provides a GUI and acts as a control instance, and the Sindrion Applet combines the functions of the aforementioned.

Modifying or setting up environmental characteristics can be thought of as an *initialization phase* but there is also a possibility of dynamically changing these characteristics. Implementing the functionality of services, customizing a description file and providing a control instance can be grouped together in the *implementation phase.*

The use case diagram allows us to group the phases, but does not contain any information about the sequence of their appearance. Naturally, the initialization is the first step in the development flow. There however exists a multitude of design flow models for remaining phases that are often aligned with company policies or specific versioning and debug tracking tools. Thus, the development environment envisaged here must assume the phases to be independent and must be able to re-iterate the phases in arbitrary order.


Modify
Environmental
Characteristics
In
Create/Edit
UPnP Description
<<uses>>
Implement
Functionality of the
Device
Implement
Control Instance for
the Device
Developer
Test, whether

This analysis helped us to understand the tasks performed by the developer during the development process of Sindrion applications without any support from tools. Some of these tasks can be automated by designing mechanisms. Apart from that the information required can be gathered from this analysis.In further proceeding of this chapter we will take a look at the development process for Sindrion Applications. Additionally the complicacy will be outlined that justifies the whole approach of the development environment in comparison to creation 'by hand'. The actions of these four phases then lead to relevant requirements for the realization of the Integrated development environment.

2.1 Phases of Software Development Process

The difficulties that arise during the development process of Sindrion Applications are sorted here into phases. Each phase describes its role in the overall development process and highlights the sequence of operations done during that particular phase.

2.1.1 Initialization

The development process of Sindrion Applications starts with the initialization phase as it serves as a basis for further development. An in-depth analysis of this process helps us to clearly know the sequence of operations performed by the developer and think of the support the development environment can offer.

Environment characteristics have to be set up as a first step as mentioned before. Sindrion applications are written in Java Programming language (uses UPnP or Web services as middleware). Each Sindrion Application is generally a separate Java project (includes Sindrion Proxy, Specific Control Point and Sindrion Applet). Setting up this project would be the first step of this phase.

The Sindrion Java project created should encapsulate all the information and sources related to the particular application in development. Firstly, it should reflect the information of the hardware platform it is developed for, e.g. Board Version, Control Protocol used for communication and the firmware it is designed for. All the information used during different steps of development process can be collected from the user and can be attached to the project. The project as well holds the Java source code files and UPnP description files based on the middleware selected for the application and the hardware platform. A good package structure allows the developer to identify the files corresponding to different parts of the application. Hence these files are hierarchically arranged in the project with respect to the corresponding Sindrion

Proxy, Specific Control Point and Sindrion Applet.

Once the project is created, selecting a dedicated target platform could implicate the usage of a corresponding library during development. In the context of Sindrion development the middleware might be a UPnP stack or Web services. These libraries have to be included into the system (i.e. attach them as resources to the contributing plug-ins or add classpath in Java case) and are necessary at compile time and runtime. This may as well influence the implementation, as the programming language could be closely linked to this middleware. In the Sindrion case the Java UPnP stack is used for instance.

After the initialization the development succeeds which is outlined in the next section.

2.1.2 Implementation

As the implementation of Sindrion software is a complex process of interdependent components relying on several sources, a range of relations between the actual source code files and sources influencing them has to be considered. Based on the analysis of these relations, the process of development can be inspected with regard to the potential of automation. In the course of this action, and underlying unified UPnP description model is presented and the development steps for the Sindrion Proxy and the Specific Control Point are outlined.

We start with the consideration of the aforementioned relations. Figure 2.2 visualizes these relations, which are explained below.

- ***UPnP Device Description*** refers to UPnP service description files (1.2.1), which define services including their actions and state variables. The actions are of interest for the developer since their implementation provides the actual functionality of the UPnP device. The dependency between device description and service description needs specific attention during creation or modification of these files. When the name of the service description file is changed or when a new service description file is created, the reference in the device description file has to be updated or added respectively. Consequently, the development environment has to keep track of UPnP service description files corresponding to a UPnP device description file.

- ***Sindrion Proxy Sources*** are influenced by the functionality of the Sindrion Transceiver and the UPnP description files. Regarding the first issue, we have seen in Section 1.2.2 that the proxy, SCP and applet are stored on the transceiver and are developed particularly for a specific

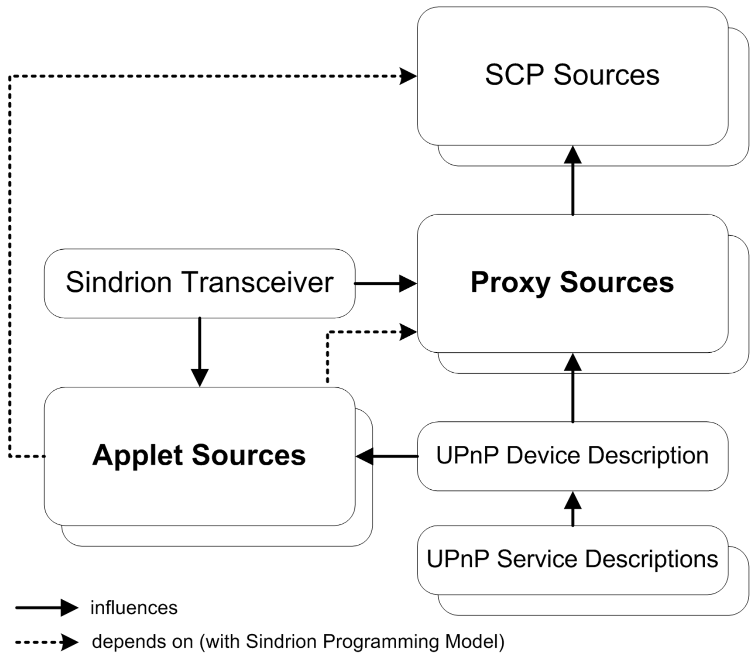


Figure 2.2: Relevant Relations of Sindrion Sources

Sindrion Transceiver. Therefore, the implementation of the binding between transceiver and proxy depends on the functionality of the control protocol implemented in the transceiver's firmware. If the transceiver's firmware changes with potential future firmware redesigns, the proxy implementation might be adapted, too.

Secondly, the UPnP description files play an important role in Sindrion Proxy development. When using a UPnP stack UPnP devices, services, actions and state variables are addressed by their identifiers specified in the corresponding description file. Therefore, the identifier in the source code has to match the one in the description file, else the UPnP functionality cannot be correctly provided by the code. During the implementation phase the correct UPnP description files have to be available for the corresponding proxy sources. Experience has shown that managing the dependencies between UPnP description and proxy code is always error-prone.

- ***Specific Control Point Sources*** are related to the Sindrion Proxy, hence with the proxy source files. A Specific Control Point is designed to control the functionality provided by typically one dedicated Sindrion Proxy, via UPnP.
- ***Sindrion Applet Sources*** are related to the device description and

> the Sindrion Transceiver similar to the proxy sources. As the Sindrion Applet does not rely on any other Sindrion component, the development environment needs to keep track of the UPnP device description and Sindrion Transceiver information as outlined for the Sindrion Proxy. Furthermore the applet additionally has to provide a GUI.

As stated above, the Sindrion Programming Model supports the creation of an applet from the SCP and proxy, so that control functionality of the proxy as well as the GUI is to a significant extent identical to the functional requirements of the applet. In this case, the relationships mentioned previously are only indirect. The applet rather depends directly on the SCP and proxy code. This is indicated by the dashed arrows in Figure 2.2.

The implementation phase contains loops, as several steps may have to be done repeatedly as corrections and adjustments of the code are an inevitable part of an usual development procedure. The UPnP device description and its service description files describe the functionality of the device, without giving any implementation details or representing the basis for any UPnP stack. Code from this stack is generated according to the description and is utilized for the Sindrion Proxy and the Sindrion Applet implementation. If these description files do not exist, they have to be created by the developer first. The creation of the Sindrion software components is tightly connected to the modification of the UPnP description files.

In next section we will consider the testing phase of Sindrion Applications.

2.1.3 Testing

All components must be executable within the IDE for testing and must be able to interact. This is of the same significance as a standard IDE with integrated debugger and compiler for pure Java development. Software testing is an essential process in the development cycle that which helps the developer to identify quality, completeness and correctness of the developed software, by verifying if the results produced are in accord with his or her expectations. The software should be able to execute and perform repeated operations yielding same results with respect to identical triggering events.

In the case of Sindrion, execution of software can get critical as there are further constraints caused by its embedded nature. Further, it has to provide the UPnP functionality on one hand and should be able to communicate with corresponding Sindrion Transceiver via Sindrion Control Protocol on the other hand. Additionally, it is tedious for the developer to deploy the software every time it needs to be tested. The ideal way of testing software that has to be deployed to a target platform is to provide a simulation environment or at least automate the deployment process and make it accessible during development.

Having verified the results from the simulated environment, it needs to be physically deployed to the Sindrion Transceiver. This phase is addressed in the next section.

2.1.4 Deployment

The final process of transferring the developed code to hardware transceiver is called deployment. It is the final step of the development process since the software deployed to the Sindrion Transceiver (hardware) cannot be altered on the hardware. If the sources are still available, they can be modified utilizing the development environment and can be deployed to the transceiver again.

Deployment involves transferring all involved source files including UPnP description files, to the hardware platform (Sindrion Transceiver). This step includes more than transferring the data. All necessary sources are to be bundled into archives (jar files) and stored on the Sindrion Transceiver's memory. For example, the sources of Sindrion Proxy, SCP and Sindrion Applet are bundled into different jars. Additionally bundles can be created for classes shared between the applications.

The next section gives a recapitulation of the analysis done and outlines how the development environment can support the developer during different phases of the development discussed in the previous sections.

2.2 Desired Support during Development

The development process of Sindrion applications has been analyzed in the previous sections. The process is subdivided into four phases that are valid for SOA compliant Sindrion with Web services or UPnP used as middleware, without regard to the utilized development environment, programming language dependent libraries or target platform specifications. This analysis is a conceptual inspection of development process performed by a developer.

Now we discuss about the potential for automation and simplification in each of the phase of development. Steps taken by the user during each phase of development is summarized and then each of these steps is examined for possible support:

Initialization Phase

The first step of this phase would be to set up the Environmental characteristics of the Development Environment. The mechanisms provided by the development environment should be easily identified and accessible by a developer. For this developer could be assisted with shortcuts to these mechanisms using special menus and tools on the menu bar and tool bar respectively with descriptive tool tips revealing its functionality.

The developer starts the process first by creating a project that serves as a container for UPnP description files and the source code files for the application. From the analysis we have inferred that it also should hold the information of the hardware, and from the user which is persistent and can be easily accessed throughout the development process. A nice package structure has to be created by the developer for partitioning the different elements of the application. The project's classpath as well has to be configured for access to the libraries it depends on during compile and runtime. An appropriate mechanism that performs these steps, viz. to configure the project and collect the information is required.

An in-depth review of the essential support from the development environment is done in the next chapter, as this phase might also perform some operations which come into picture only after a review of the complete process.

Implementation phase

After creating the necessary basis for development during the initialization phase the developer starts implementing the device i.e. writing the UPnP description files and Java source code for the device. The UPnP description files are written in XML. Although they are human readable, manually writing or editing these files is rather uncomfortable and needs knowledge of XML and UPnP concepts. A graphical representation of these files can be a significant advancement. The Java source files give the implemented functionality of the device. Analysis shows that there is strong dependency between the UPnP description files and the Java source files.

Implementation of UPnP Description Files The UPnP descriptions are split between at least one device description file and one or many service description files. Service descriptions are referenced in the device description, so a rather complex relation exists which should be supported by the development environment. The UPnP device description and all related UPnP service descriptions are to be unified into a single model. Because of the hierarchical structure of the UPnP description files, the model is hierarchical, and can be visualized in a tree data structure.

This unified tree model suits perfectly as basis for development, because the developer does not need to modify several XML files by hand but has to only work on the merged structured representation. The development environment can take advantage of this unified model and provide a GUI to the developer, thus enabling easy UPnP description modification. Using such a model, all the UPnP elements like nested devices, services, state variables, actions and arguments can be added, modified or removed by addressing the correct element. For example, an action can be added to a service or an argument can be removed from an action with simple GUI usage. Another significant advantage will be the automated control of description editing, and thus the developer is protected against writing invalid description files. For example, many identifiers should be limited in length for optimal compatibility and the development environment can automatically shorten too long names or respond with error messages.

Using this approach, the developer neither needs to know anything about XML in general, nor the UPnP device and service description files in particular. The development environment would take care of modifying the XML description files and provides a higher level of abstraction. The developer can concentrate on implementing the functionality he desires without being bothered by ‘configurational jobs’. This unified description model will benefit the Sindrion software components too, which can be seen in the next sections.

The development environment should be smart enough to identify these UPnP description files of the project as the UPnP description is split between one device description and one more or service description files. This information should be supplied to the graphical interface to build the unified model. Creation and deletion of these files also has to be assisted by development environment and it should also keep track of these files.

Sindrion Proxy and Applet Development Once the UPnP description files are created the developer has to implement the functionality of the UPnP device. This involves implementing the functionality of the actions as so-called action operators and requires read and write access to the UPnP state variables. Since the access to these elements follows certain recurring coding patterns as given by the UPnP library, the development environment should be capable of providing code snippets and insert them into the source file(s) for the Sindrion Proxy. Note that these snippets can also be used for the Sindrion Applet since the Sindrion Programming Model unifies the code of proxy and SCP and automatically generates the applet.

For providing UPnP functionality, the utilized UPnP stack has to access the UPnP description elements by the names in the XML code. Therefore the development environment has to provide a mechanism to update this identifier in the source code if corresponding identifier in the description file is altered and vice versa. This synchronization mechanism does not only react to changes on the UPnP description elements, but also to creation or deletion of these elements.

The benefit of support by the development environment increases even more in the case of complex description files, because every description element has its code representation that can be generated automatically. Note that these code snippets still have to be 'filled' with the actual implementation of the desired functionality by the developer, but the automated code template generation means a great deal of development simplification and reduction of implementation risk. In the following section the development of the third major Sindrion software component, the Specific Control Point, is explained.

Specific Control Point Development The Specific Control Point is a UPnP control point designed to provide a Graphical User Interface which controls the respective Sindrion Proxy it is designed for. Hence, it is necessary that the Specific Control Point utilizes the UPnP stack to support UPnP control functionality but additionally provide GUI elements for control. The development environment can provide GUI elements like buttons for triggering every action of the Sindrion Proxy. Here we can see the direct relation between Sindrion Proxy and Specific Control Point that can be cared about by the development environment.

We conclude that there is a strong dependency between the different components viz. UPnP description files, Sindrion proxy, Specific Control Point and Applet. Automatically keeping these sources in sync by the development environment is a highly desirable. The development environment also can use specific information that was collected during the Sindrion Proxy development to simplify Sindrion Proxy access from the Specific Control Point. This support will lead to the development of reliable code.

Once the desired functionality is implemented, we discuss how the development environment can assist the user in next development phase testing.

Testing phase

In Sindrion case, testing required interaction with the embedded Sindrion Transceiver and makes it a challenging task. We have found out that the ideal way of testing the software (deploy the version to the target platform and making it accessible) is a tedious process and this process has to be repeated if some errors are shown up during this phase. A software component is desired that emulates the hardware behavior on the development platform in the testing phase to support debugging by the developer.

A software-simulated hardware environment which seamlessly integrates into the development is highly desirable for efficiency reasons. This helps the Sindrion components to be tested inside the development environment without attaching the Sindrion Transceiver to the system. Additionally, such simulation software can be equipped with debug information outputs that actively support testing. Such a simulation environment that meets the discussed criteria is readily available. It is implemented as a tool [9] for the software development process and mimics the behavior of a Sindrion Transceiver.

This existing simulated hardware environment needs to be integrated into the development environment. The environment needs to supply the necessary information to the simulation tool by extracting it from the project information of the corresponding application. Any additional information if required can be gathered from the user during the process of its creations i.e. during the initialization process. Analysis of this tool is done in Section 2.3.2 to find out the possible support the tool requires to integrate it into the development environment.

Deployment phase

The deployment phase (last phase of the development process) should create final executable packages of the implemented and tested code, and finally should transfer these packages to the hardware platform. The development environment has to provide a way of transferring the data according to the selected platform (its specific protocols and data representation). Analysis of the deployment process gives us the different steps the user has to perform during this phase.

First the sources have to be converted into a format processed by transceiver and its platform. And the file system has to be adapted to be compatible with the hardware platform keeping in mind the hierarchical package structure of the class files. Since separate archive files are made out of the sources like Sindrion proxy, Specific Control Point and Sindrion Applet, the development environment has to categorize the files based on the respective components and supply the information.

Since the output of the project is more than one deliverable namely proxy, Specific Control Point and standard applet which makes the use of the libraries more complex than in the case of a typical Java project. It must be made sure that required libraries are as well transferred to the transceiver. Reducing the size of these binaries also should be taken into account as they are transferred through the network, so their size is directly related to actual or virtual communication cost (time, latency, etc.). Only the necessary libraries are to bundle with each component. The development environment should also categorize the libraries based on components that use them.

The Sindrion Transceiver is equipped with a flash memory device which holds all the necessary data and code it needs for its functionality. This flash memory holds the operating system, static files and dynamic files (application specific). A Flash tool that can write this data to the memory of the Sindrion Transceiver memory already exists. This tool has to be supplied with sufficient configuration information to write the appropriate files. Study of this tool is done in `Section` 2.3.3, to find possible solutions to integrate it into the development environment.

In this section we discussed about the support developer needs from the development environment during the development process of Sindrion Applications. Providing this kind of support not only helps in designing a Sindrion Integrated Development Environment, but enable the development of reliable Sindrion software components in the first place. In next section, a study of existing support for the development process is analyzed. This analysis helps us providing enough information to integrate this support into the development environment.

2.3 Analysis of Existing Support for Sindrion Application Development

In this section analysis of existing support for the development process is performed. This support is given by stand-alone tools and Eclipse Plug-ins, which can be integrated into the development environment by providing them with adequate support. By this analysis we can determine the configuration and set-up information required by these tools. The development environment should be designed to provide this information by gathering from the developer or by extracting it from existing resources.

2.3.1 Existing support for Implementation phase

The support for Implementation phase [10] of the development process is already implemented as a separate Diploma Thesis at Infineon Technologies AG. This support comprises three Eclipse Plug-ins viz. Core Plug-in, UPnP Description XML Plug-in and UPnP Java Plug-in, which help the user to develop Java based UPnP applications. Core Plug-in and UPnP Description XML Plug-in can be jointly be used by the developer to edit and write new, UPnP device and service description files by utilizing a comfortable GUI. The UPnP Java plug-in can be used to automatically generate Java source code snippets with correctly referenced identifiers from UPnP description, based on the Java UPnP stack. This automatically generated code represents a big part of the UPnP device implementation code, whereas the application-specific code that has to be manually implemented by the developer is only a small part. Thus the UPnP stack related run-time errors are reduced drastically by this approach as the interdependencies UPnP stack code andUPnP description XML code are taken care of by this plug-in.

Plug-in usage

An UPnP device description file has to be selected from the GUI menu. Then the UPnP description XML Plug-in parses the device description and the service description files referenced in the selected device description file. A model is built based on the description files which can be seen in the UPnP description view provided by the plug-in. Figure 2.3 show the UPnP description view with the model. The model can be modified using a set of actions, defined for each element of the model which can be triggered from the context menu of that particular element. The actions allow us to add or remove UPnP element viz. UPnP device, UPnP Argument, UPnP Action etc. Java source code is concurrently generated in the Java source file (the path and name of the source file is hard coded and needs to be a class of predefined package), as it listens to the UPnP model events and modifies Java code according to these events. Each model element has special properties which help in code generation and can also be edited using the Properties view. Thus the Plug-ins can assist the developer to write error free code during the implementation phase using the GUI.

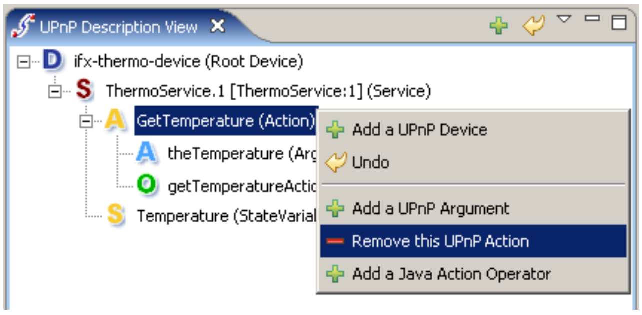


Figure 2.3: UPnP Description View

The developer using this approach needs to select the description file manually using the GUI and the Sindrion proxy and Specific Control Point file location is hard coded in the plug-in. As the source code for the applications is written in Java programming language, the Object-Oriented language approach persuades us to split the code according to the class and package hierarchy. There is no support in the present approach to handle multiple source files for filling with the code snippets. As we have discussed, there is a strong dependency between the Sindrion Proxy and Specific Control Point as they well share some classes to provide the desired functionality which arises the scope to have some common files. Additionally during the code generation the plug-in needs to access some libraries so the classpath should be updated before generating the code.

The developer can be additionally supported by the development environment by supplying this plug-ins with the information regarding the UPnP description files, Sindrion Proxy files, Specific Control Point files and common files. The development environment as well can dynamically handle the creation or deletion of the UPnP description and Java source code files based on the events. This analysis gives us a scope to integrate these plug-ins into the development environment by providing them with required data.

In next section we analyze the transceiver simulator to find out possible solutions for integration.

2.3.2 Sindrion Transceiver Simulator

During the design flow of a Sindrion System it is helpful to simulate the Transceiver at different points. The simulation should mimic the functionality of the Transceiver. The Sindrion system is split in several parts to parallelize the design process. This simulator should allow each of these task groups to be able to verify their development steps.

Sindrion Transceiver Simulator [9] has been implemented as a tool for the software development process which can play (simulate) the role of the transceiver. Its main purpose is to provide all the features of the transceiver that are used by the applications, so that transceiver and STS can be exchanged and this change is transparent to the software components that are developed. It behaves functionally equivalent to a real Sindrion transceiver, but it does not simulate the real hardware since this would not be the degree of abstraction the developers face. Its most prominent features are memory access via a web server, UPnP basic device capabilities, and interaction via the Sindrion Control Protocol. The Sindrion Transceiver Simulator as well can be connected with a simulated peripheral device. This helps to simulate on changes at its input lines. The Sindrion Transceiver Simulator comes with a GUI which shows the present status as well as the configuration information. It has already been outlined that Sindrion is a research project so Sindrion transceiver hardware as well as the Sindrion concept is under constant improvement and change. Hence Sindrion Transceiver Simulator allows easy modification and maintenance corresponding to hardware changes. Sindrion Transceiver Simulator is based on a plug-in approach to provide the flexibility to the developer. It consists of a small static core which is extensible by an arbitrary amount of plug-ins that are dynamically loaded into the STS's JVM on start up. These plug-ins are categorized into three logical groups:

- *Components* simulate mandatory parts of a transceiver.
- *Interfaces* simulate the transceiver's hardware interfaces and its firmware.
- *Functional Units* simulate peripheral devices.

Combined with the static simulator *Core* these groups forms the STS architecture. Besides the simulation of the transceiver hardware, STS architecture provide extended support (like logging, scripting and visualization of internal STS state) for the developer from *core* and *Functional Units*.

The Sindrion Transceiver Simulator can be integrated into the development environment by providing required configuration information like UUID of Basic Device, UUID of Proxy Device, udp Retake, mode, and MAC Address. Apart from the configuration details, information regarding the plug-ins (*Components, Interfaces and Functional Units*) to be dynamically activated on start-up should be also provided. This information can be gathered from the user and stored in the Java project for particular application. The environment should also support an easy access to the Sindrion Transceiver Simulator by providing a button or can be started up automatically when the developer runs the application in debug mode.

2.3.3 Flash Tool

Deployment is the final step of the development process of Sindrion Applications. We have discussed the necessary steps to be performed in `Section` 2.2. A detailed study of the Flash tool is done in this section, to accumulate the information required by the tool and the possibility for this tool to be integrated into the development environment.

The Sindrion Transceiver is equipped with a flash memory device which holds all the necessary data and code it needs for its functionality. Contents of this memory are the Sindrion Operating System as well as static and dynamic files and global data. The application specific dynamic files are Sindrion Proxy, Specific Control Point, Sindrion Applet and other files related to these components. These files have to be changed and updated frequently by the developer whenever he needs to define or change the functionality of the chip by writing or modifying his service applications. The files that define the communication via UPnP and the control protocol files are some of the static files that reside on the flash memory. These files also might be changed within the Transceivers lifetime. The developer does not deal with the Operating System unless an update is released or in the case of bugs. So the development environment does not deal with this part of memory.

The Flash Tool can be used to read, write or erase the files on the memory. A mechanism which supports the developer by providing an interface to

- edit the connection port settings to connect with the transceiver
- edit the memory settings (memory of the transceiver)
- edit the configuration and global data
- provide the information regarding the files to be written
- trigger the read, write or erase methods of the tool
- connect to and disconnect from the Sindrion Transceiver

will help the user to easily deploy the developed code. This information should also be stored in the project for further updates of the project to be written. The environment should as well be bundled with the static files and provide access for the developer to write these files. The Flash tool can thus be integrated into the development environment by supplying it with required information and providing an interface to access its functionalities.

The analysis of the existing support from Eclipse Plug-ins (Core Plug-in, UPnP description XML Plug-in and UPnP Java Plug-in), Sindrion Transceiver Simulator and Flash Tool for Implementation, Testing and Deployment phases respectively, gives an overview on the amplitude of prospects for Integration, to build a development environment that supports the developer throughout the development process of Sindrion Applications.

2.4 Summary

In this chapter the overall development process without any support for Sindrion applications is described. This analysis of this development process gives clear view of the tasks performed by the developer. Then the development process is categorized into four phases viz. initialization, implementation, testing and deployment which helps us have a clear idea of the sequence of operations performed by the user. The difficulties that arise during the development process of Sindrion Applications are sorted into four general phases a development process. Each phase describes its role in the overall development process and highlights the sequence of operations done during that particular phase. This process is followed by the analysis of existing stand-alone tools that support particular phases of development keeping in mind to integrate these components into the development environment. Since these tools are not integrated, this analysis helps us to gather the information and support required to integrate them. No technical or eclipse related information is specified here. This overall analysis of the development process and existing supports is used to formulate the design considerations in next chapter.

Chapter 3

Design Considerations

The analysis done in the previous chapter helps us to understand, the overall development process of Sindrion Applications. During the analysis we have seen problems the developer might face and later discussed the possible ways of assisting the developer during the process. The conclusions derived justify the idea of designing an Integrated Development Environment for the development of Sindrion Applications. This Integrated Development Environment should facilitate the developer in each phase of the development process.

Sections 2.2 and 2.3 put forward the key issues to be addressed by the development environment. These issues can be categorized into two types:

- the Initialization phase of Sindrion Applications is not supported by any existing tool, so the development environment should devise mechanisms to support the developer during this phase
- the Implementation phase, Testing phase and Deployment phase are supported by existing stand alone tools and Eclipse Plug-ins. These tools and Plug-ins should be integrated into the development environment.

The goal of the development environment is formulated in this chapter by refining these key issues which help us in designing an ideal Integrated Development Environment.

3.1 Designing mechanisms for support during Initialization Phase

Section 2.1.1 and Section 2.2 gives an overview of the tasks performed by the developer during the Initialization phase and possible ways to support the developer respectively. Firstly, the environment has to be set-up i.e. design means to give easy access to the developer for utilizing the facilities provided by the Integrated Development Environment. Providing the developer with special menus and tools will help him to use these functionalities. Existing support for the Implementation phase (Eclipse Plug-ins) also provide smart GUI's that help the developer during the Implementation phase for editing and writing UPnP description files. The corresponding Java code for the UPnP description files is also automatically generated. The developer needs to know the existence of this feature when he starts creating or editing UPnP description files or Java source code. These GUI's should get focused automatically when the developer attempts to create a new or edit an existing UPnP description file or corresponding Java source code.

The developer generally starts the development of Sindrion Applications by creating a new UPnP description file or by editing an already existing one. The UPnP description files and the Java source code files that belong to the Sindrion Application in development then needs to be grouped. Some information regarding this particular Sindrion Application needs to be gathered from the developer which is used for set-up or by the forth-coming phases. Writing this information by the developer to a text file is an age old process (*.ini files for many projects) which is error-prone and the developer needs to have a knowledge of all the possible configuration settings. The developers can be assisted by providing a mechanism that performs the task automatically - prompting the developer only when they must make a decision. This helps the developer to perform the task without being aware of the underlying mechanism. The developer should perform the following tasks during the Initialization phase:

- gather information for the set-up and for forth-coming phases.
- store the gathered information.
- set-up the environment
- create new UPnP description files and Java source code files in the workspace.
- group this information and the files related to the Sindrion Application.

The mechanism which guides the user throughout this process and performs the required tasks automatically is desired. It would be an add-on, if this mechanism as well creates the skeleton UPnP description files and Java source code files. A nice package-structure should be designed for the Sindrion Applications, which also could be created in the workspace by this mechanism.

3.2 Bridging the Gap between existing Tools and Plug-ins

Section 2.3 gives an analysis on the existing support for the development process. The usage of these tools becomes quite challenging, complicated and laborious task for a developer with a minimum or limited knowledge of the Sindrion or underlying technologies. Apart from the difficulties this thesis aims at designing an Integrated Development Environment that supports the user throughout the development process, from Initialization phase to Deployment Phase. These tools need to be put together and should be connected by some means which forms a major part of the Integrated Development Environment. To integrate these tools into the development environment, firstly it should be capable of supplying these tools with necessary configuration settings and required information to process their task. These tools as well use some common information and configuration settings. The shows the presence of common data that flows throughout the development process.

For example, the developer creates the UPnP description files and Java source code during the Implementation phase. The testing process needs knowledge of these files and the dependencies with other sources or libraries for testing the developed application with existing tool. The deployment process also needs this knowledge as they have to be written to the hardware platform along with required libraries. The version compatibility of the hardware and the software developed can be handled by the development environment as well. The required libraries which are set during the Initialization, and are added or updated during the Implementation also need to be transferred to the hardware platform.

The information related to the hardware platform, libraries, e.t.c. is gathered generally during the Initialization process but as we have seen, is used during the Implementation, Testing and Deployment phase. A process that handles this data during the life-cycle of the development process needs to be designed. This helps to bridge the gap between the existing supports, there-by bringing in these existing tools into the Integrated Development Environment.

3.3 Managing (Handling) Data and Related Information

The development process of the Sindrion Applications is sorted into four phases viz. Initialization, Implementation, Testing and Deployment. In this section we mainly deal with the information flow during the process of development.

Previous sections show that each of the above phases require some common information and they mainly depend on the development done in the previous section.

This shows us a need to have an underlying support plug-in which gathers data from, development environment, developer and each of the development phases. There is a possibility that this data changes during the process, so the plug-in should provide mechanisms to modify the data and as well as update itself with the data modified during the development process. For example, the developer should be able to change the libraries or version number during the development process. Apart from that the plug-in should also update itself with the changes in the workspace resource that belong to the current application. Developer while writing or editing UPnP description need to create corresponding description files in the workspace. The Java source code developed may be divided into packages and class files based on the services and actions of the device. So there is a need to keep track of the resource changes by this plug-in.

Since this data is used throughout the development process different components of the development environment try to access this data periodically based on their needs. Hence, the modeling of this data is also a key issue as it directly affects the performance of the Integrated Development Environment.

3.4 Communication between Components

In previous section we have discussed that the plug-in handling data should keep track of the resource changes in the workspace. Apart from keeping a track of the resource it also needs to communicate with other components of the development environment when a resource is changed, created or deleted. We have also discussed that the Java source code for Sindrion Applications can be generated automatically using existing support. Since the UPnP descriptions and the Java source code need to be kept synchronous the plug-ins that perform the respective tasks also need to communicate to each other and as well communicate with the underlying data plug-in to identify the resources in the workspace.

The analysis done helps us to find the difficulties faced by the developer in the development process, which in turn should be supported by the Integrated Development Environment. In this chapter we mainly derived considerations (3.5) to be taken care of while designing the plug-ins. These considerations serve as a base to design a perfect Integrated Development Environment.

3.5 Design Considerations

This section outlines the key issues which aid us during the design, to achieve the goal of developing an ideal Integrated Development Environment.

- Set-up the Environment of IDE.
- Project Configuration and Creation.
- Acquisition and Storage of Project Information.
- Data Modeling.
- Data Access Mechanisms.
- Track Resource Changes.
- Communication and Integration of the Plug-ins.

3.6 Summary

The design process of a software system should be first started by understanding the customer's (Sindrion Application developer's) needs. This is done generally by analyzing the requirements (functional and non-functional), which in turn define the scope the project. Based on this analysis design objectives (specifications or considerations) have to be formulated to satisfy the set of attributes derived from the analysis. This defines the problem, which should be solved, to satisfy the needs.

In Section 2.2 we discussed the desired support for the developer during the overall development process and Section 2.3 gives the existing support for the development process. In this chapter we have a big picture of the Integrated Development Environment which assists the user throughout the development process. This helps us to identify the gaps between existing tools and mechanisms, and coin the specifications for the design process.

Firstly,a mechanism that assists the user to perform the tasks of the initalization phase has to be designed. Then we identified the gaps between the existing support which should be filled to realize the Integrated Development Environment, and helps the user to develop the Sindrion Applications with minimal knowledge of underlying technologies (Sindrion and UPnP). To perform the assigned task the plug-ins and tools that comprise the Integrated Development Environment also need to share some information and pass on the information to the forth-coming phases. Different implications during this communication process are discussed. This chapter mainly formulates the design considerations (objectives or specifications) based on which the design of the Integrated Development Environment is done in Chapter (5).

Chapter 4

Eclipse Frameworks and Mechanisms

In Chapter 2 development process of Sindrion Applications was analyzed which defined the scope of the current project, and the design considerations were conceived based on this analysis. To realize the IDE which supports the developer by satisfy these considerations, it should utilize the underlying technologies and frameworks provided by Eclipse.

The short introduction of the Eclipse IDE in Section 1.3 pointed out that it is an open source, extensible platform for tool integration and development. This chapter gives details about the Eclipse platform, its concepts regarding plug-ins, the interfaces for plug-in communication, and data exchange and various aspects and functionalities of plug-in development which help us to design and bring in the IDE. Firstly, the plug-in concept is presented regarding to flexibility and extensibility of the platform. Then crucial interface mechanisms, so-called extension points are displayed according to communication and interaction of platform concepts.

4.1 Plug-in Concept

As mentioned before, the Eclipse platform is structured as a core runtime engine and a set of additional features that are provided as plug-ins [8]. Plug-ins add functions to the platform by contributing to pre-defined interfaces, so-called extension points, which will be described in detail in Sec. 4.3. For instance, the workbench user interface (UI) is provided by one such plug-in. The Java functionalities in Eclipse IDE are not activated when the workbench starts. Instead a platform runtime is activated which can dynamically discover registered plug-ins and activate them as needed.

When a developer wants to provide code that extends the platform, he does this by defining system extensions in his plug-in. The platform offers a well-defined set of extension points - ‘slots’ where you can hook into the platform and contribute system behavior. So the plug-in created by a developer is no different than basic plug-ins, like e.g. the workbench UI, from the platform’s point of view. In Eclipse a plug-in is an object which can be configured into the system at system deployment time and which provides any kind of service within the Eclipse environment. This is the basic assumption for contributing own components to the platform, thus realizing a tailor-made solution of a development environment. In addition to this the infrastructure supports activation and operation of a set of collaborating plug-ins in order to enable a seamless development environment integrating every required step. Within a running Eclipse system, a plug-in is represented by an instance of a plug-in runtime class, which must extend the abstract class “org.eclipse.core.runtime.-Plugin” for generic plug-in management capabilities.

The deployment of such a plug-in is performed by copying all necessary resources, including the plug-in runtime class and required libraries, into an Eclipse plug-in folder. This plug-in can be activated by the Eclipse runtime when needed, i.e. if some part of the plug-in is required to achieve some function, e.g. a button is pressed in order to perform a dedicated action. Activation of a plug-in in this matter means instantiating and initializing its runtime class. The plug-in runtime class has to do particular processing when the plug-in is activated or deactivated, for example allocate or release resources. In that case the activation and deactivation methods (*start*() and *stop*(), respectively) inherited from the class “org.eclipse.core.runtime.Plugin” can be overridden. If the designer of a plug-in does not need specific activation or deactivation behavior because of the simplicity of plug-in, a default class automatically provided by Eclipse can be used.

Besides the plug-in management kernel, also referred to as Eclipse runtime or Eclipse platform, special core plug-ins are available in every Eclipse environment and therefore they are activated in each instance of Eclipse. The more interesting case of non-core plug-in activation will be pointed out now.

4.2 Dependencies vs. Extensions

There are the two kinds of relationships from one plug-in to another:

- **Dependency** This relationship is characterized by the two roles *dependent plug-in* and *prerequisite plug-in* . A dependent plug-in utilizes functions of the prerequisite plug-in.

- **Extension** The two roles which are important in this relationship are *host plug-in* and *extending plug-in* . The functions of a host plug-in are extended by the extending plug-in.

A deployed non-core plug-in may be activated in a running Eclipse environment if it is directly or indirectly related to a core Eclipse plug-in by the dependency or the extension relation. So if a deployed plug-in is not related to a core plug-in via any relationship, it is not available in Eclipse scope of accessible plug-ins. However, even a related plug-in could remain inactive if no triggering event, e.g. a user action, occurs. This startup behavior is called lazy instantiation and prevents unnecessary waste of resources and processing time by plug-ins which are not actually needed. Next, these relations of (non-core) plug-ins are looked at more closely in order to understand the structure of the Eclipse platform including its plug-ins.

Dependency

The dependency of a plug-in to another is described in its manifest file, where every prerequisite plug-in has to be listed. This dependency comes into effect in two ways:

1. At compile time of the plug-in: Eclipse extends the class path of a dependent plug-in with the sources of all its prerequisite plug-ins. This is crucial for a plug-in developer as the dependent plug-in cannot be compiled without the sources of all prerequisite plug-ins.

2. At runtime: Eclipse takes care for the availability of a prerequisite plug-in to a dependent plug-in if the latter is activated. This means that the prerequisite plug-in is activated if it has not been activated yet.

Extension

The operation of equipping a plug-in with any number of processing elements, namely callback objects, is called extension. This operation is very general: any plug-in, i.e. the host plug-in, may allow other plug-ins to extend it by adding callback objects. The extending plug-in defines the extension and causes a changed behavior of the host plug-in. Normally, this means that processing elements are added to the host plug-in and are customized by the extending plug-in. For example, the GUI is managed by a host plug-in and an extending plug-in can contribute own buttons to the toolbar customizing them with specialized actions (see Fig. 4.1).

A single extension can add one or multiple *callback objects* to the platform, which enable communication between the host and the extending plug-in. These callback objects are plain Java objects, created and supervised by the provider and are not automatically managed by Eclipse. Note that although the concept of the callback object is general enough to allow extending plug-ins to supply their own customized objects, in simpler cases callback objects supplied by the host plug-in may be used by the extending plug-in as well, even allowing a parameterization of these objects, what may be sufficient.

Furthermore, the extension concept itself does not dictate that the host plug-in reveals every detail of its extension in its interface. For example, a host plug-in may allow extending plug-ins that act as listeners to be informed of special events. But since they are only listeners these extending plug-ins do not have the possibility to change the behavior of the host plug-in, as it would be in the former mentioned GUI example.

As a plug-in may be augmented with different kinds of extensions, each extension must be specified by a unique set of configurational and behavioral requirements. Thus, an extensible plug-in offers different types of 'slots' that one or more extensions can plug into. These special slot types are called extension points, which will be outlined in the next section.

In Figure 4.1 the cooperation of the participants of an extension is shown. Here, an XML Editor plug-in extends the extension of the Eclipse workbench by toolbar button items. In this extension, the host plug-in is the Eclipse workbench user interface, "org.eclipse.ui", whose toolbar can be extended via an extension point called "ActionSets". The extending plug-in is the XML Editor plug-in, "com.objfac.xml". For presenting the toolbar buttons to the user, the XML plug-in uses the "ActionSets" extension point to extend the workbench UI plug-in by particular toolbar buttons, namely "New XML File" and "New DTD File". Note that the extending plug-in defines the extension, which augments the workbench UI by multiple items. What we also see in the figure as well are the classes of the extension's callback objects. These classes ("OpenXMLWizard" and "OpenDTDWizard") represent actions, which are executed when the corresponding button is pressed, i.e. a certain wizard is opened in both cases.

So far we have learned that any kind of functionality can be contributed to the runtime platform of Eclipse by plug-ins, which may be related by dependency or extension. Especially the latter makes it necessary that interfaces for communication and data exchange between plug-ins exist. In the following section we will now take a closer look at the extension points.

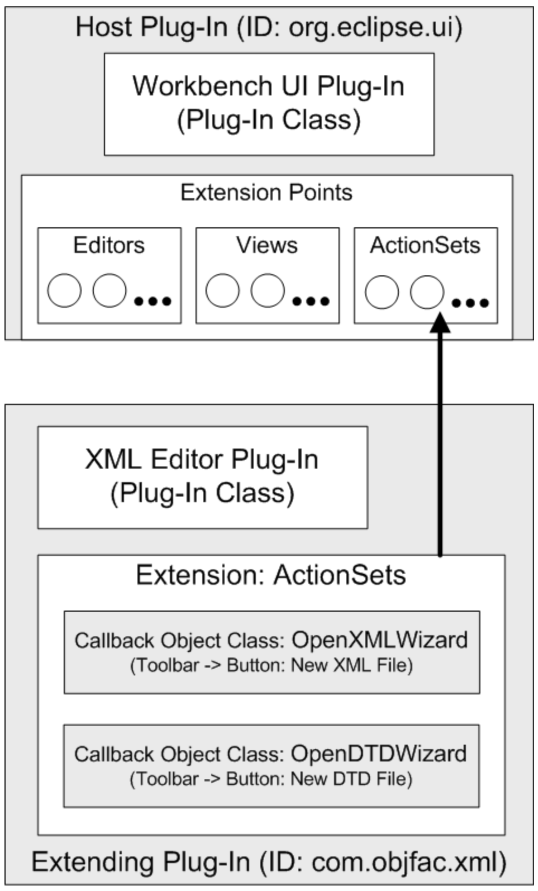

Host Plug-In (ID: org.eclipse.ui)
Workbench UI Plug-In (Plug-In Class)
Extension Points
Editors
Views
ActionSets
XML Editor Plug-In (Plug-In Class)
Extension: ActionSets
Callback Object Class: OpenXMLWizard
(Toolbar -> Button: New XML File)
Callback Object Class: OpenDTDWizard
(Toolbar -> Button: New DTD File)
Extending Plug-In (ID: com.objfac.xml)

Figure 4.1: Participants of a Plug-in Extension. The XML Editor Plug-in extends the Workbench UI Plug-in via an `ActionSet` Extension that adds Buttons to the Toolbar.

4.3 Extension Points

In this section the extension mechanism is presented, which enables a modified behavior of a host plug-in according to the extension declared by an extending plug-in. But first, the components will be illustrated by giving an example:

The Host Plug-in

A plug-in that acts in the host role defines the extension point and is extended. In addition to providing some functionality by its own, this plug-in also coordinates and controls all of "its" extensions, meaning extensions that extend its extension point. Within the host plug-in's Eclipse XML file, an extension point is declared in a corresponding XML element. Here is an example of such a declaration:

Listing 4.1: A plug-in XML File of an Eclipse Host Plug-in

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.0"?>
3  <plugin>
4     <extension-point
5       id="HostExtensionExample"
6       name="Host␣Extension␣Example"
7       schema="schema/ExtensionExample.exsd"
8     />
9  </plugin>
```

Crucial information included in this file can be found in line 5 and 7 in listing 4.1. In line 5, the ID of the extension point is specified. Internally, it will be composed of the plug-in's ID and the ID given here in order to generate a fully-qualified unique ID within the Eclipse platform. This is important since every extending plug-in has to use this ID, e.g. `HostPlugin.HostExtensionExample`. In line 7, a schema file is referenced that defines the extension point. Extending plug-ins have to use this extension point by specifying XML elements in their particular plug-in definition file (See Listing 4.2) according to the schema file. The schema file defines the configuration syntax for extensions to that extension point.

Next, the extending plug-in role is presented that has to be compatible with the schema specification outlined before.

The Extending Plug-in

A plug-in that acts in the extending role defines the extension typically making certain aspects of itself available to a host plug-in through the extension and in addition causing the host plug-in to add certain processing elements, so-called callback objects, to its environment. An extension is declared by using an extension XML element in the extending plug-in's manifest file corresponding to the schema file defining the host plug-in's extension point. Listing 4.1 contains an example of an extending plug-in XML file that extends the `HostExtensionExample` extension point of listing 4.1 by specifying a name and Java class (see lines 9 and 10).

Listing 4.2: A plug-in XML File of an Eclipse Extending Plug-in

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?eclipse version="3.0"?>
3  <plugin>
4     <extension
5       id="ExtendingExtensionExample"
6       name="Extending␣Extension␣Example"
7       point="HostPlugin.HostExtensionExample">
8       <ExtensionElement
9         class="ExtendingPlugin.ExtendingExtension"
10        name="The␣new␣Extending␣Extension"
```

```
      />
    </extension>
</plugin>
```

Note that the fully-qualified name of the extension point is given in line 7. The callback objects that are significant for plug-in interoperability, are shown next.

The Callback Object

In the context of a particular extension, an object that acts in a callback role is a plain Java object (not being part of an Eclipse plug-in) that is called by the host plug-in when certain events specified in the corresponding extension point contract are recognized by the host plug-in, e.g. the extending plug-in has been detected. This callback object is used for inter-plug-in communication and data exchange. The interface for callback objects is provided by the host plug-in and is explained in the documentation of the extension point being extended. The implementation of callback objects is typically a custom class that is specific to the particular extension and is furnished by the provider of the extending plug-in. Because the implementation of the callback object in the extending plug-in references to the callback interface, which is typically packaged with the host plug-in, an extending plug-in typically also depends on the host plug-in.

As the whole extension mechanism is very flexible, it is up to the creator of the extension point how callback objects are utilized. For example, the XML attribute “class” references to an actual Java class that can be instantiated, while the “name” attribute is used for description only. Due to this freedom, the documentation of an extension point is relevant for the creator of an extending plug-in.

For example, the host plug-in from our example may define an interface, e.g. `ICallbackObject`, for the callback object that makes sure a method called *execute*() exists. Now this host plug-in could instantiate the class `ExtendingExtension` of the extending plug-in when needed, as it is specified in the extending plug-in XML file (see line 9 in listing 4.2) and call the `execute` method of this class after instantiation. This way the extending plug-in contributes an object according to the host plug-in's interface, whereas the host plug-in has full control over this object.

4.4 Start-up and Initialization

The principle of lazy plug-in activation is very important in a platform with an open-ended set of plug-ins.

Managing the dependencies is a large part of building an Eclipse application. An extension point is an available interconnection endpoint that other plug-ins may use to provide added functionalities (extensions in Eclipse terms) and this mechanism plays an important role in lazy activation. Plug-ins are self-describing and explicitly list the other plug-ins or functions that must be present for them to operate. The RuntimeŠs job is to resolve these dependencies and knit the plug-ins together. ItŠs interesting to note that these interdependencies are not there because of Eclipse, but because they are implicit in the code and structure of the plug-ins. Eclipse allows you to make the dependencies explicit and thus manage them effectively.

A Plug-in should not get activated or started when workbench starts, but gets started only when the user needs it. This is as well to follow often sited UI design rule, the screen belong to the user. Therefore a program should not make changes on the screen that the user did not somehow initiate and this makes users feel that they are in control of what is happening builds their confidence in the UI and results in a much pleasanter user experience. This rule is followed by the Eclipse UI, but the underlying principle has been applied to a much broader scope. In Eclipse, one of the goals is to have the screen, the CPU, and the memory footprint belong to the user; that is, the CPU should not be doing things the user did not ask it to do, and memory should not be bloated with functions that the user may never need. This principle is enforced in the Eclipse Platform through lazy plug-in activation. Plug-ins are activated only when their functionality has been explicitly invoked by the user. In theory, this results in a relatively small start-up time and a memory footprint that starts small and grows only as the user begins to invoke more and more functionality. [19]

The behavior of each plug-in is in code, yet the dependencies and services of a plug-in are declared in a special XML file named `plugin.xml`. Hence, each plug-in can be viewed as having a declarative section and a code section.This structure facilitates lazy-loading of plug-in code on an as-needed basis, thus reducing both the startup time and the memory footprint of Eclipse. On startup, the plug-in loader scans the `plugin.xml` file for each plug-in and builds a structure containing this information and so is the file always available, regardless of whether a plug-in has started. This allows the platform to present a plug-inŠs functionality to the user without going through the expense of loading and activating the code segment. Thus, a plug-in can contribute menus, actions, icons, editors, and so on, without ever being loaded. If the user tries to run an action or open a UI element associated with that plug-in, only then will the code for that plug-in be loaded. This structure takes up some memory, but it allows the loader to find a required plug-in much more quickly and takes up a lot less space than loading all the code from all the plug-ins all th time. [8]

To get down to specifics, a plug-in can be activated in three ways. [19]

- If a plug-in contributes an executable extension, another plug-in may run it, causing the plug-in to be automatically loaded.
- If a plug-in exports one of its libraries (JAR files), another plug-in can reference and instantiate its classes directly. Loading a class belonging to a plug-in, cause it to be started automatically.
- Finally, a plug-in can be activated explicitly, using the API method `Platform.getPlugin()`. This method returns a fully initialized plug-in instance.

If the plug-in isnŠt meant for wider use and belongs only to the particular IDE, it can use the org.eclipse.ui.startup extension point to activate your plug-in as soon as the workbench starts up. The startup extension point allows you to specify a class that implements the IStartup interface. If you omit the class attribute from the extension, your Plugin subclass will be used and therefore must implement IStartup. This class will be loaded in a background thread after the workbench starts, and its earlyStartup method will be run. As always, however, your Plugin class will be loaded first, and its startup method will be called before any other classes are loaded. The earlyStartup method essentially lets you distinguish eager activation from normal plug-in activation. [20]

Note that even when this extension point is used, the user can always veto the eager activation from the `Workbench > Startup` preference page. This illustrates the general Eclipse principle that the user is always the final arbiter when conflicting demands on the platform are made. This also means that you canŠt rely on eager activation in a production environment. You will always need a fall back strategy when the user decides that your plug-in isnŠt as important as you thought it was. [20].

4.5 Adapters and Adapter Factories

The Eclipse Platform Runtime provides a mechanism for extending objects dynamically. Adapter framework is used translating one type of object into a corresponding object of another type. This allows for new types of objects to be systematically translated into existing types of objects already known to

Eclipse. When a user selects elements in one view or editor, other views can request adapted objects from those selected objects implementing the "org.-eclipse.core.runtime.IAdaptable" interface.

The objects must implement the `IAdaptable` interface to participate in the adapter framework. The `IAdaptable` interface contains a single method for translating one type of object into another:

`getAdaptter(Class)` - Returns an object that is an instance of the given class and is associated with this object. Returns `null`, if no such object can be provided.

Implementers of the `IAdaptable` interface attempt to provide an object of the specified type. If they cannot translate themselves, then they call the adapter manager to see if a factory exists for translating them into the specified type.

Listing 4.3: Example `getAdapter()` method implementation.

```java
public Object getAdapter(Class adapter) {
   if(adapter.isInstance(resource))
      return resource;
   return Platform.getAdapterManager()
                    .getAdapter(this, adapter);
 }
```

Mechanisms desiring to translate an object passes the desired type, such as `IResource.class`, into the `getAdapter(...)` method, and either obtains an instance of `IResource` corresponding to the original object or null indicating that such a translation is not possible.

Listing 4.4: Using adapters.

```java
if (!(object instanceof IAdaptable))
   return;
MyInterface myObject = ((IAdaptable)object)
                          .getAdapter(MyInterface.class);
if (myObject == null)
   return;
....do stuff with myObject.....
```

Implementer of "org.eclipse.core.runtime.IAdapterFactory" interface translates the existing types into new types. For example and adapter factory can be implemented to translates `IResource` into `IMyItem`. The `getAdapterList()` method returns an array indicating the types to which this factory can translate, while the `getAdapter(...)` method performs the translation.

Listing 4.5: Adapter factory.

```
public class MyAdapterFactory implements IAdapterFactory {
   private static Class[] SUPPORTED_TYPES = new Class[] {
      IMyItem.class};
   public Class getAdapterList() {
      return SUPPORTED_TYPES;
    }
   public Object getAdapter(Object object, Class key) {
      if (IMyItem.class.equals(key)) {
         MyManager mgr = MyManager.getManager();
         IMyItem item = mgr.existingFavoriteFor(object);
         if (item == null)
                                item = mgr.newFavoriteFor(
                                   object);
         return item;
       }
      return null;
    }
 }
```

Adapter factories must be registered with the adapter manager before they are used. Typically, a plug-in registers adapters with adapter managers when it starts up and unregisters them when it shuts down. The following code registers the adapter with IResource.class as the argument indicating that the adapter factory can translate from this type to others.

Listing 4.6: Registering adapters

```
myAdapterFactory = new MyAdapterFactory();
IAdapterManager mgr = Platform.getAdapterManager();
mgr.registerAdapters(myAdapterFactory, IResource.class);
```

The plug-in's stop() method must be modified to unregister the adapter

Listing 4.7: Unregistering adapters

```
Platform.getAdapterManager()
            .unregisterAdapters(myAdapterFactory);
myAdapterFactory = null;
```

Alternatively, the adapter factories can be registered declaratively using the "org.eclipse.core.runtime.adapters" extension point. Factories registered with this extension point will not be able to provide adapters until their corresponding plug-in has been activated. The adapters extension point allows plug-ins to declaratively register adapter factories. This information is used to by the runtime XML expression language to determine existence of adapters without causing plug-ins to be loaded. Registration of adapter factories via extension point eliminates the need to manually register adapter factories when a plug-in starts up.

The example in Listing 4.8 declares that this plug-in will provide an adapter factory that will adapt objects of type IFile to objects of type MyFile.

Listing 4.8: The "org.eclipse.core.runtime.adapters" extension declaration.

```
<extension point="org.eclipse.core.runtime.adapters">
   <factory
      class="com.xyz.MyFileAdapterFactory"
      adaptableType="org.eclipse.core.resources.IFile">
     <adapter type="com.xyz.MyFile"/>
   </factory>
</extension>
```

Adapter factories registered using this extension point can be queried using the method IAdapterManager.hasAdapter, or retrieved using one of the getAdapter methods on IAdapterFactory. An adapter factory registered with this extension point does not need to be registered at runtime using IAdapterFactory.registerAdapters.

4.6 Summary

In this chapter, the possibilities of extending the Eclipse platform by plugins have been outlined. These plug-ins can communicate and exchange data using dedicated interfaces, the so-called extension points. The approach of extension is a quite general concept in Eclipse and to comprehend its full generality it is helpful to summarize the types of relationships that may occur between plug-in objects, extension points, and callback objects. A plug-in may act both as a host plug-in containing multiple extension points, and as an extending plug-in. Then an overview of the eclipse mechanisms and extendable frameworks which we are interested in for design process are discussed emphasizing their benefits.

Chapter 5

Design

All design activities interact. A good software design process recognizes these interactions between the design activities and allows the design to change; sometimes radically, as various design steps reveal the need. [11]

This chapter deals with the design of development support for Sindrion applications based on the design considerations formulated in the Section 3.5. Firstly, we design the Initialization phase of the development and derive mechanisms which would help the developer to accomplish the tasks of this phase with minimum effort. With this support for the Initialization phase, each of the development phases has an individual support from the development environment or from a stand-alone tool. Next step of the design process is to unite these plug-ins to form the basis an ideal Integrated Development Environment. For this purpose, first the modeling of Sindrion project has to be done based on the analysis of the common data between the phases, done in Chapter 2. Depending on the usage of this common data by the components of the IDE the access mechanisms have to be derived. The communication between the components of the IDE is also a key issue, which can be designed only after figuring out the data that needs to be shared between the plug-ins and deriving the access mechanisms for the datamodel. Once the data is gathered the mechanisms to make the data persistent between the sessions have to be formulated.

Apart from the considerations outlined in Section 3.5 the problems that arise during the course of design are as well handled, to realize an Integrated Development Environment that supports the user throughout the development process of Sindrion Applications.

5.1 Design of Initialization phase

This phase mainly has to deal with the support for the developer and support required for forth coming phases of the development process. During the initialization phase the first task of the developer would be to create a project. A Sindrion project should encapsulate the UPnP description files and Java source code for a particular Sindrion Application. The UPnP description files are the device and service description files that describe the device and services offered by the device. The Java source code corresponds to the Sindrion Proxy, SCP and Sindrion Applet. The Eclipse plug-ins are generally designed for a particular set of projects, for example the JDT can support only the Java projects but not the C projects. As the development process of Sindrion Applications include mainly XML descriptions and Java code, the project will be basically a Java project with additional Sindrion specific information (XML data does not need any compilation and Java code needs to be compiled). Hence, the projects need to be configured so that they can be distinguished from pure Java projects and can be identified by respective plug-ins. The developers also need to associate the libraries that support the development of these applications.

Firstly, we design mechanisms to identify the project and then mechanisms that configure the project with Sindrion and Java specific information. As we have discussed, the project contains XML files and Java source files that correspond to three different components of a Sindrion application. So we also need to focus on the organization of these files in the project.

5.1.1 Identification of a Sindrion project

This section aims at providing mechanisms, which allow the plug-ins providing Java and Sindrion specific functionality to identify the Sindrion projects. A Java project contains source code and related files for building a Java program. It has an associated Java builder that can incrementally compile Java source files as they are changed. A Java project also maintains a model of its contents. This model includes information about the type hierarchy, references and declarations of Java elements. This information is constantly updated as the user changes the Java source code. The Java builder builds Java programs using a compiler that implements the Java Language Specification. The Java builder can build programs incrementally as individual Java files are saved. Problems detected by the compiler are classified as either warnings or errors. [12] The Sindrion project will be a wrapper for a basic Java project as it envelopes the Sindrion project related information along with the Java project. The Java projects in the workspace are identified using Java Project Nature.

Nature

Project natures act as tags on a project to indicate that a certain tool is used to operate on that project. They can also be used to distinguish projects that a particular plug-in is interested in, from the rest of the projects in the workspace.

The "org.eclipse.jdt.core.javanature" is added to the project description[1] of the projects in the workspace to be identified as Java projects. The Plug-in projects in the workspace are identified by "org.eclipse.pde.PluginNature". When a nature is added to a project for the first time, the nature's configure method is called. When the nature is removed from the project, the deconfigure method is called. The natures lifecycle methods *configure* and *deconfigure* can be used to associate or disassociate additional attributes, such as builders, with the project. For example, the "org.eclipse.jdt.core.javanature" adds the Java builder to the Java projects in the workspace and and "org.eclipse.pde.-PluginNature" adds the Manifest and Schema builders to the plug-in projects.

The Sindrion projects in the workspace need to be associated with a Sindrion specific nature tag, to get identified by the plug-ins interested in Sindrion Projects. Hence, the 'sindrionNature' is be defined and associated with the Sindrion projects in the workspace, to manage their association with Sindrion specific plug-ins. The configuration mechanism, as well can be used to associate Sindrion project builders that may come-up during the design process.

Since the Sindrion project envelopes a basic Java project and Sindrion specific project details, 'sindrionNature' should have the "org.eclipse.jdt.core.-javanature" as a 'requires-nature'[2] constraint.

5.1.2 Managing Project Information

The Sindrion Applications are developed specifically for a specific end device, and also depend on the platform version of the Sindrion Transceiver they are developed for (as the Sindrion research project is currently in development phase hardware and the operating system of the transceiver may vary to enhance the behavior). This information is utilized by different phases of development process, for example the libraries associated with project will depend on the version, the transceiver simulator should behave according to the transceiver platform the applications are designed for and the deployment process needs to transfer these applications to specific area of the transceiver's memory. Since this data is used by different phases of the development process

[1]Project description contains the meta-data required to define a project.

[2]The 'requires-nature' constraint specifies a dependency on another nature. When a nature is added to a project, all required natures must also be added.

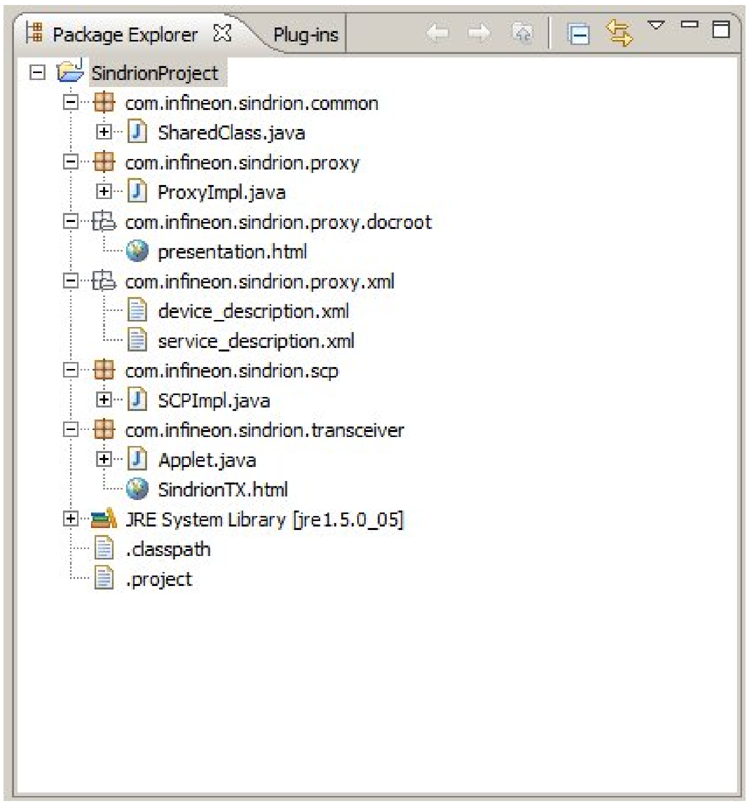
it should be attached to the project rather than the plug-in. This information also needs to be cross session persistent.

Information is gathered from the user while creating the Sindrion project and can be stored in a text file in a project. But the development environment cannot react when this data is changed without its knowledge (modification of these files outside the development environment is possible). Upon a change of this information the development environment should suggest if this change is compatible or could lead to errors. This can be done by associating this information to the project using persistent properties. Eclipse provides this persistent property mechanism to associate information with resources in the workspace, to which they belong. This information can be modified or seen by using so called preference pages. A Sindrion preference page is created which displays this information and provides ways to modify them without leading to errors.

5.1.3 Design of project Workspace

The mechanisms designed in the previous sections help the Sindrion projects to be identified by other plug-ins that provide support for development, and associate required information with the project. The Sindrion project designed will be a container for the code developed, that provides the functionality of Sindrion Proxy, Specific Control Point and Sindrion Applet. Java package is a mechanism for organizing Java classes into name spaces. Packages are typically used to organize classes belonging to the same category or providing similar functionality. Hence the Sindrion workspace needs to be designed that will help the user to distinguish and separate the code that contributes Sindrion components. This can be achieved by designing a nice package structure that will classify the workspace with respect to the Sindrion components.

Each project should have a main 'sindrion' package which holds the resources belonging to a Sindrion Application. This package can be subclassified into three packages that correspond to the three components, but analysis has shown the presence of some Java source files that provide functionality to proxy, SCP and applet as it is generated from the other two components. Hence the main package is classified in four subpackages namely 'proxy', 'scp', 'transceiver' and 'common'. The resources that contribute the functionality of the proxy are Java source files, the XML description files and files that contribute the presentation page. The XML files are assigned with proxy as the proxy code is generated based on the XML description files and the SCP code is generated to provide the GUI to control the proxy. Therefore, the 'proxy' package is further subdivided into 'docroot' and 'xml' packages. The docroot package contains the files that contribute the presentation page for the proxy and the xml package contains UPnP description files (UPnP device description and one or more UPnP device description files). The scp package contains the Java source files that contribute the functionality of the proxy. We know that the Sindrion Transceiver also acts as a webserver. The Sindrion Applet is generally embedded in the presentation page of the transceiver and thus can be accessed by other control points in the UPnP network. The files that provide this functionality viz. presentation page and applet classes are put in the 'transceiver package'. The 'common' package holds the Java source files that are common for the three components and hence can be associated with three components. This partitioning of the workspace gives a clear image of the resources that form a Sindrion Application. The Sindrion project with the designed package structure can be as shown in Figure 5.1.



```
SindrionProject
    com.infineon.sindrion.common
        SharedClass.java
    com.infineon.sindrion.proxy
        ProxyImpl.java
    com.infineon.sindrion.proxy.docroot
        presentation.html
    com.infineon.sindrion.proxy.xml
        device_description.xml
        service_description.xml
    com.infineon.sindrion.scp
        SCPImpl.java
    com.infineon.sindrion.transceiver
        Applet.java
        SindrionTX.html
    JRE System Library [jre1.5.0_05]
    .classpath
    .project
```

Figure 5.1: Sindrion Project

The phases of the development process that follow the initialization phase, also need the information regarding the location of these files viz. implementation phase needs create, edit or delete UPnP description files and Java source code files, testing phase also requires the location of the UPnP description files and Java source code developed during the Implementation phase, and the deployment phases has to bundle these resources with necessary libraries and transfer them to the transceiver. This package structure also serves the above mentioned purpose and more about this topic is discussed in Section 5.2.

In this section the design of the Initialization phase has been done. Each Sindrion project is now with a Sindrion nature which helps the plug-ins interested in Sindrion projects to identify them. For example, the plug-ins that support the user can check using this nature, if the particular project is a Sindrion project. The project as well is attached with the information gathered from the user during the initialization phase using the persistent property mechanism. This information is used during the development process to check the compatibility and provide support for particular platform version (of the transceiver). The designed package structure helps the user to identify the resources belonging to different Sindrion components within a project.

In next section we deal with design of a mechanism called wizard that helps the developer in creating the Sindrion project. Apart form creating the project the development environment can as well assist the developer by providing the skeleton package structure with basic data viz. UPnP description files, Java source code files, presentation pages for proxy and transceiver e.t.c.

5.1.4 Sindrion Project Creation support

The developer needs to perform the tasks designed above in a step by step process. First create a project, add Java and ‘sindrionNature’ to the project, add the library path to the classpath of the project, and then create the package structure and resource files. A mechanism that assists the developer to do these operations correctly is required. Eclipse wizards can provide special assistance for the developer for this purpose. Wizards automate repetitive and complex tasks through a user dialog. In Eclipse, wizards can create, import and export resources (files, folders and projects). The Eclipse platform contains many wizards ans as well makes it easy to write new “customized” wizards. A well designed wizard can considerably simplify developer tasks and increase the productivity.

A wizard is basically a series of screens or dialogue boxes that users pass through till the task is completed. The user generally needs to enter information, either by making selections or filling in the fields. Each of these fields is filled in with default values when the screen pops up, to assist the user. After entering the required data, user navigates through the screens and finally finishes the wizard which completes the task. Generally, the wizard should not be broken down into too many screens. This annoys the developer if they start feeling that the process is too long. On each screen the wizard should provide the purpose of the particular screen. The user as well should be provided with cancel option on every screen to exit the process without any effect.

The workbench defines extension points for wizards that create new resources, import resources or export resources. The ``org.eclipse.ui.newWizards'' extension point provides mechanism to add a wizard to the `File > New` menu. Since, the Sindrion Project Creation Wizard also creates a new project it has to extend this extension point. We have discussed that, every Sindrion project encapsulates a normal Java project along with Sindrion specific data. Eclipse provides a wizard to create new Java project. One way to design the Sindrion wizard could be extending the Java project creation wizard. Eclipse generally separates classes into two categories: public API and "for internal use only". The classes that contribute this wizard come into the category "for internal use only" and are internal to the plug-in that provides these mechanisms. These classes should not be referenced outside the plug-in, as may they may change drastically between different version of Eclipse. We will not extend these classes but create a new wizard.

The Sindrion project creation wizard extends the ``org.eclipse.ui.newWizards'' extension point and firstly it creates a Sindrion + Java project. The developer first needs to enter the name of the Sindrion project, the wizard will create. This wizard automatically scans the projects in the workbench and fills the field with text "SindrionProject + an integer" (so that the project with a same name does not exist in workspace). Then the user can change UUID, source package name and platform version, which are already having a default value provided by wizard. This wizard as well responds to changes and user actions. The data entered by the developer on the wizard page can have a number of errors by wrong choices or entering invalid values. The developer is informed of this error by giving a message, and when the developer corrects the error message the error message needs to be cleared.

The navigation buttons on a wizard page are managed using the JFace wizard support. Wizards with more than one page have Back, Next, Finish and Cancel buttons on each page. The Next is enabled for all but the last page and Back for all pages but the first. These buttons are controlled by implementing methods to check if the developer has selected/entered all the required information on the current page (to enable/disable Next button) or when the wizard can be completed (to enable/disable Finish button). Once the contents on the first page of the Sindrion wizard page are filled the Next and the Finish buttons are enabled. The developer can press Finish to complete the wizard or press the Next to go to second page of the wizard where he can configure the Java build settings and select the UPnP description files if already available. The wizard on finish creates the Sindrion project with name specified.

On completion (after pressing the Finish button) of the wizard, it as well creates the predefined package structure for the Sindrion applications which enables the Sindrion application to identify the location of Sindrion files. This package structure is designed in a way so that all the Sindrion plug-ins can access the DataModel created by parsing the workspace. This datamodel is explained in detail in Section 5.2.

This section described the design of the Sindrion project creation wizard, its functionalities viz. project creation, handling user data, creating package structures, prompt to switch the perspective. Next section deals with design of Sindrion project workspace and its role in the Sindrion Application development, design, creation and lifecycle during an eclipse session.

5.2 Modelling the Sindrion project

The analysis of the development process done in Chapter 2 shows that the phases of the development process depend on the contributions from the previous phases of development. They access this data and some times need to modify or provide with new information. For example, the platform version is set during the initialization phases and is used by other phases of the development process. So is with the components implemented during the implementation process and the testing and deployment process need to find these files in the workspace. This shows the flow of common data between the phases of development process. Since there is no current support from the development process to maintain this data or pass it to forthcoming phases, the developer needs to perform these actions manually. If a plug-in wants to find a proxy file it cannot do this without identifying this file. Apart from this developer also needs to the track the dependencies between the resources that belong to the different components of the development process, as we have seen the proxy Java source code is generated based on the UPnP description files and SCP Java source files in turn depend on the proxy source code. This shows the common data in the development environment which needs to be modeled.

Eclipse environment provided a Java model but does not suffice the needs in the context of Sindrion development environment, as a proxy Java class is not equal to a SCP Java class and this not reflected in the Java model. This shows the need to have an own project model which reflects the dependencies between different resources of a Sindrion project, and thus help to track the intrinsic dependencies. One solution for this would be to design access mechanism using static references to the resources with predefined naming conventions. This is not a flexible solution, as Sindrion is a research project and with a change in the project structure the access mechanisms needs to be redesigned for each of the phase. Hence we now design a datamodel independent of the project structure.

A good datamodel is foundational to allow users to access the right data quickly and easily.

Eclipse SDK comes with some predefined models viz. Workspace Model, Java Model, e.t.c. Workspace Model displays a resource based model as shown in the Resource Navigator. Java Model is the set of classes that model the objects associated with creating, editing, and building a Java program. The classes that belong to the Java model implement Java specific behavior of resources and further decompose Java resources into model elements. JDT as well defines the classes that model the elements that compose a Java program. The JDT uses an in-memory object model to represent the structure of a Java program derived from the project's class path. The Java model is hierarchical. Manipulating Java elements is similar to manipulating resource objects. The Java elements are actually the `handles` to some underlying model objects. Java elements comprise of elements that represent the root Java element, corresponding to the workspace (the parent of all projects with the Java nature), element representing Java project in workspace (child of Java model) and, further elements representing Java packages and program contents. Generally when a Java project is created from a simple project, `JavaCore` will check if the project is configured with Java nature. The Java nature is added when a "New Java Project" is created which helps the JDT plug-in to identify it as a Java project.

Plug-ins can configure the projects with Java behavior in addition to their own behavior by adding Java project nature and their own custom natures or behavior. Sindrion project is configured with Java nature in addition to the Sindrion project nature, for the plug-ins to recognize the Sindrion projects and perform corresponding actions. For example, the plug-in that builds the Sindrion model needs to filter the Sindrion projects from the workspace. In next sections we design the Sindrion datamodel, and discuss how other plug-ins can access, visualize the datamodel.

5.2.1 Sindrion DataModel

The Sindrion model designed should hold the common data in the development environment. This includes the resources that correspond to the Sindrion components (proxy, SCP and applet). The elements of the model should represent these components and their resources. Hence the model elements can be categorized into three types. Therefore it is a set of classes that model the objects associated to Sindrion applications. This model also should maintain a hierarchical in-memory object model to represent the structure of a Sindrion Application. Firstly, the Sindrion model should have a root node that represents the root of the workspace. This node is the parent of all the Sindrion Java projects in the workspace (all projects with Sindrion and Java natures). Then each of the Sindrion project is represented with a 'SidrionProject' element. These projects can be identified in the workspace using the 'Sindrion Nature' associated with the projects. The contents of these projects viz. Sindrion Proxy, Specific Control Point, e.t.c are of importance for the plug-ins that work with Sindrion Applications. So these contents also have to be as well automatically identified by the mechanism and modeled.The contents of the UPnP description files and Java source files are not modeled by the Sindrion datamodel as UPnP XML plug-in and UPnP Java plug-in model these resources and assist the developer in creating the xml description and automatically generate the Java code. These are the only plug-ins interested in contents of the xml and Java resources.

Sindrion Proxy resources include mainly the XML UPnP description files (one Device description and one or more Service Description files) and Proxy files. The proxy files can be categorized into two types viz. Sindrion Files and Web files. Sindrion files comprise Java source code files for Sindrion Proxy and other common files for Sindrion Proxy and Specific Control Point. The Web files comprise of presentation page and associated files as each proxy has a presentation page and other files associated with it. Sindrion Proxy has libraries different from other parts of the application. As the Sindrion Proxy is transferred over the network the libraries are packed individually for each part of application. These library paths are also to be attached to the proxy node. Each Sindrion Proxy also has a UUID (Universally Unique Identifier) which also has to be attached to the proxy node.

Specific Control Point resources include Sindrion files and classpath entries for SCP libraries. Sindrion files include Java source code files for Specific Control Point and common files for both proxy and SCP.

The third category is the Sindrion web files, which contains Web Files (presentation page and associated files for Sindrion Transceiver) and Applet. The Applet node includes Java source code files which represent the Sindrion Applet code. Each applet also has a node for the classpath entries for the libraries specific to Sindrion Applet. The datamodel of the project will be as shown in the Figure 5.2.

This datamodel helps to categorize the resources of a Sindrion project and therefore track the dependencies between the resources. The plug-ins can get

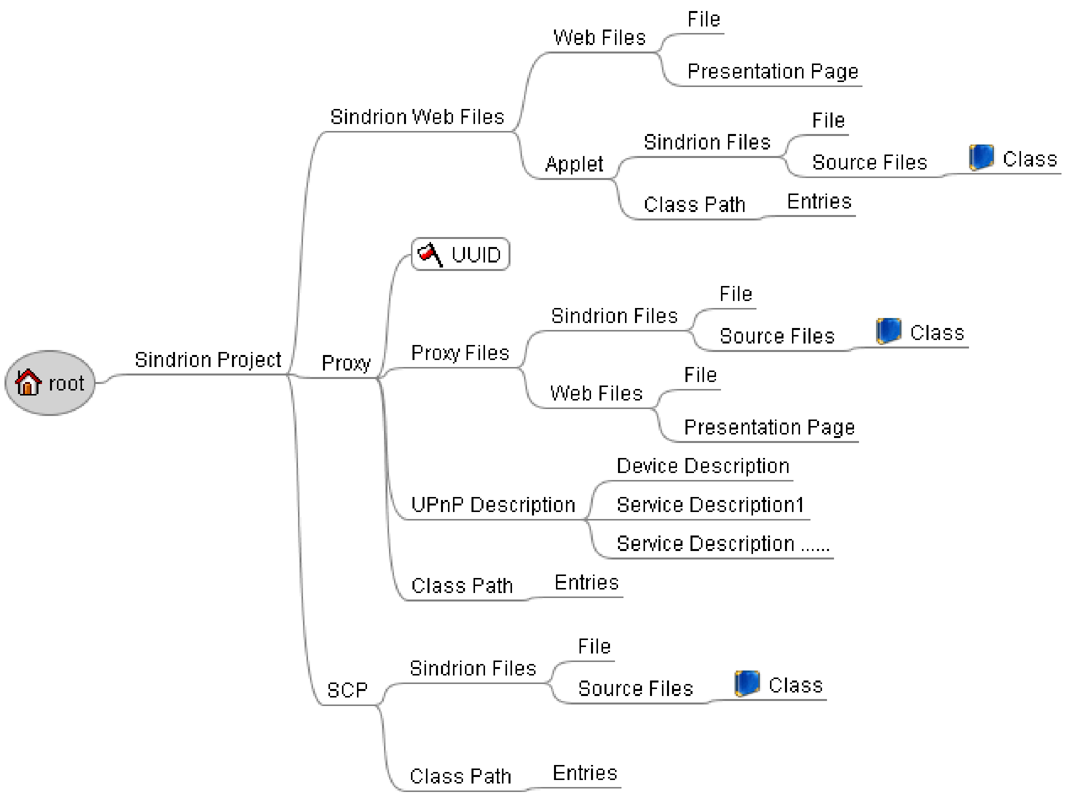


Figure 5.2: Sindrion DataModel

hold of the project and then each of the resources. For example, when a proxy file is opened in an editor the UPnP description view which listens to the editor input can check if this Java source file is a proxy file. If it is proxy file it can access the UPnP description files and generate its model which helps to create or edit description files. In next section we discuss the mechanisms that build this model.

5.2.2 Synchronization with Eclipse workspace

The Sindrion DataModel designed contains elements that represent a Sindrion project and its resources in the workspace. The Sindrion projects in the workspace can be identified with the nature, but to identify the resource we need to derive some mechanism.

The resources in the workspace can be identified by marking them with the persistent properties. Each resource can be tagged with a cross session persistent properties, which reflects its role in Sindrion project. Marking these resources is also good way when the Sindrion wizard creates the project in the workspace. But when the developer wants to add a resource to an existing Sindrion project, this resource has to be manually tagged with this persistent property based on the type of the file. IDE can provide some actions for this purpose, but still it is error prone.

One more way is to divide the resources in a project into packages that represent the parts of the application respectively. Design of the Sindrion project workspace done in Section 5.1.3 serves this purpose. The Sindrion projects should adhere to a `Package Structure` which allows these packages to be recognized as corresponding elements of the application. The Sindrion project creation wizard creates this package structure along with skeleton resources. This helps the developers a lot as they just have to modify the existing skeleton resources and then add necessary additional Java source files or UPnP description files.

The elements of the DataModel hold a reference to the respective resource in the workspace. This helps in the design the access mechanisms discussed in forthcoming section. Now a mapping between the elements in the datamodel and the Sindrion project has to be done. The proxy node in the model represents the package proxy. The UUID of the proxy is given by the wizard when creating the project using wizard. The Proxy Files child has two children Sindrion Files and Web Files. The Sindrion files are the Java source code files in proxy package and common package of the project. The proxy main class should be named as 'ProxyImpl.java' to differentiate from other source code classes. The Web Files represent the docroot package and it contents are added here viz. the presentation page of proxy and its resources. The UPnP description contains the description files from xml package. The UPnP device description file should be named as 'device_description.xml' and UPnP service description files should be named as 'service_description*.xml', as there can be more than one service description files, to differentiate between the resources. The ClassPath entries contain paths to all libraries the proxy uses.

The Specific Control Point elements represent the Specific Control Point package, which contains Sindrion Files element and ClassPath entries. The Source Files contains elements representing the Java source code files for proxy. The SCP main class should be named as 'SCPImpl.java'. The ClassPath entries contain the library paths on which the SCP depends on.

The Sindrion Web Files package represents the resources that reside on the transceiver viz. presentation page of the transceiver and applet. These files can be access as each transceiver also acts as a webserver. The Web Files child contains the file representing the presentation page and the Applet element contains the source files for Applet component. The Java source code file for proxy should be named as 'Applet.java'. The packages and resource files have to follow these naming conventions strictly, for these resources to get identified and modeled.

Building Sindrion DataModel

Sindrion datamodel should gathers the required data from the workspace and build its own model to support the Sindrion plug-ins. A mechanism that builds this model from the workspace model or Java model is devised. Firstly, the workspace handle is obtained from the 'ResourcePlugin'. The IJavaModel that represents the root Java element, corresponding to the workspace is obtained from JavaCore using the workspace handle. Sindrion model uses the **singleton** design pattern, to restrict instantiation of the class to one 'object'. This is used generally to coordinate the actions across the system. This root Java element that represents the workspace has all projects with Java nature, as children (IJavaProject). Then each of these Java projects is checked to see it the project is configured with 'sindrionNature'. For each of the Sindrion project in the workspace a 'SindrionProject' node is added to the workspace node, and this project is parsed to add the elements corresponding to the resources in the project. This mechanism is called in the start method of the Initialization plug-in.

In this section we have designed the Sindrion DataModel and mechanism that builds the Sindrion model on start-up of the Initialization plug-in. This helps other plug-ins to access the Sindrion resources in the workspace automatically.

Workspace Listener

The developer might create, delete or edit resources in the workspace during the development process. Sindrion DataModel should react to these changes in the workspace and update itself. The developer might delete the existing UPnP service and device descriptions and copy some existing description files into workspace. The Sindrion DataModel should update accordingly by deleting or adding its elements that represent these resources. The Eclipse system generates resource change events indicating, for example, the files and folders that have been added, modified, and removed during the course of operation. Interested objects can subscribe to these events and take actions to keep themselves synchronized with Eclipse. [8]

Eclipse uses the 'org.eclipse.core.resources.IResourceChageListener' interface to notify registered listeners when a resource has changed. The Sindrion DataModel can use these notification to synchronize itself with Eclipse, by registering 'SindrionWorkspaceListener' (implements IResourceChangeListener) for resource change events on start-up of the plug-in. In addition, SindrionWorkspaceListener should as well deregister on plug-in shutdown so that the listener is no longer notified of resource change events.

Eclipse provides several ‘IResourceChangeEvent’ constants that can be used in combination to specify when an interested object should be notified of resource changes like: POST_CHANGE, PRE_CLOSE, PRE_DELETE, e.t.c. It also specifies several methods to query its state:

`getDelta()` Returns a resource delta, rooted at the workspace, describing the set of changes that happened to resources in the workspace.

`getType()` Returns the type of event being reported.

`getResource()` Returns the resource in question.

Each individual change is encoded as an instance of a resource delta that is represented by the IResourceDelta interface. Eclipse provides several different constants that can be used in combination to identify the resource deltas handled by the system like: CHANGED, OPEN, DESCRIPTION, REMOVED, REPLACED, TYPE, e.t.c. The ‘IResourceDelta’ also defined methods to find the kind of resource delta, project-relative path of the resource delta, or a handle for the affected resource.

SindrionWorkspaceListener deletes the corresponding Sindrion projects from the model on PRE_DELETE, and PRE_CLOSE events. On a POST_CHANGE event,the resource delta is requested and handled to the ‘IResourceDeltaVisitor’. The *`visit()`* method is called for each resource change in the resource delta. The visitor uses a return value to indicate whether deltas for child resources should be visited. The delta has a REPLACED flag if the resource is renamed; hence the reference to the IResource is updated in the Sindrion element. When delta has ADDED or REMOVED flag the corresponding Sindrion element is added to or removed from the model. When the delta has OPEN flag, the particular Sindrion project is added to model and is parsed for its contents to be added. These listeners and visitors help the Sindrion DataModel to be synchronous with Eclipse.

In this section we had designed mechanisms to synchronize the DataModel with Eclipse. The model is built on start-up and listens to workspace events and updates itself. The plug-ins can thus depend on the model to access the Sindrion projects and resources in the workspace which gets automatically synchronized with Eclipse. Next section designs how Sindrion model communicates the changes in its model to other interested plug-ins.

5.2.3 Eventing Mechanism

The plug-ins that depend on the Sindrion DataModel, need to be notified of the changes on the model. The Sindrion DataModel extends the 'Default-TreeModel' class, which provides the default tree implementation methods along with the eventing mechanism. The model can fire events when a tree node is inserted, removed,and changed. Interested objects should implement the 'TreeModelListener' to be notified of the events on the Sindrion Data-Model. For example, the UPnP XML plug-in gets notified of the changes in the Sindrion DataModel, if it is accessing the resources of the Sindrion project that got changed. The Sindrion Model View also needs to get notified of the changes in the model to update the view with new elements or remove the deleted elements.

In this section we designed the Sindrion DataModel that acts as a basis for the Integrated Development Environment, by gathering the information from each phase of the development process, and user and passing in on to the forthcoming phases. Mechanisms that provide easy access to the plug-ins interested in Sindrion data are designed. Two views, one that displays the Sindrion DataModel and the second one for testing purpose are as well designed. Functionalities that keep the DataModel in sync with the workspace are also designed. In next section we give an overview of the complete Sindrion Integrated Development Environment and solve the issues that come up when looking at a big picture of the development environment.

5.2.4 Accessing DataModel

Plug-ins that provide services for the developer during the development of Sindrion applications need to access the Sindrion model, for example the Java UPnP plug-in needs to know the corresponding UPnP description files of the application to generate the Java application specific source code snippets. The UPnP XML plug-in as well need to get hold of the UPnP service description files corresponding to a UPnP device description while adding services to the UPnP devices.

Eclipse provides adapter framework (Section 4.5) which supply generic facilities for mapping objects of one type to objects of another type. This mechanism is used throughout the Eclipse Platform to associate behavior with objects across plug-in boundaries. The Sindrion elements implement the `IAdaptable` interface to participate in the adapter framework. The `IAdaptable` interface contains a single *getAdapter*(class) method for translating one type of object into another. Initialization plug-in on start-up, registers the 'Sindrion-AdapterFactory' that translates the objects of 'IResource.class' and 'IJavaElement.class' to Sindrion elements. When the adapter factory gets a request to translate the objects of the 'IResource.class' and 'IJavaElement.class' it parses the Sindrion datamodel to check if it can supply an adapter from the Sindrion model. The elements in the Sindrion model can also be adapted to the objects of type 'IResource.class' and 'IJavaElement.class'.

Each Sindrion model element representing a resource in workspace when created, holds a reference to the 'IResource' of the workspace elements. The adapter factory uses this reference to translate the elements from one type to other. For example, when a resource is opened in a editor the UPnP Core plug-in tries to find a Sindrion adapter for the resource currently open in the editor. If the input has a Sindrion adapter it automatically can query the Sindrion model for the Sindrion project to which the resource belongs to. The UPnP XML plug-in and the UPnP Java plug-in get reference to the description and Java source code files, from the Sindrion project. The developer just needs to open a resource and the IDE checks if this resource belongs to a Sindrion project. If the resource belongs to a Sindrion project, the UPnP description view is automatically populated with the UPnP description files to assist in developing the application. The developer can use this UPnP description view to develop this particular Sindrion application.

The Sindrion elements as well are translated into existing types viz. IResource.class' or 'IJavaElement.class'. This can be done by calling the *getAdapter*(class) on the element inherited from IAdaptable. Thus the Sindrion elements can be dynamically from and to the exiting types, which makes the access mechanism really useful in the development environment. Since the model has mechanism to synchronize with Eclipse the plug-ins can directly access datamodel.

Hence the Sindrion datamodel provides mechanism for other plug-ins to access its model using the adapter framework from eclipse, which facilitates the plug-ins to automatically get holds of the required Sindrion resources, keeping this job transparent to the user. With any changes in the Sindrion project structure the mechanism that builds the model only should be changed, thus keeping the workspace transparent to other plug-ins. Figure 5.3 shows the role of datamodel. Next section describes a view provided by the development environment for the Sindrion DataModel.

5.2.5 Visualization of DataModel

Eclipse plug-ins add a new Eclipse view or enhance and existing Eclipse view as a way to provide information to the user. It typically used to navigate a hierarchy of information, open an editor, or display properties for the active editor. The development environment is added with two new views 'Sindrion

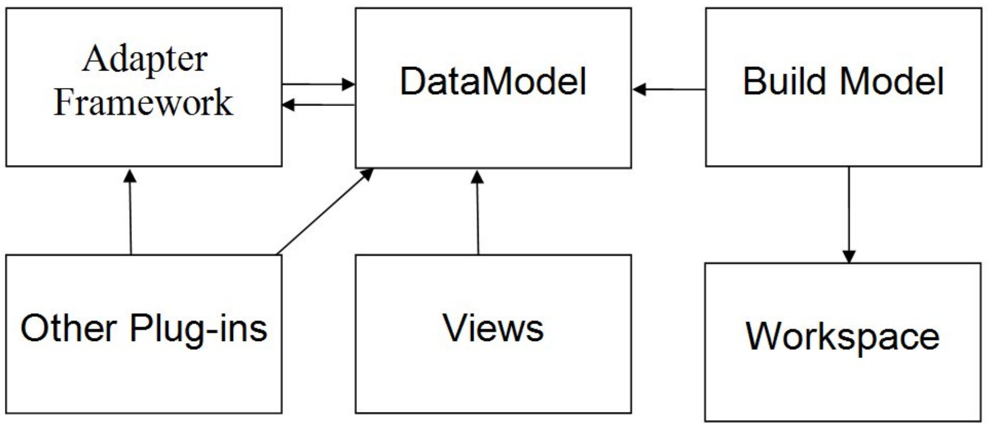


Figure 5.3: Role of Sindrion DataModel

Explorer' and 'Sindrion Adapter View' which are designed in the following subsections.

Sindrion Explorer

The Sindrion Explorer allows to take a look at the Sindrion projects in the workspace. It shows the logical structure of the Sindrion projects as they modeled in the Sindrion DataModel. In a way the content of the view is the Sindrion DataModel. Apart from visualizing the datamodel the view can also be used to change the properties associated with the elements. For example, the proxy node of each Sindrion Application has a UUID [3] associated with it. This is gathered from the user while creating the Sindrion project. Views can also have context menus populated by actions targeted at the view or selected objects within the view. The actions in these context menus for DataModel View can perform tasks like opening the resource associated with the particular Sindrion model element selected, deploying the Proxy/SCP code to the Sindrion Transceiver, or start the Proxy/SCP application for testing purpose. This explorer is an added value to the environment but still the user needs to access the projects Java related properties through the Package Explorer.

Sindrion Adapter View

The Sindrion adapter view is designed for testing the Sindrion DataModel. The DataModel parses the Java Model and builds its own model by identifying the Sindrion projects in the workspace. During the development of the IDE the developer might create new resources in the workspace and delete some exist-

[3]Universally Unique Identifier

ing resources. This view allows the developer to test if a resource is a Sindrion element or not. This view requests a selection service from the Workbench Window [4]. The returned selection service is added with a selection listener with partID for Package Explorer ("org.eclipse.jdt.ui.PackageExplorer"), which is notified when selection changes in Package Explorer. The selected object in the Package Explorer is checked if it can be adapted to the Sindrion Element and the result is displayed in the view. This helps the developer to see if the selected object is a Sindrion Element and what does it represent.

This section described the two new views added to the development environment which assist the user in viewing, testing, and performing actions on the Sindrion DataModel. The Sindrion DataModel view and the Sindrion Adapter View should be added to the shortcut `Window > Show View`, which makes them easily accessible to the developer.

5.3 Set-up the Environment

During the development process the developer needs to have a knowledge of the mechanisms provided by this Integrated Development Environment. Even if the developer has an idea of these, figuring out how to access these mechanisms is also significant issue. The development environment can make these mechanisms easily accessible to the developer by providing shortcutss .

In addition to the views and editors that make up the bulk of the display area, Eclipse also includes a large number of menus (Top-level menus and context menus) and toolbar buttons that represent the various commands or actions available in the system. [8] Eclipse as well gives some limited control to customize the items that appear on the toolbar and main menu bar. The commands are a part of the *command groups* known as *action sets* that can be enabled and disabled using filters. A top-level menu is great way to promote a new product that has been installed in Eclipse IDE, providing a good way for a potential customer to become accustomed to new functionality. On the other hand menu bar would be cluttered and Eclipse would become unusable if every plug-in defines a top-level menu. One of the options is to tie the top-level menu or action set to a particular perspective so that the menu and actions are only visible when the particular perspective is active.

The various views and editors visible within the Eclipse workbench used to work with various resources are known as a *perspective*. Hence the *perspec-*

[4] A workbench window is a top level window in a workbench. Visually, a workbench window has a menubar, a toolbar, a status bar, and a main area for displaying a single page consisting of a collection of views and editors.

tives are a way to group Eclipse views and actions for particular task such as coding or debugging. The action sets can be enabled or disabled for a particular perspective. Eclipse enhancements that involve multiple plug-ins generally provide their own perspectives, by enhancing existing perspectives or by providing entirely new ones. Eclipse enhancements that provide one or two new Eclipse views typically enhance existing perspectives.

Eclipse IDE provides a variety of perspectives which are used while performing different tasks viz. coding, testing, debugging or developing plug-ins. An existing perspective can be extended by adding new views, placeholders[5], shortcuts, and action sets. The views as well can be combined into a single tabbed area or can be added to the fast view bar to save some space in the perspective.

Resource perspective is the initial perspective shown in the workbench window. The primary view within the `Resource` perspective is the `Navigator` view. The `Navigator` view presents a hierarchical view of the resources (projects, folders and files) loaded in the workbench. The `Navigator` view has its own toolbar and view menu that provide various viewing and filtering options.The `Resource` perspective also shows up the `Outline` view. The `Outline` view shows an outline of the structural elements of the selected editor. The contents vary depending on the type of editor in use. For example, when editing a Java class, the `Outline` view displays the classes , fields, and methods in the Java class being edited. The `Outline` view includes a number of options to control the elements displayed withing the outline (using filters).

Java perspective has the `Package Explorer` within as the primary view and the `Hierarchy` view tabbed to it. The `Package Explorer` shows the hierarchy of Java files and resources within the Java projects loaded into the current workbench, providing a very Java-centric view of resources rather than a file-centric view. Rather than showing Java packages as nested folders as in the `Navigator` view, the `Package Explorer` shows each package as a separate element in a flattened hierarchy. Any JAR file that is reference within a project can also be browsed using this view. The `Java` perspective also shows up the `Outline` view.

Java Browsing perspective includes as series of linked views. The first view shows a list of loaded projects. Selecting a project shows its contained packages within the `Packages` view; selecting a package shows its types in the `Types` view; and selecting a type shows its members in the `Members` view.

[5]A placeholder can be added to the perspective for a particular view so that when the user opens this view, it appears in the correct place

Java Type Hierarchy perspective has the `Type Hierarchy` view as the primary view. The `Type Hierarchy` view shows the superclasses and subclasses of a given type. The view also has options for showing just the super-type hierarchy (both superclasses and implemented interfaces) or sub-type hierarchy (subclasses and interface implementers) of a type.

Debug perspective is used to debug programs and easily find and correct run-time errors in Java code. One can step through individual statements within the code, set breakpoints, and inspect the values associated with individual variables.

The development process of Sindrion Applications mainly includes development of XML (UPnP device and service description files) and Java code (Proxy, SCP and Applet). With existing support for the Implementation phase the developer can edit or write new UPnP device or service description files by using a comfortable GUI (UPnP description view [10]). This support also can automatically generate Java source code snippets from correctly identified UPnP description, based on Java UPnP stack. But these code snippets (generated automatically) still have to be 'filled' with the actual implementation of the desired functionality, by the developer. The perspective provided by the development environment should be convenient for the development of Java code.

While the `Resource` perspective provides a nice, general way to look at the resources in a system, it is not the ideal perspective to use for general Java development. The `Java` perspective and `Java Browsing` perspective included in Eclipse are optimized for the development of Java code. The Java Browsing perspective has four views viz. Projects, Packages, Types and Members above the editor. These views show only the content related to Java projects and Java classes. The Projects view displays only a list of the Java projects in the workbench. When expanded the project shows only the source folder and the referenced libraries, the project related files (.classpath, .project) other than these are not reflected in the view. When selected a project in the Projects view the Packages view only shows a list of packages contained in the project. Selecting a package shows its types in the Types view and selecting type shows its members in the Members view. The Sindrion project as well contains the XML files and HTML files viz. the UPnP description files and presentation pages for Sindrion Proxy and Sindrion Transceiver. The developer needs a view that can display all the contents of the projects. For example, the developer may need to open the presentation pages to edit the content. Hence extending the `Java Browsing` will not be a correct solution.

The Java perspective as discussed above has a Package Explorer view tabbed with Hierarchy view on the left side of the editor and Outline view on the right side of the editor. The package explorer displays all the projects loaded in the workbench, as well allows to browse the project display all files and packages irrespective of their types (XML, Html, .project, .classpath e.t.c). The display of the content can be refined using the filters provided by the view. The content in the Outline view is based on the type of the editor in use. New editors can as well define their own outline page. The editors provided by Eclipse (default, XML and HTML editors) has Outline pages predefined that show the structural elements of the selected editor.

Hence, extending the Java perspective would be ideal and optimal solution for the development process of Sindrion Applications. New views, placeholders for the views, shortcuts and action sets related to the Sindrion Application development will be added during the design of the Integrated Development Environment, which in turn will ease the process of development. Consequently, opening this perspective gives easy access to the developer to the mechanisms provided by the Integrated Development Environment. Further, the Debug perspective provided by Eclipse can be enhanced for Testing phase of the development process by integrating Sindrion Transceiver Simulator into the Java debugger (provided by Eclipse) and thus easily find and correct runtime errors in Java code.

The project creation wizard as well prompts the user to switch to Sindrion perspective once the project is created, by specifying the Sindrion perspective as the final Perspective in the extension point. This helps the developer, by making available the mechanisms provided by the development environment which the developer can use in the process of Sindrion application development. The Sindrion Project Creation Wizard should be added to the `File > New` menu of the Sindrion perspective which makes it easily accessible to the developer. The views provided by the environment viz. Sindrion Explorer, Sindrion Adapter View and UPnP Description View also are added to the Window -> Show View which helps the developer to access these views easily.

This section mainly lays a base for the set-up of the environment by enhancing an existing perspective (Java perspective) which provides shortcuts for, views and menus for the mechanisms provided by the Integrated Development Environment. The necessary view are also opened along with the perspective to ease the development of applications.

5.4 Sindrion Integrated Development Environment

Sindrion Integrated Development Environment aims at supporting the developer during the complete development process of Sindrion Applications. We have already discussed in Section 2.3 regarding the existing support for development process of Sindrion Applications. The Core Plug-in, UPnP Description XML Plug-in and UPnP Java Plug-in, help the user during the Implementation phase to develop Java based UPnP applications. The Sindrion Transceiver Simulator mimics the functionality of the Sindrion Transceiver and provides all the features of the transceiver that are used by the applications, so that transceiver and STS can be exchanged and this change is transparent to the software components that are developed. This tool can be used to debug the applications during the Testing phase, inside the development environment. The existing Flash Tool can be used for deployment by supplying the required information to transfer the components to transceiver's memory. The support designed in this chapter assists the developer during the unsupported Initialization phase, to set-up the environment and create Sindrion projects. It also brings in a mechanism (DataModel) to bridge the gaps between the existing support by providing a data flow channel that gathers and pass the information between these phases of development process. This DataModel as well acts as means of communication between the plug-ins. The integration of this supports mechanisms builds an ideal development environment for Sindrion Applications. The integration and its consequences are discussed in next section.

Sindrion software components are realization of Service-oriented architecture [6] (SOA) and provide their functionalities as services. The implementation of the Sindrion services uses Universal Plug and Play (UPnP) as the middleware for discovery advertisement and communication. The components are implemented in Java as it would allow us to run them on any platform that has a Java virtual machine and a network connection available to connect the components.

Service-oriented architectures also come with many benefits like programming language independent and software reuse. Sindrion project also plans to extend its support by using 'Web service' instead of UPnP or 'C' instead of Java to implement the components. This increases the scope of the Integrated Development Environment four-fold. Hence the Sindrion Integrated Development Environment should be designed in a way so that new plug-ins that contribute additional functionalities and mechanisms can be added to

[6] A SOA is a concept that homogenizes distributed heterogeneous environments by declaring arbitrary functional units as services.

the development environment. In next section we describe the plug-ins which collectively form the Sindrion Integrated Development Environment.

5.4.1 Plug-in Dependencies

The Sindrion Integrated Development Environment is a set of plug-ins that provides functionalities to assist the developer from Initialization to Deployment phase. In this section we will have an overview of the plug-ins that supply the assistance to the developer. Figure 5.4 shows the big picture of the plug-in components including their models that form Sindrion Integrated Development Environment.

The development environment has the SOA plug-in as the central plug-in, and has a SOA specific model of the workspace. It provides and extension point for the plug-ins that can extend this SOA model. The SOA model is extended by UPnP XML and WebServices XML plug-in. The UPnP XML plug-in provides an extension point for implementation specific plug-ins. The Sindrion Java Init plug-in designed as a part of this thesis extends the UPnP XML plug-in. A plug-in that implements the Sindrion components in other programming languages like C or C++ can also extend this extension point. In software development, reuse of code is desirable as it minimizes the development effort and simplifies maintenance. Hence, the plug-ins are developed to facilitate the reuse of the code for further support like support for development of Sindrion application that use Web services as middle ware C as a programming language.

Presently we have the support for Sindrion applications which use which UPnP as middleware and Java as programming language. Hence we proceed the discussion on this part of the IDE. UPnP XML plug-in has a programming language independent model for the UPnP projects in the workspace and SOA plug-in model as well gets updated. Based on the type of the project the Sindrion Java Init plug-in or Sindrion C plug-in is activated. The Sindrion Java Init plug-in depends on the Workspace and JDT to scan the Sindrion projects available and build its own datamodel. The UPnP Java plug-in has a model which is independent of Sindrion specific details. This plug-in directly depends on the SOA plug-in to builds its model based on SOA model. Sindrion Java plug-in has a direct dependency on UPnP Java plug-in and builds two models one for Sindrion Proxy and one for Specific Control Point. This plug-in also depends on Sindrion Java Init plug-in to get the Sindrion project details. The Test/Debug plug-in used for testing the Sindrion applications retrieves the project from Sindrion Java Init plug-in and used JDT for testing the applications. Deployment plug-in fetches the ready to deploy Sindrion project and related information from Sindrion Java Init plug-in and deploys the application to the Sindrion Transceiver.

The plug-ins designed support extensions that provide enhancements to the existing features in a controlled yet loosely coupled and flexible way. By deriving these dependencies and extensions the plug-ins in the development environment follow the Eclipse rule of lazy activataion. One issue to handle is the start-up of these plug-ins in an Eclipse way which is designed in next section.

5.4.2 Start-up of Plug-ins

The principle of lazy plug-in activation is very important in Eclipse platform. Plug-ins are activated only when their functionality has been explicitly invoked by the user, which in turn results in a relatively small start-up time and a memory footprint. So the designed components as well should follow the eclipse principle for start-up. Hence, designing the start-up of the IDE components is also a challenging and critical issue. For example, the developer doest not utilize the support for Testing and Deployment phases while implementing the Sindrion Applications which need not be started until explicitly invoked by the user. We have discussed the various possibilities of activating a plug-in in section 4.4. The extension point mechanism plays an important role in lazy activation. The SOA plug-in first gets started when the developer starts the SOA view or activates the Sindrion perspective which in turn opens the SOA view. Based on the type of the project, the SOA plug-ins searches the available extensions and activates the UPnP XML or Web services XML plug-in. The UPnP XML in turn searches for the extensions and activates Sindrion Java Init plug-in or Sindrion C Init plug-in. The UPnP Java plug-in listens to the changes on SOA model to update itself and thus has a direct dependency on SOA plug-in. The UPnP Java plug-in also has direct dependency on the UPnP Java plug-in. The Test/Debug and Deployment plug-ins depend on the Sindrion Java Init plug-in and gets activated when either or these plug-ins are started by the user. The dependency relation mentioned in the above cases, loads a class of the prerequisite plug-in which causes the it to be started automatically. These relations help the plug-in that contribute to the Sindrion Integrated Development Environment to follow the lazy plug-in activation principle of Eclipse platform.

In this section we first discussed about integration of the contributing plug-ins to form the Sindrion Integrated Development Environment. Then an overview of the complete system components is given, by explaining the functionalities of individual plug-ins and the role they play. The start-up issue of the plug-ins that rises due to the lazy activation principle is solved by defining the relationships between the plug-ins so that they get activated only when their functionality is explicitly invoked.

5.5 Summary

In this Chapter the development environment is done based on the considerations made in Chapter 3. The design process addresses the problems faced by the developer and mechanisms to realize an IDE. The support designed for the initialization phase using the Eclipse wizard helps to set-up and configure the initial project with Sindrion nature and create the resources, which helps the user to do perform the tasks of implementation phase with minimum effort. This wizard as well prompts the user to switch to a perspective which exposes the funtionalities and support provided by the environment for development of Sindrion applications. Modelling of the Sindrion project serves the purpose, by forming the common data channel between the plug-ins thus integrating them into the development environment. Access mechanism for model are designed which provide an abstraction layer upon the Eclipse workspace. Integrating plug-ins developed for supporting the individual phases of development forms the Sindrion Integrated Development Environment. The start-up issue of these plug-ins was considered to comply with the lazy activation principle of Eclipse. The relationship between these plug-ins is defined so that they get activated only when their functionality is explicitly invoked. In next Chapter implementation of the concepts derived here is described.

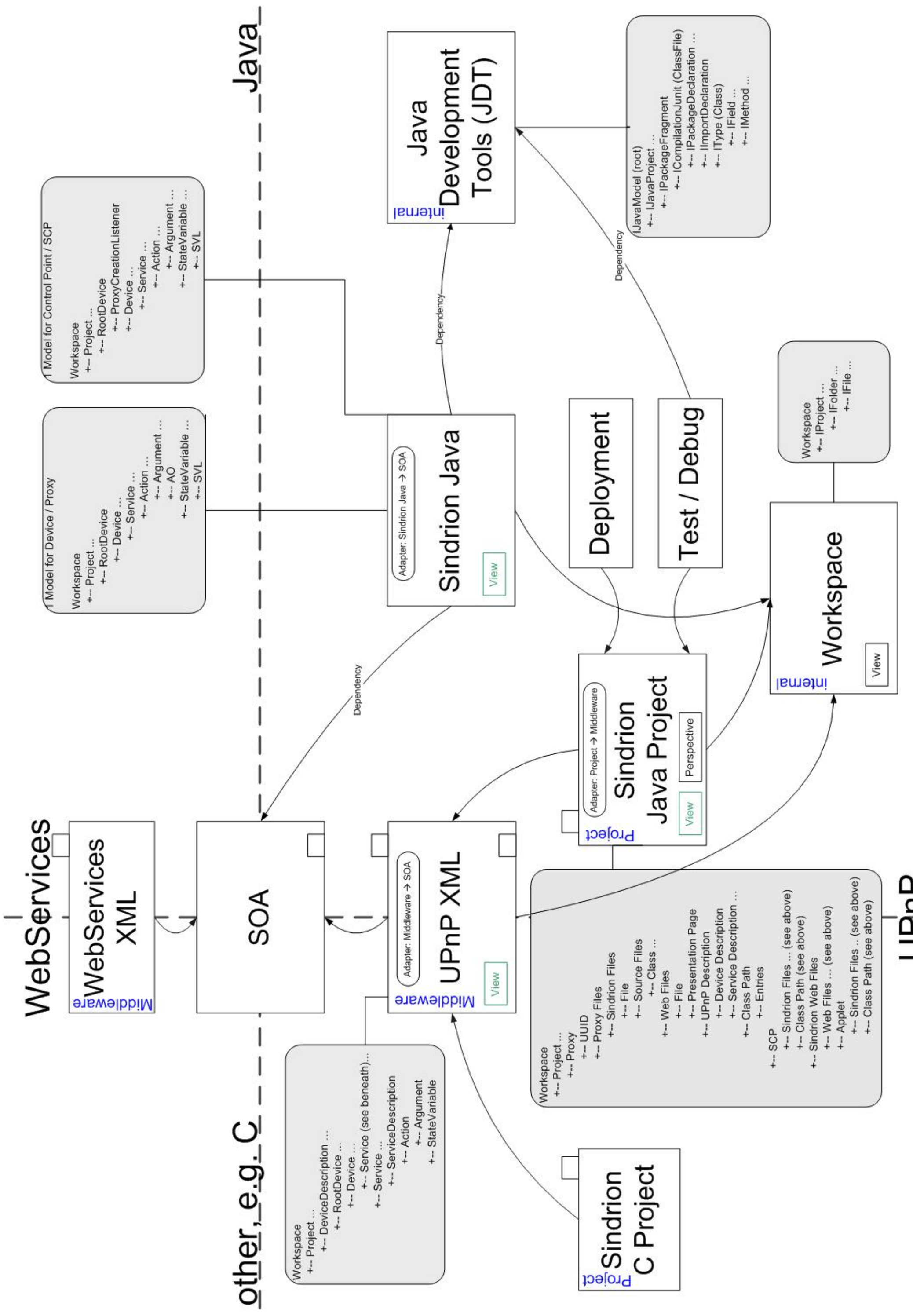


Figure 5.4: SiDE Plugin Components and Models

Chapter 6

Implementation

This chapter gives the implementation details of the plug-in which support initialization and integration of the Sindrion Application development environment. This proves the concepts derived in this Thesis, and shows that Sindrion Integrated Development Environment eases the development of Sindrion software. This implementation is realized using the Eclipse Software Development Kit (SDK) Version 3.1.2 in connection with Java Standard Edition Development Kit (JDK) version 1.5.0_06. In the following sections, functionalities of the Initialization plug-in are presented.

6.1 Sindrion Nature

Sindrion Nature is used by the plug-ins to identify the Sindrion projects in the workspace. A new "org.eclipse.core.resources.natures" extension is created in the Initialization plug-in manifest. Then the name and the local identifier are assigned. Natures also have behavior to configure and deconfigure a project. For example, the nature can add a builder to the project's build spec. The 'sindrionNature' is associated with a runtime class "com.-infineon.sindrion.sde.natures.SindrionNature" which implements "org.eclipse.-core.resources.IProjectNature". When the Sindrion nature is added to a project, this class is instantiated and the *setProject*() method is called followed by *configure*(); the *deconfigure*() is called when the nature is removed. Nature can also be used to associate builders with Sindrion projects. These two methods are be used to add or remove a builder from the buildspec of the project.

We have discussed that each Sindrion project is an encapsulated Java project with Sindrion specific details. Hence we have to express the dependency of Sindrion Nature on Java Nature. This can be done by adding the Java nature with a require-nature tag to the declaration. The Sindrion nature declaration can be seen in Listing 6.1.

Listing 6.1: The ‘sindrinNature’ declaration.

```
<extension
     id="sindrionNature"
     name="Sindrion␣Nature"
     point="org.eclipse.core.resources.natures">
   <requires-nature id="org.eclipse.jdt.core.javanature"/>
      <runtime>
         <run class="com.infineon.sindrion.sde.natures.
            SindrionNature"/>
      </runtime>
</extension>
```

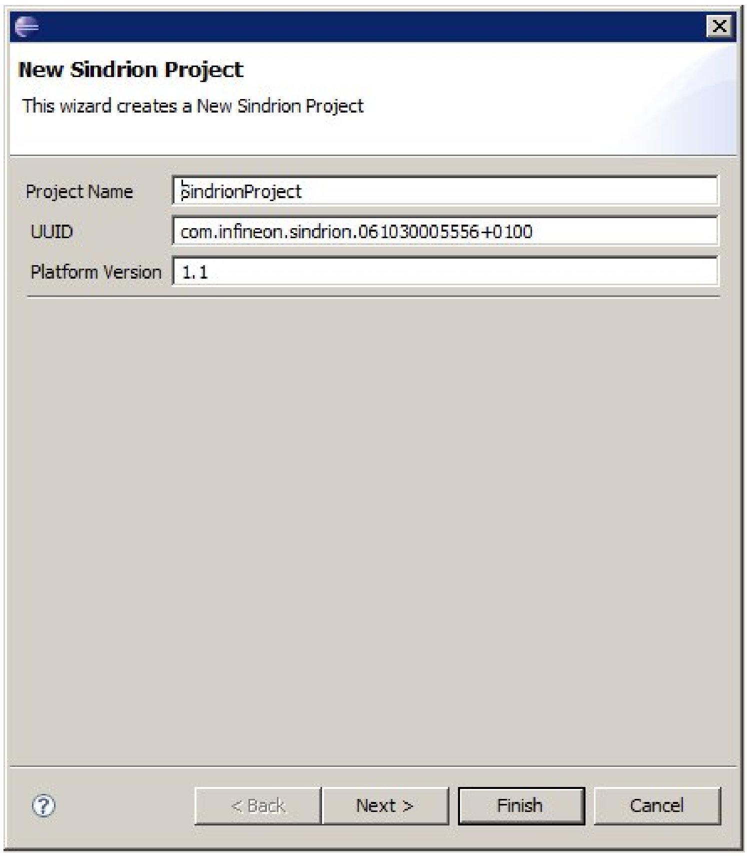
6.2 Project Creation Wizard

With the help of this wizard the developer can create a Sindrion project by passing through a series of screens and entering the required information. This wizard on successful completion creates a new Sindrion project in the workspace and hence an extension of the “org.eclipse.ui.newWizards” is created. The ‘SindrionWizard’ implements the ‘INewWizard’ interface which allows to obtain pages to be displayed and notify the wizard of the user interaction. This wizard contains a ‘SindrionMainPage’ which extends the ‘WizardPage’ class to present a page of information to the user, and validate the information entered by the user on that page. The project attribute of the wizard should be set to true to show up in the `File > New > Project` menu. The plug-in manifest entry for the wizard can be seen in Listing 6.2

Listing 6.2: The plug-in manifest entry for Sindrion wizard.

```
<extension
     id="com.infineon.sindrion.sde.init.wizards"
     name="SindrionWizards"
     point="org.eclipse.ui.newWizards">
   <category
        id="com.infineon.sindrion.sde.init.wizards.
           SindrionWizards"
        name="Sindrion␣Wizards"/>
   <wizard
        category="com.infineon.sindrion.sde.init.wizards.
           SindrionWizards"
        class="com.infineon.sindrion.sde.init.wizards.
           SindrionWizard"
        descriptionImage="icons/sindrion-logo.gif"
        icon="icons/sindrion-logo.gif"
        id="com.infineon.sindrion.sde.init.wizards.
           SindrionWizard"
        name="Sindrion␣Project"
        project="true"/>
</extension>
```

When the user launches using `File > New > Project > Sindrion Project`, a wizard screen as shown in Figure 6.1 is opened. It contains a text field for the project name. The wizard automatically searches the projects in workspace and suggests a name for the user. The SindrionMainPage also implements the Listener interface to handle the user events on the fields. The project name field is added a listener which disables the finish button on the page if the field is empty or if a project with the same name already exists in the workspace. Once the user enters the project name that satisfy the requirements, the finish button gets activated and the user can press this button to finish the wizard.



Project Name: SindrionProject
UUID: com.infineon.sindrion.061030005556+0100
Platform Version: 1.1

Figure 6.1: Project Creation Wizard Screen

On pressing the finish button the page calls the *performFinish*() method of the wizard dialog. This method creates and configures the new Sindrion project. First a project is created in the workspace and added with Sindrion and Java natures.

```java
private void createSindrionProject(String projectName,
   IProgressMonitor monitor){
   IWorkspace workspace = ResourcesPlugin.getWorkspace();
   IProject proj = workspace.getRoot().getProject(projectName);
   try {
      if (!proj.exists())proj.create(monitor);
      proj.open(monitor);
      IProjectDescription desc = proj.getDescription();
      desc.setNatureIds(new String[] {
      JavaCore.NATURE_ID,SindrionDEInitPlugin.getDefault().
         getBundle().getSymbolicName()+ ".sindrionNature" });
      proj.setDescription(desc,monitor);
      IJavaProject javaProj = JavaCore.create(proj);
      ......
      } catch (CoreException e) {
      e.printStackTrace();
      } catch (FileNotFoundException e) {
      e.printStackTrace();
      } catch (IOException e) {
      e.printStackTrace();
      }
  }
```

Once the project is created and the Java nature and Sindrion nature are assigned to the project. The wizard creates the package structure in the project. It creates the packages for Sindrion Proxy, Specific Control Point, transceiver, docroot and e.t.c, according to package structure designed in Section 5.2.1. Then the skeleton UPnP description files and Java source code files are as well created. This helps the user to right away start the application development.

This wizard will be extended to support the user, by providing extra support for configuring and requesting additional project related data to meet the complete design requirements.

6.3 DataModel

This section describes the implementation of DataModel, the central component of Sindrion Integrated Development Environment. It gathers the Sindrion project data from workspace and acts as a data channel between the plug-ins that contribute the support for Sindrion Application development. First the implementation details of this hierarchical tree model are given and then in the next sections the mechanism that update and provide access to the model are described.

Sindrion model uses the **singleton** design pattern to restrict instantiation of the class to one 'object', which helps to co-ordinate the actions across the system. This is done by providing a method which checks if model already exists, if not creates a new model.

```
private SindrionModel() {
   super(new SindrionModelRootElement(null));
 }

public static synchronized SindrionModel getModel() {
   if (model == null) {
      model = new SindrionModel();
      IWorkspace workspace = ResourcesPlugin.getWorkspace();
      IJavaModel javaModel = JavaCore.
                          create(workspace.getRoot());
      ((SindrionModelElement) model.root).addChild(new
         SindrionModelRootElement(javaModel.getResource()));
   }
   return model;
 }
```

The *getModel*() method first checks if an instance of the model already exists. If not, the root of the IJavaModel is obtained from JavaCore using IWorkspace. The model extends DefaultTreeModel which provide the basic functionality of a tree model and brings along the eventing mechanism to the model, using which the model can fire an event when an element is inserted, removed or changed. The ISindrionElement the interface for Sindrion elements extends IAdaptable to participate in the Adapter Framework and TreeNode to participate in the TreeModel. Elements of the model implement the ISindrionElement interface. Each element of the Sindrion model that represents a resource in the workspace holds a reference to the respective resource. This is used by the access mechanisms provided by the datamodel. The SindrionModelElement *abstract* class provides generic and common behavior to the elements of the model by providing variable declarations and methods viz. maintains a array list of children, provides methods to add or remove child for particular element and provide get and set methods for the resource associated with the element.

Elements of the datamodel viz. Proxy, SCP, SindrionWebFiles e.t.c extend this abstract class use this behavior provided and add more specific methods on the element, like the SinidrionProject element can only have children only of type Proxy, SCP or SindrionWebFiles. Each element is restricted to have only children that are permitted according to the model designed in Section 5.2.1 which can be seen in Figure 5.2.

Now that we have implemented the datamodel, this model need to be built based on the workspace resources. The Initialization plug-in gets activated when the user explicitly invokes the functionality of the plug-in. In the *start*() method of the plug-in, the method that parses the workspace for Sindrion projects and builds the model is invoked.

Listing 6.3: The Initialization plug-in class.

```java
public class SindrionDEInitPlugin extends AbstractUIPlugin {

  private static SindrionDEInitPlugin plugin;
  private SindrionAdapterFactory sindrionAdapterFactory;
  private SindrionWorkspaceListener swl;
  ...
  public synchronized void start(BundleContext context) throws
     Exception {
     super.start(context);
     BuildSindrionModel.parseForSindrionProjects();
     addListener();
     ...
   }

  public void stop(BundleContext context) throws Exception {
         ...
     removeListener();
     plugin = null;
   }
  public SindrionDEInitPlugin() {
     plugin = this;
   }

  private void addListener() {
     if (swl == null) {
        IWorkspace workspace = ResourcesPlugin.getWorkspace();
        swl = new SindrionWorkspaceListener();
        workspace.addResourceChangeListener(swl,
           IResourceChangeEvent.PRE_DELETE |
           IResourceChangeEvent.POST_CHANGE);
        System.out.println("Workspace␣Listener␣Added");
      }
   }

  private void removeListener() {
     if (swl != null) {
        IWorkspace workspace = ResourcesPlugin.getWorkspace();
        workspace.removeResourceChangeListener(swl);
        System.out.println("Workspace␣Listener␣Disposed");
      }
     swl = null;
   }
  ...
 }
```

Listing 6.3 shows that the SindrionDEInitPlugin class that extends the AbstractUIPlugin on start-up supplies the super class with bundle context and calls BuildSindrionModel.*parseForSindrionProjects*() static method. This method first gets hold of Java model and identifies the Sindrion projects from the model with the help of Sindrion nature.

Listing 6.4: The *parseForSindrionProjects*() method.

```java
public synchronized static void parseForSindrionProjects() {
   IWorkspace workspace = ResourcesPlugin.getWorkspace();
   IJavaModel javaModel = JavaCore.create(workspace.getRoot());
   try{
     IJavaProject[] javaProjects = javaModel.getJavaProjects();
     for (IJavaProject j:javaProjects){
       if(j.getProject().hasNature(SindrionDEInitPlugin.
           getDefault().getBundle().getSymbolicName()+ ".
           sindrionNature")){
         createSindrionProject(j);
        }
      }
    }
   catch (JavaModelException e){
      e.printStackTrace();
   }catch (CoreException e) {
      e.printStackTrace();
   }
 }
```

Then the model is added with the elements that represent the Sindrion projects and its elements in the workspace, by parsing the projects and identifying the elements based on the defined package structure and naming conventions defined in Section 5.2.1. This builds the datamodel which represents the Sindrion Projects in workspace.

6.3.1 Resource Change Listener

The model should also be kept synchronous with Eclipse. This task is accomplished by implementing SindrionWorkspaceListener which implements IResourceChangeListener. This listener is registered to the workspace, which notifies the model of the changes in the workspace which can be seen in Listing 6.3 lines 23-30. Based on the events this listener adapts the model to keep in sync with Eclipse, like deleting the Sindrion Project when a project is closed or deleted and adding it when a project is opened or created. The changes to the elements of the project are handled by the SindrionResourceDeltaVisitor which implements IResourceDeltaVisitor, whose *visit*() method is called for each change in the resource delta. The *stop*() method of the plug-in class i modified to call the *removeListener*() so that the listener is no longer notified of the resource changes once the plug-in has been shutdown (Listing 6.3 lines 32-39).

6.4 Adapter Factory

The plug-ins that support the development of Sindrion applications might need to translate existing types into new types of objects viz. Sindrion elements. The Adapter factory (Section 4.5) can used to to accomplish this task. The Sindrion Initialization plug-in provides a SindrionAdapterFactory which implements the "org.eclipse.core.runtime.IAdapterFactory" interface. The *getAdapter*(...) method of the factory can translate IResource and IJavaElement objects into ISindrionElement objects. The *getAdapterList*() returns an array indicating the types to which the factory can translate, in this case it returns ISindrionElement.class. If an adapter of type ISindrionElement.class is requested the adapter factory parses through the complete tree and checks if it can provide an adapter. Listing 6.5 shows the implementation of the Sindrion adapter factory.

Listing 6.5: Sindrion Adapter Factory class.

```
public class SindrionAdapterFactory implements IAdapterFactory{

   private static Class[] SUPPORTED_TYPES =
             new Class[] { ISindrionElement.class };

   public Class[] getAdapterList(){
      return SUPPORTED_TYPES;
    }

   public Object getAdapter(Object object, Class key){
      if (ISindrionElement.class.equals(key)){
      ISindrionElement root = SindrionModel.
                                getModel().getRoot();
      try{
         Object adapter = findAdapter((ISindrionElement) root,
                                    object);
         if (adapter != null)
            return adapter;
         }catch (RuntimeException e) {
         e.printStackTrace();
         }
      }
      return null;
   }

   private Object findAdapter(ISindrionElement ele, Object
      object) {
```

```
      if (((SindrionModelElement) ele).isAdapterFor(object))
      return ele;
      else{
      ArrayList<ISindrionElement> children = ele.getChildren();
      for (ISindrionElement c : children) {
         Object o = findAdapter(c, object);
         if (o != null)
            return o;
         }
      }
       return null;
   }
}
```

6.4.1 Register the Adapter

The adapter factory is registered declaratively by initialization plug-in using the “org.eclipse.core.runtime.adapters” extension point. This information is used to by the runtime XML expression language to determine existence of sindrion adapters without causing plug-ins to be loaded. Registration of adapter factories via extension point eliminates the need to manually register adapter factories when a plug-in starts up.

The Listing 6.6 declares that our plug-in will provide an adapter factory that will adapt objects of type IResource.class to objects of type ISindrionElement.class.

Listing 6.6: Registering the adapters.

```
<extension point="org.eclipse.core.runtime.adapters">
   <factory
      class="com.infineon.sindrion.sde.datamodel.
         SindrionAdapterFactory"
      adaptableType="org.eclipse.core.resources.IResource">
     <adapter type="com.infineon.sindrion.sde.datamodel.
        ISindrionElement"/>
   </factory>
</extension>
```

For example, the Core plug-ins checks if the resource presently opened in the editor has a ISindrionElement.class adapter by querying the Sindrion adapter factory. If it finds an adapter then the UPnP description files that belong to the Sindrion Project are parsed and the UPnP Description view shows the contents on these description files. Thus without users knowledge the plug-in can get access to the Sindrion data using Eclipse adapter framework.

6.5 Sindrion Explorer

The Sindrion Explorer shows the hierarchy of the Sindrion elements in the workbench. It provides a Sindrion specific view of the resources, derived from the Sindrion DataModel. Each project shows the logical structure of the resources in its source folders tagged with Sindrion specific properties. Views must implement org.eclipse.ui.IViewPart interface or can extend the "org.eclipse.ui.ViewPart", and thus are the subclasses of "org.eclipse.ui.WorkbenchPart", inheriting much of the behavior needed to implement the IViewPart interface. In spirit of lazy initialization, the IWorkbenchPage holds on to instance of org.eclipse.ui.IViewReference rather than the view itself so that views can be enumerated and referenced without actually loading the plug-in defining the view. Firstly, an extension is added to the org.eclipse.ui.views extension point, to which a new category 'SindrionDEInit' is added. The new view is defined with identifier 'com.infineon.sindrion.sde.init.views.SindrionModelView' which comes into the above category. The 'com.infineon.sindrion.sde.init.views.SindrionModelView' class which will contain code to define the view's behavior is assigned to the view declaration. Listing 6.7 shows the view declaration after performing the above steps.

Listing 6.7: Declaration of Sindrion views.

```
<extension
      point="org.eclipse.ui.views">
   <category
         name="Sindrion Initialization"
         id="SindrionDEInit">
   </category>
   <view
         category="SindrionDEInit"
         class="com.infineon.sindrion.sde.init.views.
            SindrionExplorer"
         icon="icons/sindrion-logo.gif"
         id="com.infineon.sindrion.sde.init.views.
            SindrionExplorer"
         name="Sindrion Explorer"/>
   <view
         name="Sindrion Adapter"
         icon="icons/sindrion-logo.gif"
         category="SindrionDEInit"
         class="com.infineon.sindrion.sde.init.views.
            SindrionAdapterView"
         id="com.infineon.sindrion.sde.init.views.
            SindrionAdapterView"/>
</extension>
```

The Sindrion Explorer class extends the ViewPart abstract class. The *createPartControl*() creates the controls comprising the view. Then a tree viewer created using the given SWT style bits is added to view. The viewer should be provided with input, contentprovider, label provider, sorter and filters. The Sindrion DataModel is designed to as well serve as an input model for this viewer, and holds the Sindrion model objects. The ISindrionElement helps to abstract the differences between different types of Sindrion objects. A content provider links the objects created into the view. The 'SindrionModelView-ContentProvider' is responsible for extracting objects from the input, the Sindrion model, and handles them to the table viewer for displaying. This content provider is as well made responsible for updating the viewer of the changes in model, by implementing the 'TreeModelListener'. The 'SindrionModelView-LabelProvider' extends the 'LabelProvider' which takes the objects returned by the content provider and extracts the value to be displayed. A view action can appear as a menu item in a view's context menu, as toolbar button on the right side of the view's title bar, and as a menu item in a view's pull-down menu. The actions can be added to the view programatically, which can perform operations on the model objects like creating, deleting or modifying the information of the Sindrion objects. The *createPartControl()* is shown in Listing 6.8, adds the tree viewer to the view and this viewer is in turn set with content provider and label provider.

Listing 6.8: *createPartControl()* of SindrionModelView class.

```
public void createPartControl(Composite parent) {
   try {
      viewer = new TreeViewer(parent,
         SWT.MULTI | SWT.H_SCROLL | SWT.V_SCROLL);
      drillDownAdapter = new DrillDownAdapter(viewer);
      viewer.setContentProvider(
         SindrionModelViewContentProvider.getProvider());
      viewer.setLabelProvider(new
         SindrionModelViewLabelProvider());
      viewer.setSorter(new NameSorter());
      viewer.setInput(SindrionModel.getModel());
      SindrionModelViewContentProvider.getProvider().
         inputChanged(viewer,null,SindrionModel.getModel());
    } catch (Throwable e) {
      e.printStackTrace();
    }
 }
```

Figure 6.2 shows the Sindrion Model View which displays the datamodel of the Initialization plug-in.

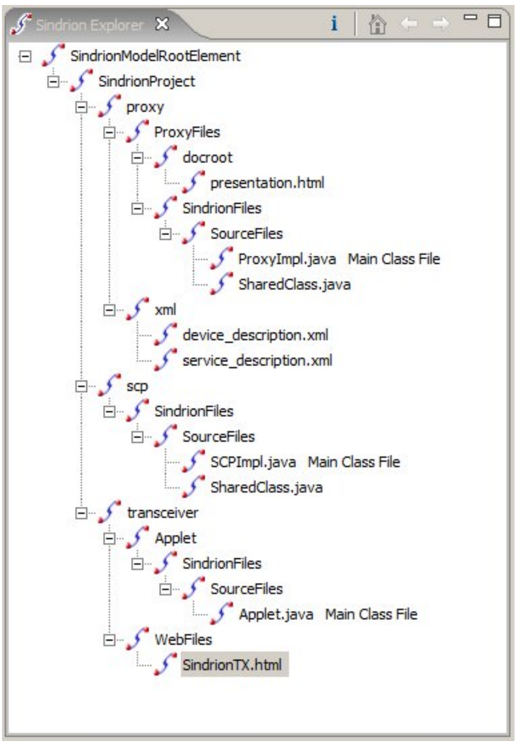


Figure 6.2: Sindrion Explorer

6.6 Sindrion Adapter View

Sindrion Integrated Development Environment also provides the Sindrion Adapter View which helps the developer for testing if a particular resource selected in the Package Explorer is a Sindrion element or not. The user may need to check this when he adds a new resource to and existing Sindrion project or wants to check if a particular project is a Sindrion resource or not.

The Sindrion Adapter View adds a selection listener to the workbench selection service for Package Explorer view, which notifies the view when the selection is changed in the Package Explorer. When the view is closed it removes the listener, since it no longer need to be notified of the selection.

Listing 6.9: Methods that add and remove the selection listener.

```java
private void addListener(){
  if(sl==null){
    selservice= getSite().getWorkbenchWindow().
       getSelectionService();
    sl = new PackageViewSelectionListener(this.viewer);
    selservice.addSelectionListener(JavaUI.ID_PACKAGES,sl);
    System.out.println("Package␣View␣Selection␣Listener␣Added")
        ;
    }
  }

private void removeListener(){
  if(sl!=null){
    selservice.removeSelectionListener(JavaUI.ID_PACKAGES,sl);
    System.out.println("Package␣View␣Selection␣Listener␣
        Disposed");
    }
  sl=null;
  }
```

The *selectionChanged*() method of the listener gets notified of the selection. The method requests the Sindrion adapter for this object and updates the view. The code which performs this operation can be seen in Listing 6.10.

Listing 6.10: *selectionChanged()* method of listener.

```java
public void selectionChanged(IWorkbenchPart part, ISelection
    selection) {
  try {
    Object selectedObject = ((IStructuredSelection)selection).
        getFirstElement();
    String className = "";
    if(selectedObject!=null){
      if(((IAdaptable)selectedObject).getAdapter(
          ISindrionElement.class) != null)
        className = ((IAdaptable)selectedObject).getAdapter(
            ISindrionElement.class).getClass().getName();
        ((SindrionExplorer.ViewContentProvider)(viewer.
            getContentProvider())).updateElement(((IAdaptable)
            selectedObject).getAdapter(ISindrionElement.class));
        }
    viewer.refresh();

    } catch (Throwable e) {
        e.printStackTrace();
    }
  }
```

6.7 Summary

This chapter gives the implementation details of the Initialization plug-in designed for the development of Sindrion Applications. This implementation proves the concepts designed to be functional. The Initialization process can be done conveniently and efficiently with the support from the development environment. The integration of the existing Implementation plug-in into the development environment using the support from DataModel also proves the concept of Sindrion Integrated Development Environment.

Firstly, the implementation details of the Sindrion nature used to identify the projects is given. The project creation wizard which configures and creates a Sindrion project in workspace along with required packages and skeleton resources is described. Then a detailed insight of the DataModel, how it is created, and listens to the workspace events to keep in sync with eclipse are given. The chapter proceeds with the implementation details of Adapter factory and how it can be used to convert objects from IResource.class and IJavaElement.class to ISindrionElement.class, which in turn is used as a mechanism to access the elements of DataModel. The internal mechanism of Sindrion Explorer and Sindrion Adapter view are given, which are used for visualization and testing purpose of the DataModel.

Conclusion and Recommendations

The goal of this master thesis was to design and implement the support for development process of Sindrion applications. With the implementation of the support for Initialization phase and a Sindrion DataModel this task has been addressed conceptually, and laid basis for the realization of Sindrion Integrated Development Environment.

In order to derive the general development steps and requirements for the development environment, a profound analysis has been conducted. The analysis indicated great potential for automation and moreover vindicates the creation of a sophisticated development environment for developing Sindrion applications. The conceptual design of an idealized design flow has been conducted as a result from the preceding requirements analysis. The main problems that needed to be solved were support for Initialization phase, track the dependencies between the Sindrion resources and support for the integration of development support. As a second step, existing stand-alone tools which support dedicated parts of the Sindrion software development process have been reviewed. Due to advantages concerning flexibility and extensibility, the Eclipse platform has turned out to be a suitable basis for the development environment. As the phases of development process share common data, mechanisms are to be generated to integrate them to form an IDE.

Firstly, the support for the initialization phase was designed, which provides an Eclipse wizard. The Sindrion project creation wizard helps the user to create and set-up the project with minimum effort. Next step was to integrate the stand alone tools that support the individual phases of development process. The Sindrion datamodel acts as a common data channel between the phases of development process thus serving the purpose of a basis for Integrated Development Environment. Mechanisms to access this data model were derived, that allow the plug-ins to share the common information. The issues that came up after the integration of the plug-ins viz. dependencies between the plug-ins and start-up, and initialization were also solved. In a concluding step, the concepts and existing technologies were brought together and the overall idea was refined up to an implementation state. Thus, the approach was not only proven to be valid and functional, moreover a concrete implementation example demonstrates the final system in action. Considering the Sindrion project to be a research project with continuous evolution changes to the developed tool set are very likely, hence the data model anticipates this requirement and includes abstraction layers that allow transparent modification of the workspace or programming language. The Sindrion Integrated Development Environment is based on a plug-in structure providing the ability to adapt to new middleware and programming languages.

Implementation of the most important components of Sindrion Integrated Development Environment specifically directed towards Java-based UPnP application development has proven flexibility and suitability of the concept. The results of the analysis enable further work to be done as the implementation part of this thesis focuses support for initialization phase and integration of the support for individual phases. The remaining parts that include the development of integrated debugging environment and mechanisms that deliver the applications are currently being implemented following the guidelines developed in this thesis.

Bibliography

[1] UPnP Forum. "UPnP Device Architecture 1.0, ver 1.0.1". http://www.upnp.org/download/UPnPDA10_20000613.htm, May 2003.

[2] Don Box, David Ehnebuske, Gopal Kakivaya, Andrew Layman, Noah Mendelsohn, Henrik Frystyk Nielsen, Satish Thatte and Dave Winer, editors. "Simple Object Access Protocol (SOAP) 1.1". http://www.w3.org/TR/2000/NOTE-SOAP-20000508, May 2000.

[3] Erik Christensen, Francisco Curbera, Greg Meredith, Sanjiva Weerawarana. "Web Services Description Language (WSDL) 1.1". http://www.w3.org/TR/wsdl.html, March 2001.

[4] Intel Corporation. "Intel Software for UPnP Technology - Technology Overview". http://www.intel.com/cd/ids/developer/asmo-na/eng/downloads/upnp/overview/index.htm, 2003.

[5] Yaron Y. Goland, Ting Cai, Paul Leach, Ye Gu, Shivaun Albright. "Simple Service Discovery Protocol/1.0 Operating without an Arbiter". http://www.upnp.org/download/draft_cai_ssdp_v1_03.txt, October 1997.

[6] R. Droms. "Dynamic Host Configuration Protocol". RFC 2131, March 1997.

[7] Stuart Cheshire. "Dynamic Configuration of IPv4 link-local addresses". IETF draft, November 2000.

[8] Eric Clayberg & Dan Rubel. *Building Commercial-Quality Plug-ins*, Addison Wesly, Boston, USA, 2005.

[9] Daniel Barisic. *Conceptual Design and Realization of a Development Framework for Smart UPnP Transceivers* , University of Dortmund, 2005.

[10] Stefan Budde. *Design and Implementation of a Development Environment for Smart UPnP Transceivers* , University of Dortmund, 2006.

[11] Jack W. Reeves. "*Article: What is Software Design?*", http://www.developerdotstar.com/mag/articles/reeves_design.html, 23 February 2005.

[12] "*Eclipse help system*". http://help.eclipse.org/help32/index.jsp .

[13] "PDE Does Plug-ins". http://www.eclipse.org/articles/Article-PDE-does-plugins/PDE-intro.html.

[14] J. Cohen, S. Aggarwal, Y. Y. Goland. "General Event Notification Architecture Base: Client to Arbiter". http://www.upnp.org/download/draft-cohen-gena-client-01.txt, September 2000.

[15] "Eclipse Platform Technical Overview". http://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.html.

[16] "Eclipse Plug-in Architecture". http://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.

[17] The OSGi Alliance. "OSGi Service Platform, Release 3". IOS Press, 2003.

[18] Robert Harris, Rob Warner. "The Definitive Guide to SWT and JFACE". Apress, 2004.

[19] "When does a plug-in get started?" http://wiki.eclipse.org/index.php/FAQ_When_does_a_plug-in_get_started%3F.

[20] "Can I activate my plug-in when the workbench starts". http://wiki.eclipse.org/index.php/FAQ_Can_I_activate_my_plug-in_when_the_workbench_starts%3F.