DEPARTMENT OF COMPUTER SCIENCE, IOWA STATE UNIVERSITY

Programming Manual for the ISU Smart Home

Version 1.2

Ryan Babbitt

2/21/2011

Table of Contents

1. Introduction to the ISU Smart Home Lab 5

1.1 What is a Smart Home? 5

1.2 What makes a Smart Home "smart"? 5

1.3 How does one program a smart home? 5

1.4 Can I use Smart Home Lab software or develop my own applications or become more involved? 6

2. Installation of the Lab Environment on a Local Machine 7

2.1 Overview 7

2.2 The Lab Environment 7

2.3 Installation Alternatives 8

2.3.1 Alternative #1: Setting up the lab environment on a Virtual Machine 9

2.3.2 System Requirements for Alternative #1 9

2.3.3 Alternative #2: Installing Components Individually 11

2.3.4 System Requirements for Alternative #2 11

3. Knopflerfish OSGi Framework Overview 13

3.1 The Knopflerfish Desktop 13

3.1.1 The Main Menu Bar 14

3.1.2 Bundle Toolbar 14

3.1.3 Bundle and Bundle Information Windows 15

3.2 Anatomy of an OSGi Bundle 15

3.2.1 Service Interface 16

3.2.2 Service Implementation 16

3.2.3 Bundle Activator 16

3.2.4 Bundle Context 16

3.2.5 Manifest File 16

3.2.6 Other Resources 17

3.3 Bundle Management in OSGi 17

3.3.1 Bundle Lifecycle – Bundle States 17

3.3.2 Observing and Changing Bundle State 18

3.3.3 Bundle Lifecycle - Bundle Events 18

3.3.4 Observing Bundle Events Graphically 20

3.3.6 Bundle Dependencies 20
3.3.7 Observing and Refreshing a Bundle's Dependencies 20
3.3.8 Refreshing Bundle Dependencies 20
3.4 Service Management in OSGi 21
3.4.1 Service Lifecycle – Service States 21
3.4.2 Service Lifecycle – Service Events 21
3.4.3 Service Programming Tasks for Providers 22
3.4.4 Service Programming Tasks for Clients 23
4. Tutorial Part 1: Developing A Simple Service 25
4.1 Introduction to the PopupDisplayer Service 25
4.2 Checking out a Bundle Project from the Repository 25
4.3 Browsing the Service Interface 26
4.5 Verifying the Manifest File 26
4.6 Implementing the Activator Class 28
4.7 Executing the Service 28
4.7.1 Deploying the New Bundle 28
4.7.2 Interacting with the Bundle 29
5. Tutorial Part 2: Creating a Composite Service 31
5.1 Checking out the WebCameraService Bundle 32
5.2 Inspecting and Installing the WebCameraService Bundle 32
5.3 Checking out the CameraViewerService Bundle 32
5.4 Inspecting the CameraViewerService Interface and Implementation 32
5.5 Implementing the CameraViewerService Activator 35
5.5.1 Implementing the stop() Method 36
5.5.2 Implementing the stop() Method 36
5.5.3 Add error checking for NULL service references 36
5.6 Inspecting the Manifest 36
5.7 Deploying the Bundle 36
6. Tutorial Part 3: Creating an Application Bundle 37
6.1 Creating the Main Project Folder 37
6.2 Creating the Remaining Directory Structure 37
6.3 Editing the Bundle Packaging 37

6.4 Implementing the Activator Class 38
6.4.1 Declaring Member Variables. 38
6.4.2 Implementing the start() method. 38
6.4.3 Implementing the stop() method. 38
6.5 Deploying the Application Bundle 38
7. Glossary of Terms 40

1. Introduction to the ISU Smart Home Lab

1.1 What is a Smart Home?

The term “smart home” carries different implications depending on the research institution which is using the term. Typically, however, the concept carries with it the idea of embedding of computer technology in “normal” living spaces to provide some benefit to the inhabitant(s) of that space. For example, in the case of illness a smart home can enable the monitoring of a health condition and network the home with doctors or emergency personnel. In the case of someone with cognitive or physical disabilities, a smart home can provide some support for various activities of daily living, such as cooking or controlling electronic appliances. Ultimately, though, the goal of a smart home is to enhance independence and improve the quality of life of its inhabitants. In the case of elderly persons, we also speak of “aging in place” as an alternative to a group care facility.

1.2 What makes a Smart Home “smart”?

Smart homes make use of different hardware devices and software services to monitor and control the physical environment. *Sensors* provide the home with data about some thing, person, or situation in the home. They serve as inputs to get data about the physical world to the system. *Actuators*, on the other hand, allow the computer to act upon objects in the environment, such as controlling electronic appliances, sending Infrared commands, or moving something via a motor. A software *service* is a invocable, reusable software artifact with a well-known interface that provides some sort of functionality, such as controlling a hardware device, accessing a local application, computing some results, etc. Services may also be composed from other existing services, making use of functionalities already provided. The idea of reusing existing services to streamline design is a major emphasis of the service-oriented paradigm and one of the reasons why it is becoming so popular. A *service-oriented architecture (SOA)* is a software framework that manages a collection of collaborating services as they are added and removed from a system. This list of services in the system and their properties is maintained in a *service registry*, which can be searched to obtain a handle for specific services (i.e. information necessary for invoking the service). The process of searching for a service and obtaining a handle for it is called *binding*, whereas *invocation* is the actual calling of an operation on the service. The concept of a registry is also central to SOA because service-oriented systems are expected to be very dynamic, meaning that the collections of the services in the system may frequently change. This makes SOA a common paradigm for smart homes since the needs of an inhabitant change over time and may change rapidly.

1.3 How does one program a smart home?

Broadly speaking, there are three major considerations to designing and developing software for smart homes applications: knowing existing resources, knowing the development process, and knowing the needs of end users.

First, in order to design any meaningful software, one has to know the existing hardware and software resources in a system and what they are capable of doing. These are the building blocks out of which more complicated things can be built. In some cases, developers may be introducing new low-level services to interface with some new hardware device (i.e. sensor or actuator), but most often, a developer will be introducing a new service that is composed by adding some meaningful application logic to use already existing services. We call such a service a *composite service* and the reused services it is composed from *component services*. Developing a composite service involves being able to think through the flow of the service in a structured manner, noting how component services will be invoked and handling possible error conditions (remember SOAs can be very dynamic). Sections 4 and 5 walk the reader through creating a simple service and a composite service respectively.

Secondly, one should have an appropriate development environment and know the basic processes, standards, and conventions followed so that new software may conform to the assumptions and requirements of the target system. The existent software in the lab has been developed with a certain structure and organization and since it is the goal that newly developed services eventually be deployed in the actual smart home lab, any software that does not satisfy these assumptions and requirements may not run correctly. Section 2 discusses how to setup a development environment similar to the one in the smart home lab and the various standards and conventions used are explained in the service tutorials.

Lastly, and most of all, one needs to have a firm grasp of what the actual needs of end users are. It is not enough just to design and implement *something*. The new service or application should actually be *useful*, *beneficial*, and *usable* to potential inhabitants of the smart home. It should meet *their* needs and solve *their* problems. The primary goals of smart home technology are to improve the quality of life of its inhabitants and enable living independently for aging in place, and software that is developed without taking user problems/needs into heavy consideration will not reach these goals. On the contrary, it will be a cumbersome burden to its users and quite possible ignored or rejected. The lab is working in conjunction with gerontology program[1] to identify these needs and problems from the target population.

1.4 Can I use Smart Home Lab software or develop my own applications or become more involved?

In the next few chapters, we will walk you through the steps needed to download and setup the necessary developing environment and provide a basic tutorial on how to develop software for an environment similar to the Smart Home Lab. For more information about the lab's software platform, existing hardware and software resources used in the lab, or to become more involved with the lab, contact the lab's chief architect Ryan Babbitt at rbabbitt@iastate.edu.

[1] http://www.gerontology.iastate.edu

2. Installation of the Lab Environment on a Local Machine

2.1 Overview

This section walks the reader through the installation and setup of the smart home lab environment on a local or personal computer. Our use of the term *environment* comprises all of the software and supporting files necessary to develop, deploy, and execute a smart home service or application in the lab. Without duplicating all of these supporting components, software that we have developed is not guaranteed to run correctly on your local system, and software you develop is not guaranteed to run correctly in our system. With that being said, there are two different options for setting up of a copy of our lab environment on a local machine. The first uses the concept of *virtual machines* to run an exact replica of our lab software on a local machine in a way that is completely separate from the local machine. The second option is more labor intensive with each software component being individually installed directly on the host machine. After describing the lab environment and listing the basic system requirements to support it, each installation option is covered.

2.2 The Lab Environment

The primary components of the lab environment are the Java-based Service-Oriented Architecture OSGi [2] and XML-based Web Services (WS). Properly speaking OSGi is a *centralized* SOA, meaning that all the services in the system are under the management of one entity and all services are executed locally. This managing entity is known as the *OSGi Framework*, and all services that are added, removed, executed, or invoked are done so with this single Framework. The binding and invocation processes are fairly simple as the Framework simply provides methods to obtain references to services objects, which are essentially just a (locally executing) Java object. Web services, on the other hand, present quite a different case. They are de-centralized, meaning that the services are (usually) under the management of different entities and are executed remotely. Thus, binding to and invoking a web service is more complicated because network communication is needed. Each service provider interested in offering services must register their service interfaces and all necessary binding information, such as the URL of the service, at a common, well-known registry. Every service consumer interested in using a particular type of web service queries the common registry to obtain binding information of some instance of that service and then invokes that service over the network using standardized protocols[3]. Web services also require a *web service container* at the service producer that provides the runtime environment for web services. It receives service requests, manages the invocation and execution of the actual services, and returns the responses over the network.

Strictly speaking, both OSGi and WS are only *specifications*[4] of SOAs, meaning they only define the set of interfaces to be used in implementations and what those methods should do, but they do not provide any implementations themselves. There are several implementations of

---

[2] The Open Service Gateway Initiative, http://www.osgi.org
[3] Web services communication protocols are defined over HTTP with XML payloads
[4] The OSGi spec is maintained by the OSGi alliance, and the WS spec is maintained by W3C.

OSGi frameworks and WS containers. The OSGi implementation is an open source distribution called Knopflerfish.[5] The Knopflerfish framework itself is developed in Java version J2EE 1.5, and thus executes in a Java VM. We assume that the reader is familiar with programming in this version of Java. The WS containers that we use is Apache Axis[6] and the HttpService[7] provided by OSGi. The former is for hosting web services outside of OSGi, and the latter is for hosting web services from within OSGi. Chapter 3 gives more information about programming with OSGi services their lifecycle, and chapter 4 discusses the integration of web services with OSGi.

In addition to OSGi and WS, the execution environment contains several other software artifacts. Most notably, MySQL[8] is used as our primary database application, and several vendor provided software libraries serve as low-level APIs for the various types of hardware devices used in the lab. These types of devices are discussed in Chapter 5.

Since both OSGi and our WS container implementations are based on Java, so is the development environment. In addition to developing code, the development environment is also responsible for managing the building process (i.e. compiling source files and other deliverables), as well as unit testing. We develop software using Eclipse IDE 3.4.2[9] to create and manage our software projects. This version of Eclipse has several features related to web services such as wizard to create web services, an editor for web service interfaces, an explorer to find web services and manually execute their methods, and a code generator for clients.

The basic Eclipse distribution is augmented with a few plug-ins to provide the extra functionality we need. The central plug-ins are the Knopflerfish plug-in[10] to manage OSGi bundles, the Visual Editor plug-in to manage GUI design[11], and the Subversion plug-in Subclipse[12] for version control. Again, section 2.5 discusses the installation of the development environment in detail. Using this version of Eclipse with the Knopflerfish plug-in, there are two ways of building projects. The Knopflerfish plug-in has a feature that uses the Ant[13] build system to automatically re-build projects when source files are changed, but a secondary Ant build file can also be used for more fine-grained control over the build process. This manual will assume that the Eclipse plug-in has been installed, and comments on how to use the Ant build system in a customized manner can be found at the Ant website. Unit testing of OSGi services is accomplished via some special services provided by Knopflerfish, and testing of WS is mainly done using Eclipse web service explorer manually execute web services and observe their results.

2.3 Installation Alternatives

This section describes two options for installing the smart home lab environment on a local machine. All remaining sections of this manual assume that the environments are installed

[5] See http://www.knopflerfish.org
[6] See http://ws.apache.org/axis/
[7] See http://www.osgi.org/javadoc/r4v42/org/osgi/service/http/HttpService.html
[8] See http://www.mysql.com
[9] See http://www.eclipse.org/downloads/packages/release/ganymede/sr2
[10] See http://www.knopflerfish.org/eclipse_plugin.html
[11] See http://www.eclipse.org/vep/
[12] See http://subclipse.tigris.org/
[13] See http://ant.apache.org/

correctly, and that the reader is familiar with components in it such as Java, SVN version control, and MySQL. Links to additional tutorials on these topics can be found at their websites. Questions or problems in the installation process should be directed towards the chief architect, Ryan Babbitt, at rbabbitt@iastate.edu.

2.3.1 Alternative #1: Setting up the lab environment on a Virtual Machine

There are two different ways to install a copy of the smart home lab environment. The first is simple and makes use of the concepts of virtualization and virtual machines to provide a virtual copy of the lab environment, and the second is the manual installation of each individual software component to reproduce the actual lab environment on a target machine.

Virtualization is a technique in which one computer, the *host operating system*, has special software that can provide containers, or *virtual machines*, for other systems, called *guest operating systems*, to run. The host provides a software API that mimics the hardware API to each guest system, such that each guest operating system executes as though it is communicating with the host's actual hardware. The virtual hardware (e.g. CPU, memory, I/O devices) is actually made up of processes/threads running on the host machine, and the state of the virtual machine (i.e. its CPU registers, device registers, memory contents, and hard disk) are stored in the host's memory. Each guest system is isolated and independent from other guest systems as well as from the host system, making virtual machines a very safe way of testing software.

The virtualization software that we are using is VMWare Server 2.0[14]. The installation of the virtual lab environment requires three steps: 1) installing the VMWare virtualization server on your local machine, 2) creating a virtual machine on which to do the development, and 3) installing the image of the lab environment on the new virtual machine. No additional software needs to be installed because all of the necessary applications and files are stored in the provided image. However, you will need access to a Windows XP/Vista installation CD and have administrator privileges on the host machine.

2.3.2 System Requirements for Alternative #1

- Available hard drive space – The host machine will require at least 11 GB of free hard drive space to install the virtualization software and the lab's virtual machine.
- Working internet connection – Some services will be running on remotely on lab machines, so a working internet connection is required.
- USB or Serial Port – Some devices require a USB or a RS-232 Serial port to interface with the computer. If a USB device (e.g. Phidgets) or serial device (e.g. Insteon) is going to be used on the host machine, then the host machine will need these ports.

[14] https://vmware.com/products/server/

- Operating System – Any operating system supported by VMWare will be sufficient on the host machine, but a Windows XP Setup disk is required to install Windows XP on the guest system.

To install VMWare Server 2.0:

1. Download VMware Server 2.0 (http://www.vmware.com/products/server/). If you do not have an account established with VMware, then you will need to create one.

2. The license key required to install VMware Server will be provided to you on the download page sent in the reply email.

3. Install VMware Server. You may leave all options default. Take note of the directory that will contain virtual machine images. By default, this is C:\Virtual Machines, and make sure the drive has enough free space. An icon should be installed on the desktop for the server application.

To Install a Virtual Machine running Windows XP:

1. After installing VMware Server and rebooting your computer, access the management interface (Click Start --> Programs --> VMware --> VMware Server --> VMware Server Home Page). Alternatively, you can access the management interface by opening https://localhost:8333/ in your browser or double-clicking the icon on your desktop

2. Log in with Windows account that has administrator privileges on the host system.

3. Create new virtual machine:
   3.1. Click Virtual Machine --> Create Virtual Machine
   3.2. Follow the wizard to select hardware parameters for new virtual machine.
   3.3. Use default settings unless you want to change specific virtual hardware parameters, such as RAM or hard drive size.

4. Before "powering on" your virtual machine, make sure Windows XP installation CD is in the CD-ROM drive.

5. After the virtual machine has been powered on, select it from the Inventory list on the left side of the management interface, and click Console tab.

6. Click on the black area inside of this tab to establish a console connection to the virtual machine. If this is the first time you use VMware Console, your browser will prompt you to install the VMware tools plug-in. Install the plug-in, restart your browser,

and reconnect to the management interface. Repeat Steps 5 and 6 again to open the console connection.

7. Install the operating system on the virtual machine using a setup disk

To install the lab environment image:

1. Download the “rav4-VM-image” from the [WWW] and place it into the “Virtual Machines” directory in the installation directory (e.g. C:\Virtual Machines)
2. Start the VMWare server application via the desktop icon or its web interface.
3. Configure the newly installed virtual machine by selecting it in the “Inventory” Pane on the left-hand side of the screen and selecting the “Summary” tab in the center of the screen.
    a. The network adapter should be in “Bridged” mode.
    b. All necessary USB devices should be passed through to the guest system by clicking on the purpose USB icon in the menu bar of the webpage and selecting the necessary devices.
4. Execute the virtual machine by selecting it in the inventory and clicking on the green “play” icon.

2.3.3 Alternative #2: Installing Components Individually

If you choose not to install the environment using virtual machines, you may also install each component individually. This amounts to 1) installing OSGi and the supporting bundles for executing services, 2) installing Eclipse and the requisite plug-ins for developing services, and 3) installing some third party device libraries and database software.

2.3.4 System Requirements for Alternative #2

The system requirements for alternative #2 are mostly the same as for alternative #1, except the significantly less memory is required.

To Install the Java 1.5 JDK:

1. Download the JDK installer file (http://java.sun.com/javase/downloads/index_jdk5.jsp). Run it, and follow the instructions.

To install the Eclipse IDE and requisite plug-ins:

1. Download and install Eclipse Ganymede for J2EE from `http://www.eclipse.org/downloads/packages/eclipse-ide-java-ee-developers/ganymedesr2`
2. Use the update-site to install the Knopflerfish plug-in from `http://www.knopflerfish.org/eclipse-update/`
3. Use the update-site to install the Subclipse version control plug-in from `http://subclipse.tigris.org/update_1.6.x`

4. Use the update-site to install the Android mobile phone simulator plug-in (if desired) from `https://dl-ssl.google.com/android/eclipse/`

To Install Knopflerfish OSGi:

1. Download the Knopflerfish Framework jar file from `http://knopflerfish.org/releases/2.2.0/knopflerfish_osgi_2.2.0.jar`
2. Double-click on the jar file and follow instructions.

To Install ISU Smart Home Lab bundles:

1. Determine the bundles necessary for present purposes (See related documentation on "Resources and Applications in the ISU Smart Home Lab") and their URLs.
2. Run the Knopflerfish framework by double clicking the framework jar file.
3. Click on the "Open Bundle Location" button on the top left of the Knopflerfish console and enter a bundle URL. Be sure to install each bundle after ALL component bundles have been installed. These dependencies are noted in section 8.4 and 8.5
4. Repeat that process for these bundles as well:

To install third party device libraries and database software:

Phidgets Sensors and Actuators (USB Devices)

1. Download the Phidgets installer from `http://www.phidgets.com/drivers.php`
2. Double-click on the .msi installer and follow the system prompts
3. Copy the phidgets.jar file to the /lib/ext directory of the JDK installation
4. Configure the Phidget21Manager service by running the executable in the directory specified.

Tira for IR Control (USB Device)

1. Download the Tira 2 Windows Drivers from `http://www.home-electro.com/download/TiraLatestDrivers.zip`
2. Unzip the folder and execute DPInst.exe.
3. Follow the system instructions.

X10/Insteon for Appliance and Lighting Control (Serial Device)

1. Install the javax.comm package for serial port communications from `http://llk.media.mit.edu/projects/picdev/software/javaxcomm.zip`
2. Follow the instructions at the website

a. Copy the **win32com.dll** to the \bin in JAVA_HOME's directory. (i.e. C:\Program Files\Java\jdk1.5.X\jre\bin)
b. Put the files **comm.jar** and **javax.comm.properties** in JAVA_HOME's \lib\ext directory. (i.e C:\Program Files\Java\jdk1.6.0_05\jre\lib\ext\)

MySQL database server:

1. Download the MySQL 5.1 installer from `http://dev.mysql.com/downloads/mysql/`
2. Follow the prompts.
3. Download the MySQL GUI tools 5.0 installer from `http://dev.mysql.com/downloads/workbench/5.2.html`
4. Follow the instructions.

Android Mobile Phone SDK:

1. Install the Android X.X SDK as described in `http://developer.android.com/sdk/index.html.`

3. Knopflerfish OSGi Framework Overview

This section provides an overview of the important features of the OSGi specification and about the Knopflerfish Desktop, a graphical user interface for user control the Knopflerfish OSGi implementation.

3.1 The Knopflerfish Desktop

The Knopflerfish implementation of OSGi comes with a very nice graphical user interface that displays important information about bundles and allows for control of bundles. The main features of the GUI are described below.

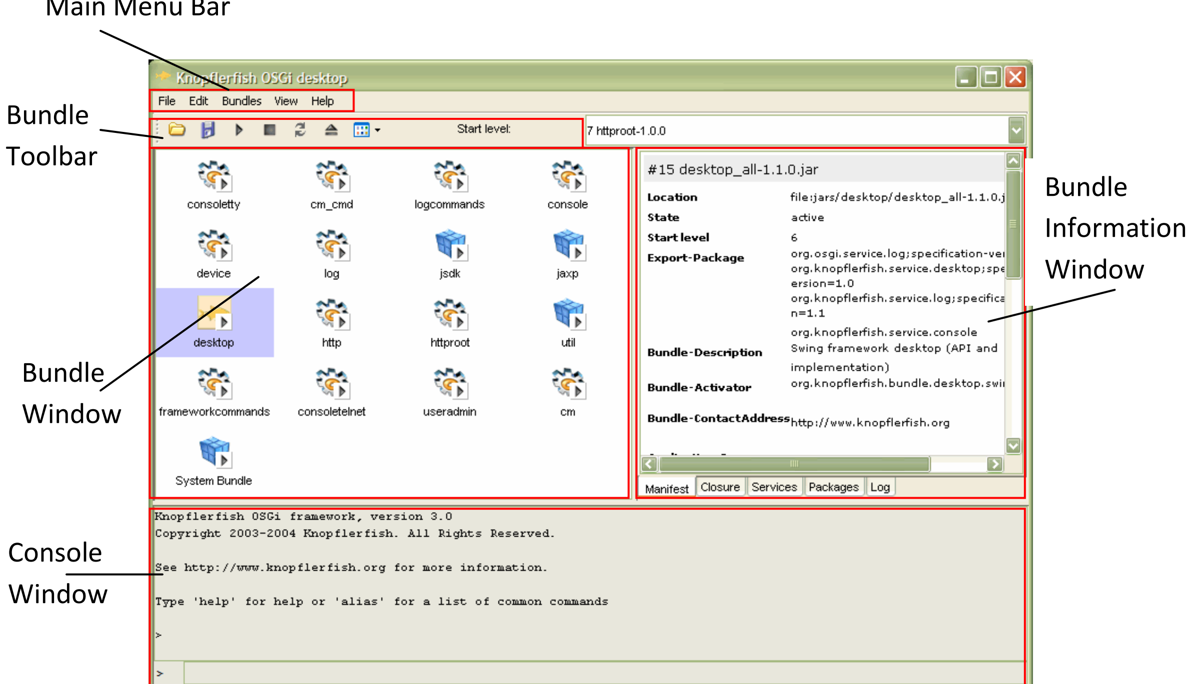


Figure 1 The Knopflerfish Desktop

3.1.1 The Main Menu Bar

Like all applications the Knopflerfish Desktop has a Main Menu that provides different options for user interaction with the Framework.

The **File Menu** allows a user to install a bundle from the local file system using a file open dialog, install a bundle from a url, create an archive of the framework, and quit the framework.

The **Edit Menu** allows a user to select or unselect a bundle or clear the console. Once a bundle is selected important information about its manifest, dependencies, and state is viewable in the bundle information window.

The **Bundles Menu** allows a user to start, stop, update, or uninstall a bundle. The effects of these functions are described in section 3.3. Also it allows the user to force a refresh of all bundles manually, which may be needed in certain circumstances when updating a bundle.

The **View Menu** allows for the customization of the desktop by allowing the user to select which windows and controls are visible.

Lastly, the **Help Menu** shows some framework information (but no manual!)

3.1.2 Bundle Toolbar

The bundle toolbar essentially defines quick access to several important menu functions, such as installing a bundle from the file system or URL, starting, stopping, updating, or uninstalling a bundle. This toolbar also contains the options for the mode of graphical display of bundles in the bundle window.

3.1.3 Bundle and Bundle Information Windows

The Bundle Window shows the bundles in a framework graphical for easy visualization of state and type of the bundle. The Bundle Information Window shows various facts about a bundle's state, dependencies, and execution history. It has a tabbed pane that can be used to select the type of information displayed.

The **Manifest tab** shows the headers and values of the bundle's manifest file which essentially describes the bundle's important properties.

The **Packages, Services, and Closure tabs** all give information about service dependencies. The **Packages tab** providers the list of imported and exported packages defined in the manifest. The **Services tab** gives the list of services currently imported and exported by an executing bundle. Each service has a unique id number that may be clicked on to see the service objects interface(s) and property values. The **Closure tab** provides both the package and service dependencies.

The **Log** and **Events tabs** provide information about the execution history of the bundle. The **Log tab** displays the times and types of various information messages about major events in the selected bundle's lifecycle and in the lifecycles of the services it hosts since the framework's start up. The **Events tab** shows similar information about all bundles in the framework and provides a simple interface to create filters for viewing different subsets of bundle/service events.

Console Window

The Console Window displays messages redirected from the standard output and standard error streams of executing bundles in the large upper box, and provides a command line interface for interacting with the framework in the smaller lower box.

3.2 Anatomy of an OSGi Bundle

Recall that a service is a reusable, invocable software artifact that conforms to a well known interface. The two parts of any service are its *interface*, defining the operations that the service provides, and the *implementation*, providing the code that realizes the interface. In OSGi, service interfaces, their implementations, and a few other supporting resources are packaged into what is known as a *bundle*, which essentially is a jar file with a special *manifest* describing its contents and properties. This section gives a brief overview of the anatomy of these parts of an OSGi bundle.

3.2.1 Service Interface

The operations that a service provides need to be well-defined and well known so that services may be invoked in a meaningful way. Just like a Java interface, a Service Interface defines a contract between a service provider and a service consumer about how the service can be used. In OSGi, a service interface is just defined as a regular Java interface. More than one interface file may be needed to define a service if a method argument is not a basic data type such as an Integer, String, Boolean, etc.

3.2.2 Service Implementation

The interface must also have an implementation in order to be executed at run time. Like in Java, the class implementing an interface must explicitly declare this to the Java compiler with the **implements** keyword. The class body is then implemented just as a normal Java class implementation.

3.2.3 Bundle Activator

In order for the OSGi framework to interact with the bundle, a special class called the *bundle activator* needs to be implemented. This class implements the *org.osgi.framework.BundleActivator* interface, which provides two important methods for managing the service: *start()* and *stop()*. The *start()* method is responsible for retrieving resources from the bundle and the framework, whether they be files or other services, registering resources with the framework, and executing the service implementation. The *stop()* method is responsible for releasing and unregistering resources. Only one activator should be implemented per bundle.

In some cases, bundles can be implemented without activators. These bundles are known as *library bundles* because they provide Java class files to the framework and other bundles. Bundles with Activators are shown pictorially in the bundle window as gears ( ) and library bundles are represented as blue cubes ( ).

3.2.4 Bundle Context

The Bundle interface to the OSGi framework is given through the *org.osgi.framework.BundleContext* class. Both the *start()* and *stop()* methods are called with this as the sole argument. It is the *BundleContext* that provides the methods that the Activator invokes to register, unregister, and obtain services and resources.

3.2.5 Manifest File

Each bundle jar file also has a special header file called, the manifest file, that defines important properties about the bundle. Some of the important properties that a bundle developer needs to know about are as follows:

- The *Bundle-Activator* Header specifies the full path name of the bundles activator class.

- The *Bundle-Classpath* header gives the names and paths of resources, such as files that are internal to the bundle.

- The *Bundle-Name* header gives a name for the bundle which will be displayed by the Framework.

- The *Import-Packages* header gives a list of packages that will be imported from the OSGi Framework. Package names are given their full path and separated with commas.

- The *Export-Packages* header gives a list of packages that will be provided to the OSGi Framework for other bundles to use. Package names are given their full path and separated with commas.

- The *Bundle-SymbolicName* header gives the internal name that the Framework will use to refer to the Bundle. It should be package-delimited

There are many other well-defined manifest headers[15], and it is even possible to define other custom headers for use within your OSGi bundle.

3.2.6 Other Resources

Other types of resources such as images files , libraries, or text files may also be stored in a bundle and accessed by the bundle's activator via the *getClass().getResourceAsStream(String x)*, where x is the path name of the resource as it was packaged into the bundle. This method returns an *InputStream* object that may be used to read the resource's contents.

3.3 Bundle Management in OSGi

An OSGi bundle is a collection of resources, services, and applications that will be managed by the OSGi Framework. There are several important concepts to be aware of in the context of bundle management, namely managing the bundle lifecycle, with its states and events, and managing bundle dependencies.

3.3.1 Bundle Lifecycle – Bundle States

An OSGi bundle can take on one of several states depending on its relation to the Framework and what part of the bundle is executing.

[15] See http://www.osgi.org/javadoc/r4/org/osgi/framework/Constants.html

- INSTALLED – The bundle's jar file is known to the framework and framework resources have been allocated to the managing the Bundle.

- RESOLVED – The bundle's static dependencies (i.e. the Java packages it imports) are all present in the framework and consistent with the bundle's expected versions.

- STARTING – The Activator's *start()* method is executing.

- ACTIVE – The Activator's *start()* method has successfully finished executing and any service it registers can be invoked.

- STOPPING – The Activator's *stop()* method is executing, the service can no longer be invoked. After executing this method, the bundle returns to the RESOLVED state.

- UNINSTALLED – The bundle has been uninstalled from the framework but some other service has a lingering reference to some class provided by the bundle, preventing the bundle from being completely removed.

3.3.2 Observing and Changing Bundle State

The Knopflerfish framework provides two ways of visually observing a bundle's state. If the bundle is installed but not active it shows up as a gear with the name of the bundle below it. If the bundle is active, then it has a small square with a triangle ( ). Otherwise, the state of the bundle can be read by selecting the bundle with the cursor and selecting the Manifest Tab of the Bundle Information Window. It may be necessary to temporarily select another bundle and reselect the original bundle to see the upated state. A bundle's state can also be queried programmatically with the *Bundle.getState()* method.

Similarly, a bundle's state can be changed by using the Framework GUI or programmatically. There are buttons in bundle toolbar to install a bundle via a file dialog (the folder icon) or URL (the gear icon), start a bundle (triangle icon), stop a bundle (square icon), update a bundle (refresh icon), or uninstall a bundle (eject icon)**.** Bundles can be installed programmatically with the *BundleContext.installBundle()* method, which takes a string argument of a URL path of the bundle. Bundles can be started, stopped, updated, and uninstalled with the method with the same name of the *Bundle* class.

3.3.3 Bundle Lifecycle - Bundle Events

Based on the states defined above, a bundle may undergo several types of state changes when the bundle transitions from one state to another. Such transitions are called *bundle events* and indicate something meaningful has happened to the bundle.

Bundle events cannot be observed visually as such, but the Framework does record bundle events in the **Log** and **Events** tabs of the Bundle information window. Below is a list of the commonly recorded Bundle Events. Note that most events correspond with a bundle state of the same name.

- BundleEvent.INSTALLED - a transition to the INSTALLED state indicating that the Bundle has been installed in the Framework This is the initial state of a bundle.

- BundleEvent.RESOLVED – a transition from INSTALLED to the RESOLVED state indicating that the Bundle's static (i.e. compile-time) dependencies have been successfully resolved (see 3.3.6)

- BundleEvent.STARTING – transition from RESOLVED to STARTING indicating that the Activator's *start()* method has been invoked

- BundleEvent.STARTED – a transition from STARTING to ACTIVE indicating that the Activator's *start()* method has finished executing and the bundle's services can be invoked

- BundleEvent.STOPPING – a transition from ACTIVE to STOPPING indicating that the Activator's *stop()* method has been invoked

- BundleEvent.STOPPED – a transition from STOPPING to RESOLVED state indicating that the Activator's *stop()* method has finished execution and its services cannot be invoked

- BundleEvent.UNRESOLVED – a transition to the INSTALLED state indicating that a Bundle's bound dependencies have been released. This transition only happens when updating or uninstalling a Bundle. The Bundle's resources are no longer obtainable within the Framework.

- BundleEvent.UNINSTALLED – a transition to the UNINSTALLED state indicating that the Bundle is scheduled for removal, pending all references to it are deleted. The Bundle's resources are no available, and the Framework will soon deallocate the resources it has provided to manage the Bundle

- BundleEvent.UPDATED – an event indicating that the Bundle has been updated. This is a complex event involving 1) stopping the bundle if it is active, 2) unresolving its dependencies, 3) reloading the new bundle code and resolving its dependencies again, and 4) starting the bundle if it was active.

3.3.4 Observing Bundle Events Graphically

As previously mentioned, most events in a Bundle's execution can be observed by selecting the Bundle icon in the Bundle Window and selecting the **Log** tab in the Bundle Information Window. Each record displayed by this tab includes a globally unique identifier for the Bundle Event, a globally unique identifier for this particular bundle, the time of the Bundle Event, the type of Bundle Event, and any error messages generated by the Bundle Event.

3.3.5 Observing Bundle Events Programmatically

A service, Activator or other Java object can handle Bundle events by implementing the *org.osgi.framework.BundleListener* interface and invoking the *BundleContext.addBundleListener(…)* method with self as the parameter. This allows the specified object to receive notifications of Bundle Events and take the appropriate actions to deal with them.

3.3.6 Bundle Dependencies

Most of the time, a service is not developed in isolation. It makes use of other already existing services and resources, resulting in dependencies between one or more bundles. A major part of managing the lifecycle of a service is keeping track of its dependencies. When a bundle is installed, several dependencies need to be resolved. Among other things, this includes the bundle classpath as specified in the manifest, all imported and exported packages. The bundle's classpath and exported packages are resolved internally with the bundle, but when the Bundle requires a package from another bundle, it must be imported from another bundle. If some other bundle exists that exports that package, the OSGi framework establishes a *static dependency* between the two bundles. It is called static because it is determined at compile time before the bundle code executes. On the other hand, while it is executing, the bundle may also dynamically obtain a reference to a service or resource provided by another bundle creating a *runtime dependency*. It is called runtime because the bundle code is executing.

3.3.7 Observing and Refreshing a Bundle's Dependencies

Both the static and runtime dependencies of a bundle can be viewed by selecting the bundle of interest in the bundle information window and selecting the **Closure tab** of the Bundle Information Window. This displays a list of class definitions that are imported from other packages (static dependencies) and a list of service objects currently referenced by the bundle(run-time dependencies). To see the full package names of the classes and objects and the bundles the dependencies have been resolved to, select the **Packages** or **Services tab**, respectively.

3.3.8 Refreshing Bundle Dependencies

Normally, when a bundle is updated, the Framework will (after the Bundle's static dependencies have been resolved) replace all of the class definitions of exported

packages with their new versions. However, if another bundle still has a reference to an object provided by that bundle, then the old package provided by the bundle is not updated and the old service/object is still used. This is somewhat of a necessity to maintain the consistency of the depending Bundles. However, an update of the package dependencies within the framework may be forced by selecting the Bundles -> Refresh Bundle Packages option from the Bundle Toolbar. This has the effect of stopping, and un-resolving, re-resolving, and starting any bundle that is currently exporting a package from an updated bundle. Any pending uninstalled bundles are also removed from the framework because their outstanding references have been released.

3.4 Service Management in OSGi

A bundle can be viewed as a structured container for service implementations. It holds the manifest file, all of the class files (both interfaces and implementations), and all related resources that the bundle provides in a way that the OSGi Framework can interact with and manage the Bundle. This management activity produced the ideas of a bundle lifecycle and static dependencies as discussed above. However, bundle management is only one aspect of the OSGi Framework's responsibilities. It is designed more importantly to manage the services that are offered from among the set of active bundles and the run-time dependencies that exist among bundles and services.

As with a Bundle, a service can also be viewed as having states of execution and transitions between those states. Fortunately, the Service lifecycle is considerably less complicated than the Bundle lifecycle. This is in large part because the resolution of package dependencies is done when the bundle is installed as opposed to when the service is registered. Recall from the introduction that in OSGi a service is just a Java object whose interface must be published with a well-known registry. This enables clients to find and bind to the service. Similarly, it must be removed from the registry, or unregistered, to prevent further acquisition of the service.

3.4.1 Service Lifecycle – Service States

The OSGi specification does not explicitly address the state of services. Probably due to the fact that, unlike Bundles, each service will have a different set of states and thus they do not lend towards standardization. This is because the logic of each service is defined by the service developer and is not known to the Framework.

3.4.2 Service Lifecycle – Service Events

However, there are three main types of service events that can happen in the framework that correspond to the registration and deregistration processes.

- ServiceEvent.REGISTERED – The service object has been registered with the framework and is ready to be used by clients.

- ServiceEvent.MODIFIED – When a service is registered, the registration may be accompanied by a Dictionary object describing the service. If a service's properties are changed at run-time, this event is triggered.

- ServiceEvent.UNREGISTERING – The service object is being removed from the registry and will not be able to be bound to or invoked.

3.4.3 Service Programming Tasks for Providers

The service provider is the entity responsible for developing an offering the service implementation, making it (un-)available for clients to use. There are basically two tasks that a provider must take care to perform: publishing the service and its description with the Framework registry and removing the service from the Framework registry. Optionally, in OSGi, service providers may alter the description of a service once it is registered with the Framework. In OSGi, a service description is a Java Dictionary object consisting of a set of key-value pairs[16] registering values for a set of service attributes. Most of the time, service attributes will be defined in an application-specific manner, and their meaning should be defined in the service's interface.

- **Registering a Service** – The *BundleContext.registerService(String, Object, Dictionary)* method is used to place a service object in the OSGi Framework registry. The operation has three arguments: the fully delimited path name of the interface that the service is to be registered as[17], the actual service object itself, and the possibly null set of service properties. This method returns a *ServiceRegistration* object that can be used to overwrite a service's properties or unregister the service (see below). After this method is called, clients may retrieve and use this service from the Framework. Also, this method triggers the Framework to fire an event of type ServiceEvent.REGISTERED as discussed above.

- **Modifying Service Properties –** In some cases, a service's properties may need to be changed. The *ServiceRegistration.setProperties(…)* replaces the current service properties with a new set of properties. This causes the Framework to throw an event of type *ServiceEvent.MODIFIED* as discussed above. All new references that clients obtain to this service will have the new list of properties.

- **Unregistering a Service** – Lastly, the service provides may need to remove the service implementation from the Framework (in order to do maintenance, for example). This is accomplished by calling the *ServiceRegistration.unregister ()* on

[16] A key-value pair is a pair <K,V> where K is an identifier, and V is the value assigned to identifier K. For Dictionary objects in Java, K is a String and V can be any object.
[17] This can be obtained by accessing the feature *class.getName()* on the implemented service interface.

the *ServiceRegistration* object associated with the service to be removed. The execution of this method removes the service from the Framework registry, causing all ServiceReferences to this service to be invalidated and unusable and firing a *ServiceEvent.UNREGISTERING* event.

3.4.4 Service Programming Tasks for Clients

Clients are the consumers in a service-oriented environment. They use the services that providers have made available in the Framework through three main tasks: querying the service registry, examining the service's properties, invoking the service, and releasing the service handle.

- **Querying the Service Registry** – The *BundleContext.getServiceReference(String)* method stakes one parameter, a fully delimited class name of the interface that the service has been registered as and returns a *ServiceReference* object that functions as a handle to the particular service implementation. If no such service has been registered, the result will be null. Alternatively, the *BundleContext.getServiceReference(String, Filter)* can be used to obtain an array of references for all services matching a particular filter expression.

- **Examine a Service's Properties** – The *ServiceReference* object returned by the *getServiceReference()* has several methods that can be used to examine the properties of a service registration. *ServiceReference.getPropertyKeys()* returns an array of the property identifiers associated with the service, and *ServiceReference.getProperty(String)* returns the value associated with a particular key.

- **Binding to a Service –** After obtaining the desired *ServiceReference*, the actual service implementation is obtained from the Framework by using the *BundleContext.getService(…)* and passing it the ServiceReference object previously obtained. The result of this method is instance of a Java Object class and must be cast to the given service interface to be used as that service. Note that if the service that the *ServiceReference* object was linked to was unregistered from the Framework after the reference was retrieved, it may be possible for the result to be null. Thus, it should be tested before it is used.

- **Invoking a Service –** After the service implementation is obtained, the client may invoke any operation complying with the service's interface. This is done exactly as any Java method invocation. Note that since SOA is expected to be dynamic, it may be possible that the service object is removed from the framework in between the client's method calls. If this is the case, the service object is set to null. Therefore, it is good practice to always test the validity of

the service object before invoking it or register a *ServiceListener* with the Framework and implement the appropriate event handler.

- **Releasing a Service –** When the client is finished using the service, it must release the handle (or binding) it previously obtained. This is accomplished by calling the *BundleContext.ungetService(ServiceReference)* with the ServiceReference of the service to be released. After a client has released all references to a service (a client may have multiple references to the same service), the service object has been deallocated from the client and the service object and its references cannot be used any more.

4. Tutorial Part 1: Developing A Simple Service

Now that the basics of OSGi, Bundles and Services have been discussed, this chapter is devoted to presenting a brief tutorial for developing a simple (i.e. non-composite) software service, including implementing, deploying, and managing a simple service and its respective OSGi bundle are presented here. For this part of the tutorial you will download a partially implemented bundle that will allow you to both see the major concepts of OSGi programming as well as practice them.

4.1 Introduction to the PopupDisplayer Service

The service that you will be developing in this section is one that enables simple graphical notifications to users. The service will be able to display a text message to the user as well as to display a text message accompanied with a picture. Since OSGi is Java-based, the actual service logic will be very simple, allowing you to focus on developing service and bundle development.

4.2 Checking out a Bundle Project from the Repository

1. From the Eclipse File Menu, select File -> New -> Project.
2. In the Project Dialog, select SVN -> Checkout Project from SVN and click Next.
3. Create the new repository location **svn://smarthome.cs.iastate.edu/smarthome/Documents/Tutorials (Code)** and click Next.
4. When prompted for authorization, enter "guest" as the username, leave the password blank, and select 'OK'.
5. Select the folder **PopupDisplayerService** and click Finish. You should see the project in your workspace, and there should be no build errors. If by chance OSGi classes are not recognized, check that the OSGi framework has been added to Eclipse preferences for this workspace.
    a. Select Window -> Preferences -> OSGi ->Frameworks
    b. If no framework is listed, create a new one.
        i. Enter "Knopflerfish-2.2.0" as the Framework Name.
        ii. Select "Knopflerfish 1.3/2.0" as the Framework Type.
        iii. Enter "C:\knopflerfish\knopflerfish.org\osgi" as the Framework home directory.
        iv. Use the default settings.
6. Please take notice the directory structure. The lab convention for organizing the contents of an OSGi project is as follows:
    - The toplevel directory contains the **bundle.manifest** and files.
    - The **/src directory** contains all interface and implementation files. You should be able to locate PopupDisplayService interface as well as the Activator and PopupDisplayServiceImpl files, all of which are located in their proper packages.
    - The **/bin directory** contains all class files compiled from the /src directory and the final packaged jar for the bundle[18]. Open this directory in the Eclipse Resource Perspective and you can see the class files and the packaged jar file.

[18] If the Ant build system is being used, then the /build folder should contain the final packaged jar

- The **/lib directory** contains any pre-compiled jar files that will be needed at compile-time or run-time by the bundle. If this bundle/service depended on other jar files that need not be made available in the framework, they would be placed here.
- The **/res directory** contains any other data files needed by the service, such as images, data files, or properties files. There should not be anything in this directory for the current bundle.

4.3 Browsing the Service Interface

You should now have the Bundle Project in your local Eclipse workspace. Peruse the PopupDisplayerService interface file *PopupDisplayService.java* located in the **/src** sub-directory in the *edu.iastate.cs.smarthome.tutorials.popupdisplayer* package.

```
package edu.iastate.cs.smarthome.tutorials.popupdisplayer;

…

public interface PopupDisplayService {

      public void display(String msg);

      public void display(String msg, Image img);
}
```

You should notice two methods, *displayText(String)* and *displayTextAndImage (String, Image)* that correspond to contracts for the two functionalities mentioned above. The first displays a simple messagebox with the text passed to it as an argument, and the latter displays a messagebox with text and an image it is passed via arguments.

4.4 Browsing the Service Implementation

Now, browse the implementation of this service *PopupDisplayServiceImpl* for in the edu.iastate.cs.smarthome.tutorials.popupdisplayer.impl package. Notice that this class is declared to implement the PopupDisplayService interface. The two method implementations are relatively straightforward calls to static method *JOptionPane.showMessageDialog(…)*. One version just displays text, and one version displays a text and an image. Note that the use of the JOptionPane class requires the *javax.swing* package to be imported. Since there is no jar providing these classes in the Bundle libraries, this package will need to be imported from the OSGi Framework.

4.5 Verifying the Manifest File

Double check that the values for the following headers in the manifest file are correct by double-clicking on the file **bundle.manifest**[19] in the project directory. The Eclipse Knopflerfish plug-in provides a nice interface for editing manifest file headers and values and embedding

[19] The manifest file is named **manifest.mf** if Ant is being used (See Appendix A).

resources in the target jar file. This editor provides three tabs: a **Bundle Manifest Tab** to inspect and edit header values, a **Bundle Packaging Tab** to indicate how resources are packed into the Bundle[20], and a **bundle.manifest Tab** that gives the "as is" view of the manifest file. Use this editor to confirm the following properties of the manifest file:

1. The **Bundle name** header is PopupDisplayService.

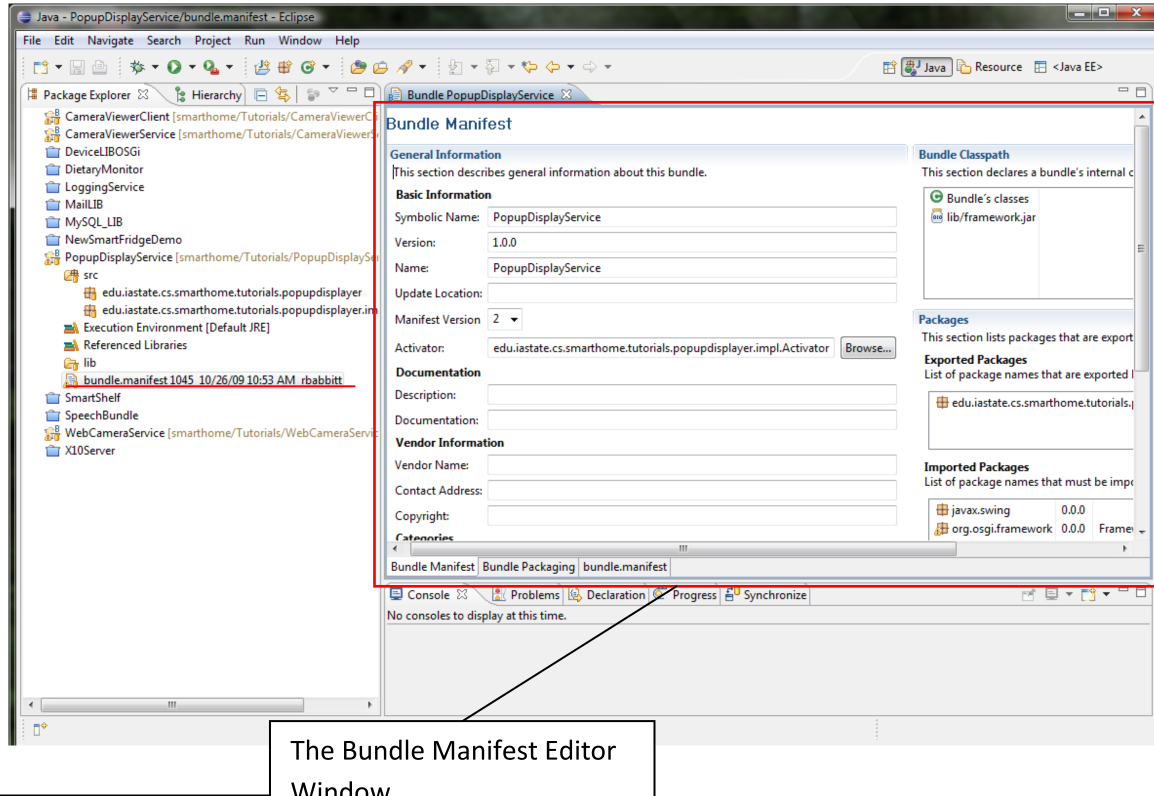
2. The **Symbolic Name** header is the package-delimited name of PopupDisplayService

3. The **Bundle Activator** should be in edu/iastate/cs/smarthome/tutorials/display/impl. Note that there are two activator classes, Activator and ActivatorSkeleton. The former is the full implementation of the latter, included for your reference.

4. The **Bundle Classpath** should include local class files.

5. The package edu.iastate.cs.smarthome.tutorials.display is exported. Why?

6. The package javax.swing should be imported. Why?



[20] The .bundle-pack file in the top-level directory is an XML file describing which resources should be packed into the bundle. On certain occasions, this file should be edited directly (and carefully) to update the Bundle's packing instructions.

4.6 Implementing the Activator Class

The interface and implementation files have been provided to you already finished. They do not need to be modified. However, the ActivatorSkeleton class has some code that needs to be added. Recall that the Activator class is what the OSGi framework uses interact with the Bundle, and that the *start()* method is where the logic to create, register, or obtain services and resources is placed, and the *stop()* method is where the logic to unregister and release services and resources is placed.

1. Add a local variable named *displayReg* of type ServiceRegistration. This is where the result from registering the PopupDisplayerService is stored.

2. Locate the Activator's start() method and fill in the code to register the service with the OSGi Framework. This is done by calling the BundleContext.registerService(String, String, Dictionary) method. The first parameter is the name of the type of service being registered, and the second argument is the service object being registered. The third parameter is a set of properties describing the service object, but should be left null in this case. The result of method call should be stored in the *displayReg* object so that the service can be removed from the registry at a later time.

3. Locate the Activator's stop() method and fill in the code in the Activator's stop method to unregister the service from the Framework by calling unregister () operation on *displayReg*.

4.7 Executing the Service

In order to execute the service, like all software, it must first be compiled from the source into an executable format. This process is also called *building* the executable. Normally, Eclipse automatically compiles source files as you edit them, and the Knopflerfish plug-in re-packages the bundle jar file in the same way. However, a bundle may be forced to be re-compiled by cleaning the workspace, using the Project -> Clean menu option (as long as the Project->Build Automatically option is selected).

It is possible that some OSGi bundles developed prior to the lab's adoption of the Knopflerfish plug-in may use Ant build script to build the project. See http://ant.apache.org/ for a tour of the Ant build system.

4.7.1 Deploying the New Bundle

After being compiled, the bundle must be loaded into the OSGi Framework. There are several ways that a bundle can be installed using the Knopflerfish Desktop as listed below. Choose one way to load the bundle in the framework. If there are no errors loading the bundle (i.e. missing packages/classes), then the bundle is started,

calling Activator's *start()* method, and the service should be activated. If there are errors, double check the above steps and try again.

1. **Option 1: Drag and Drop.** Bring up the Navigator View in Eclipse (Window -> Show View -> Navigator), and select the bundle jar file with the mouse and drag it into the Bundle Window. The Framework will automatically install the bundle.
2. **Option 2: Install from Filesystem.** From the File Menu, select "Open Bundle File" and navigate to the bundle jar file on the local file system.
3. **Option 3: Install from URL.** Alternatively, from the File Menu "Open Bundle Location" and enter the URL of the bundle (may be a local file using the file:// protocol).

4.7.2 Starting the Bundle

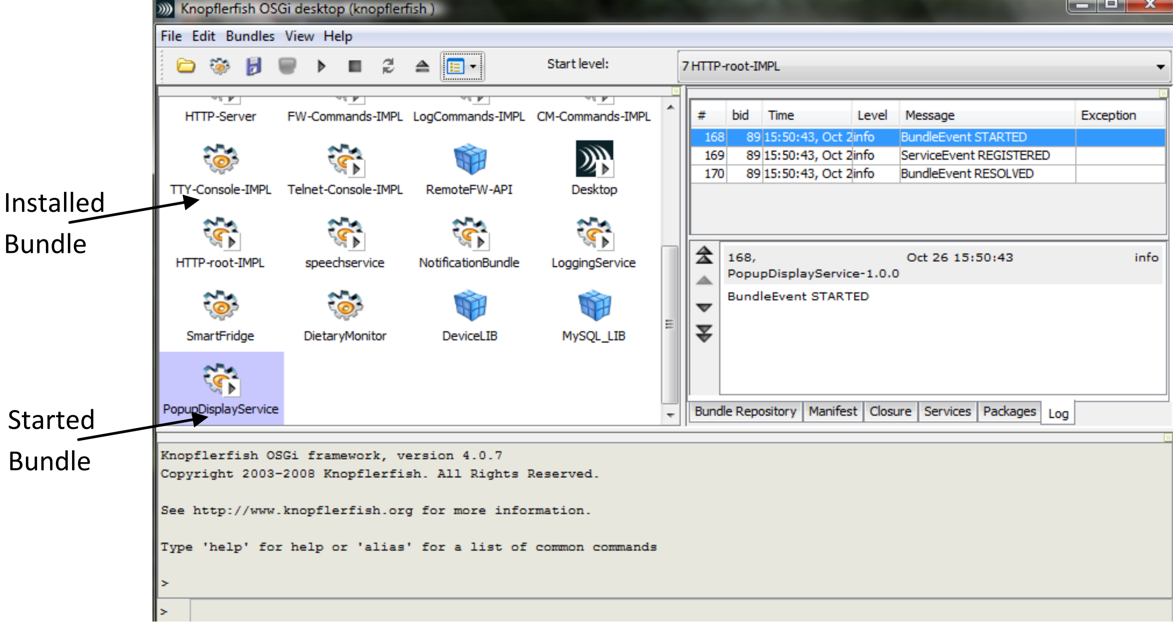
When the bundle is successfully installed, its package dependencies have been resolved, but is still needs to be started for its service dependencies to be resolved.

1. Start the bundle by selecting the "play" icon (resembling the play icon of CD/DVD players) OR
2. Start the bundle by selecting the PopupDisplayService bundle and selecting Bundles --> Start from the Menu bar.

After starting successfully, the bundle's service will be registered and its icon will be updated to indicate its active status. See the figure below.

4.7.3 Managing the Bundle

The PopupDisplayService Bundle should now be installed in the Framework, and you should see an active bundle icon with the name "PopupDisplayService" in the Bundle Window as shown below. Nothing else happens yet, because the service has not been invoked.



1. Use the Knopflerfish Desktop to inspect the manifest file, observe the various package and service dependencies, examine the service interface and properties, and view the Bundle and Service events associated with this Bundle by selecting the appropriate tabs in the Bundle Information Window.

2. Practice starting, stopping, refreshing, uninstalling, and re-installing the Bundle, observing the changes in the Bundle and Bundle Information Windows. What does each of these actions cause the framework to do? What effect does each of these actions have on the framework? Remember that you may need to de-select and re-select a bundle in order to view the changes in the Bundle Information Window. Use all the different methods mentioned above.

Congratulations! You've installed your first OSGi bundle and made your first OSGi service.

5. Tutorial Part 2: Creating a Composite Service

Now that you have developed a Bundle for a simple OSGi service and have a working understanding of the basics of Knopflerfish OSGi. This next tutorial will incorporate the reusability and composability aspects of SOA into your skill set. In this section, you will practice retrieving services from the Framework, create a new *composite service* from existing services. In addition to the PopupDisplayService developed in the previous chapter, this tutorial will make use of another simple service, the WebCameraService, that enables images to be retrieved from one of the network cameras in the smart home lab. Its interface, as defined below, consists of a set of constants that designate the resolutions that the camera supports and a method to retrieve an image at a given resolution.

```
package edu.iastate.cs.smarthome.tutorials.camera;

public interface WebCameraService{

        public static final String RESOLUTION_320x240 = "320x240";
        public static final String RESOLUTION_640x480 = "640x480";

        public Image getImage(String resolution) throws IOException;
        public void recenter()throws IOException;
        public void panLeft() throws IOException;
        public void panRight()throws IOException;
        public void tiltUp()throws IOException;
        public void tiltDown()throws IOException;
}
```

In fact, the WebCameraService Activator will register five different web camera services, each associated with an internet camera with a different IP address from 129.186.93.32 to 129.186.93.36. Our implementation registers each WebCameraService with a property “HOST” that is the IP address of the corresponding web camera. Another property “SOAP.service.name” is used to declare that this service object should also be registered as a web service. The code for doing this is shown below.

```
package edu.iastate.cs.smarthome.tutorials.camera.impl;

public class Activator{

        //Construct a properties object
        Properties p = new Properties();
        p.put("HOST", "129.186.93.32");
        p.put("SOAP.service.name", "WebCameraService");

        //Register camera services
        camReg1 = context.registeService(
              WebCameraService.class.getName(),
              new WebCameraServiceImpl("http://129.186.93.32:80"), p);

        //Repeat for other WebCameraServices
        …
}
```

5.1 Checking out the WebCameraService Bundle

1. From the Eclipse File Menu, select File -> New -> Project.
2. In the Project Dialog, select SVN -> Checkout Project from SVN and click Next.
3. Select the existing **svn://smarthome.cs.iastate.edu/smarthome/Documents/Tutorials (Code)** repository and click Next.
4. Select the **WebCameraService** folder and click Finish. You should see the project in your workspace, along with your previous project.

5.2 Inspecting and Installing the WebCameraService Bundle

1. Again, browse the directory structure.
2. Install the network camera bundle by dragging the WebCameraService.jar file into the Bundle Window. As before, the Knopflerfish Desktop should update to show the activated Bundle.
3. Inspect the various tabs of the Bundle Information Window to see the manifest header values and package dependencies of this Bundle. You should see five different WebCameraService objects registered.

In addition to the simple PopupDisplay and WebCamera services, in this tutorial you will develop a composite CameraViewerService that retrieves images from a WebCameraService and displays them using the PopupDisplayService. The interface of the CameraViewerService, given below, defines two operations. The first will initiate a loop which repeatedly displays images from the network camera with the corresponding host, and the second stops the viewer from displaying images. The implementation has been partially completed for you, so the next task is to finish the implementation.

```
package edu.iastate.cs.smarthome.tutorials.cameraviewer;

public interface CameraViewerService {

	public void displayImages(String ip, int delay, String
		resolution);

	public void stopViewer();
}
```

5.3 Checking out the CameraViewerService Bundle

1. From the Eclipse File Menu, select File -> New -> Project.
2. In the Project Dialog, select SVN -> Checkout Project from SVN and click Next.
3. Select the existing **svn://smarthome.cs.iastate.edu/smarthome/Documents/Tutorials (Code)** repository and click Next.
4. Select the **CameraViewerService** folder and click Finish. You should see the project in your workspace, along with your previous project.

5.4 Inspecting the CameraViewerService Interface and Implementation

Inspect the partial CameraViewerService implementation located in the edu.iastate.cs.smarthome.tutorials.cameraviewer.impl package. There are several important things to note about this implementation.

1. **Member Variables.** First, there are two sets of member variables: one for the component services and their properties and the other for the image retrieval and display logic. Each component service that is used in the implementation needs to have a variable. Thus, there is one member variable for the single PopupDisplayService that will be used and a Vector<WebCameraService> member to store the multiple services. Additionally, there is a Vector<String> member to store IP addresses of each of the cameras, obtained from the Properties object registered with the service. These variables are initialized in the constructor.

```
//Component service variables
private PopupDisplayService displayer = null;
private Vector<WebCameraService> cameras = null;
private Vector<String> cameraIPs = null;
```

2. **Methods to add Component Services.** Second, there are two package-visible methods use the component service objects obtained at run-time to populate the appropriate member variables. The implementation should be straight forward. It will be the Activator's responsibility to obtain the necessary ServiceReferences and their associated service objects and call these methods to configure CamerViewerService appropriately.

```
//Configure this service for a given PopupDisplayer
void setDisplayer(PopupDisplayService p){
      this.displayer = p;
}

//Configure this service to include a given WebCamera
//with the given host name
void addCamera(WebCameraService cam, String host){
      this.cameras.add(cam);
      this.cameraIPs.add(host);
}
```

3. **Method Implementations.** Thirdly, the implementations of the interface methods are fairly straight forward. The *displayImages(…)* records the input parameters, selects the appropriate WebCameraService based on their IP addresses, invokes the displayText(…) operation on the PopupDisplayerService to inform the user that the

service is starting to execute, sets an execution indicator variable (proceed) and starts an image viewing thread.[21]

```
//Display images from the camera
public void displayImages(String ip, int delay, String res){

	//Set parameters for image capture
	this.milliseconds = delay;
	this.resolution = res;

	//Select the appropriate WebCameraService based on the ip
	this.theIP = ip;
	this.theCam =
		this.cameras.get(this.cameraIPs.indexOf(ip));

	//Execute viewer is separate thread
	if (displayer != null){
		displayer.display("Execution is starting");
		this.proceed = true;
		(new Thread(this)).start();
	}
}
```

The image viewing thread executes in a loop as long as *proceed* is set, sleeps for a period of time in milliseconds, invokes the *WebCameraService.getImage(…)* to retrieve an image at the desired resolution, and then displays the image by invoking *PopupDisplayerService.displayTextAndImage()* with the retrieved image.

```
while(proceed){

	…

	//Sleep the allotted time
	Thread.sleep(milliseconds);

	…

	//Retrieve the image and display it
	if(proceed){

		//Make sure camera service is non-null
		if (theCam != null)
			img = theCam.getImage(resolution);
		else{
			proceed = false;
			System.err.println("Null Camera service");
		}

		//Make sure display service is non-null
```

[21] It is not necessary to understand what a thread, how to use it, or how to use thread indicator variables since this part of the implementation is given. However, a nice tutorial about multi-threading in Java can be found at Oracle's site http://download.oracle.com/docs/cd/E17409_01/javase/tutorial/essential/concurrency/

```java
				if (displayer != null)
					displayer.display("Image " + i + " on host " +
						theIP, img);
				}
				else{
					proceed = false;
					System.err.println("Null Display service");
				}
		}
}
```

The stopViewer() method sets the execution indicator to false so the thread does not capture any further images and notifies the user by invoking the *displayText()* method of the PopupDisplayService.

```java
public void stopViewer(){

  //Set monitored variable to false
  this.proceed = false;

  //Notify user
  if (displayer != null)
        displayer.display("Execution is stopping");
}
```

4. **Error Checking.** Lastly, you should notice that before service objects are used, they are compared with the NULL object. This is good programming practice in OSGi because services are expected to dynamically leave and join the Framework. Once a service leaves the Framework, any lingering references are set to null, hence this test. However, the same is true for service references. If a reference (or set of references) is requested for a service that is not registered with the framework, the result is also a NULL object, and any subsequent call to that object will create a run-time exception.

5.5 Implementing the CameraViewerService Activator

The primary task of this part of the tutorial is to implement the Activator method. The appropriate member variables have been declared for you.

```java
//Service References
private ServiceReference displayerRef = null;
private ServiceReference[] cameraRefs = null;

//Service objects
private PopupDisplayService displayer = null;
private WebCameraService[] cameras = null;
private CameraViewerServiceImpl viewer = null;

//Service registrations
private ServiceRegistration viewerReg = null;
```

There is a member for each type of ServiceReference and Service that will be obtained from the framework. An array is used to contain the web camera references and services because multiple instances will be obtained from the framework. These component services will be obtained from the framework and used to configure the CameraViewerServiceImpl. Note that the CameraViewerServiceImpl class is being used instead of the CameraViewerService class so the package-visible configuration methods can be called. Lastly, there is a ServiceRegistration member is provided for the resulting composite service.

5.5.1 Implementing the stop() Method

Fill in the Activator's start() method to do the following:

1. Obtain service reference and service object for a PopupDisplayerService.

2. Obtain the service references and service objects for all of the WebCameraService objects registered to the framework. You will need to query the ServiceReference to obtain the value of the "HOST" property. The code for the querying and binding to the PopupDisplayBundle has been provided to start you off.

3. Create a new CameraViewerService object using the provided constructor.

4. Configure the new service to include all of its component services

5. Register the configured service with the Framework.

5.5.2 Implementing the stop() Method

Fill in the Activator's stop() method to do the following:

1. Unregister the CameraViewerService.
2. Release all of the component services.

5.5.3 Add error checking for NULL service references

Be sure to add error checks for NULL service references.

1. Print error conditions to the Framework console.
2. Test your error handling by stopping the required component services and trying to start the composite service. Does it report the errors correctly without starting/stopping?

5.6 Inspecting the Manifest

Double check the manifest file to be sure that the activator is found, the bundle class path is correct, and the correct packages are imported and exported.

5.7 Deploying the Bundle

When finished with the implementation, install your Bundle in the OSGi Framework, fixing any errors that occur. Once successful, try repeatedly stopping and starting your bundle, paying special attention to the package and service dependencies in the Bundle Information Window Tabs. You should see the two packages imported, one package exported, and six service dependencies.

Congratulations! You've created your first composite service.

6. Tutorial Part 3: Creating an Application Bundle

In this part of the tutorial, you will be responsible for creating an *Application Bundle*, which does not provide any classes or services to the OSGi Framework but only consumes them. This application will be called CameraViewerClient and basically binds to the CameraViewerService and uses it to display images based on input parameters obtained from a local input file.

6.1 Creating the Main Project Folder

1. Select File -> New -> Project -> OSGi -> Bundle Project.
2. Enter CameraViewerClient as the name of the project,
3. Make sure to name the output folder to "bin" in the Project Setting Pane.
4. Click Next
5. On the next screen, select Create Bundle Activator
6. Use edu.iastate.cs.smarthome.tutorials.viewerclient as the package name, and create the Activator.
7. Click Finish.

6.2 Creating the Remaining Directory Structure

1. The **/src** and **/bin** directories will be automatically created
2. Create the remaining directories by selecting the appropriate project in Project Explorer Window of Eclipse and selecting File -> New -> Folder and entering the name of the directory.
3. In the **/res** directory, create an input file named "input.txt". This file will provide the parameters to configure the CameraViewerService.
4. Add input parameters to input.txt. On the first line of the file, type an IP address from 129.186.93.32 to 129.186.93.36. On the second line of the file, type a number of milliseconds greater than 2000. On the third line type a resolution (either 320x240 or 640x480).

6.3 Editing the Bundle Packaging

You will not use the Bundle Packaging Table of the Manifest Editor, add /res and /res/input.txt as resources.

1. Double click on the bundle.manifest file and select the Bundle Packaging Tab

2. Click on the “Add” button
3. Enter “/CameraViewerClient/res” as the path to the resource in the project and “res” as the path in the bundle.
4. Select OK.
5. Repeat steps 2-5 to add “/CameraViewerClient/res/input.txt” as res/input.txt to the bundle.

6.4 Implementing the Activator Class

6.4.1 Declaring Member Variables.

This Application Bundle only will only have one service reference to a CameraViewerService. Declare the necessary ServiceReference and CameraViewerService member variables.

6.4.2 Implementing the start() method.

1. Read configuration parameters from the input file using the following code:

```
//Get a reference (stream) to the config file
BufferedReader in = new BufferedReader(new InputStreamReader(
      getClass().getResourceAsStream("/res/input.txt")));

//Get CameraViewer parameters from config file
String host = in.readLine();
int delay = Integer.parseInt(in.readLine());
String resolution = in.readLine();
```

2. Query the OSGi registry for a CameraViewerService reference and bind to it.

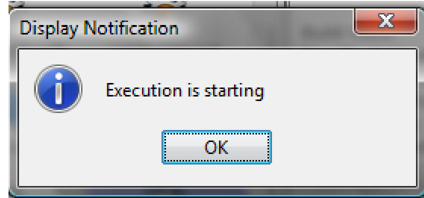
3. Invoke the *displayImages(…)* method passing in the parameters read from the configuration file.

6.4.3 Implementing the stop() method.

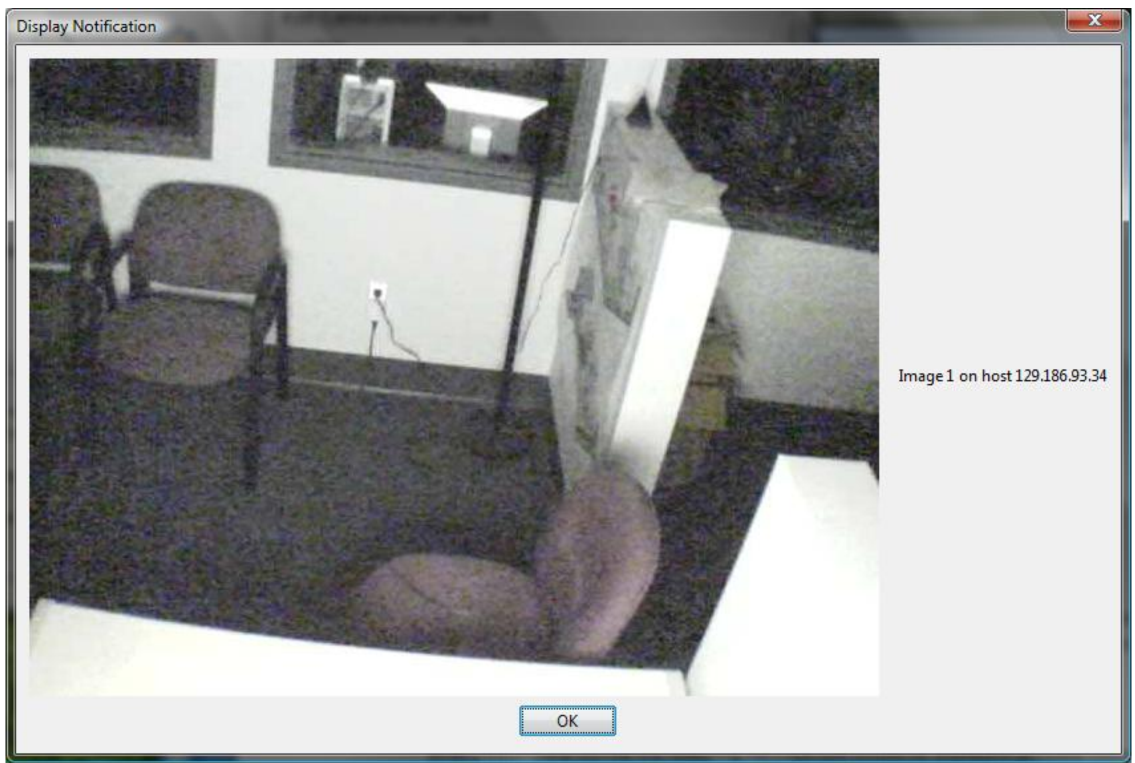
1. Release the CameraViewerService.

6.5 Deploying the Application Bundle

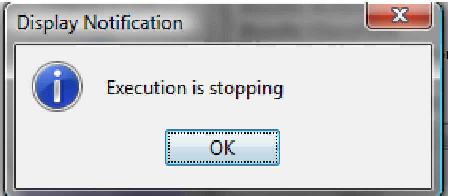
Upload the new Application Bundle to the Framework and play with it. Upon a successful activation, you should see the message box.



After clicking OK, you should see another series of message boxes displaying web camera images at the desired resolution. Depending on the web camera you have selected, the images should look like the following:



After stopping the bundle the following message box should be displayed (possible followed by another image due to concurrency issues.)



If you edit the configuration file to change cameras, delay, or resolution, remember to refresh the Bundle in the Framework before starting it. Otherwise, the input file will not change in the installed bundle.

Congratulations! You've created your first OSGi application.

7. Glossary of Terms

Activator – see Bundle Activator

Activities of Daily Living (ADL) – Basic activities, such as eating, bathing, dressing, etc. that an individual must be able to do on their own to be considered able to live alone

Actuator – A device that a *smart home* uses to control objects in the real world, such as a motor, switch, light, etc..

Bundle – A collection of files, including the *service interface*, service implementation, *activator*, and *manifest file* that can be deployed to an executing *OSGi framework*.

Bundle Activator – The class that the *OSGi framework* interacts with to start and stop a *bundle*. The path to a bundle's *activator* is given in the *manifest file* with value of the BUNDLE_ACTIVATOR property.

Deployment Environment – The collection of resources, *services*, and applications expected and required for deploying and running a service or application to the ISU lab smart home

Development Environment - The collection of resources, *services*, and applications expected and required for developing a service or application to the ISU lab smart home

Eclipse – The principle *IDE* used in the ISU smart home lab for software development

Hyper-Text Transfer Protocol (HTTP) – The well-known *protocol* for retrieving and posting documents and webpages on the Internet

Integrated Development Environment (IDE) – A software program that assists users in the process of managing, implementing. and debugging code.

Jar – Short for "Java archive", a compressed collection of files, like "zip" a file, that is used by the *Java* language and platform.

Java – The principal programming language on which *OSGi* and the development in the ISU smart home lab are based

Java Virtual Machine (JVM) – The program that interprets and executes Java applications. The current JVM version used in the lab is 1.5

Knopflerfish – the particular family of implementations of the *OSGi* specification that is being used in the ISU smart home lab. The current version in use is 2.2.0.

Open Service Gateway Initiative (OSGi) – the centralized, Java-based *service-oriented architecture* used in the Smart Home Lab

OSGi Framework – the term used in this document for the group of classes that manage and execute bundles. Both graphical and programmatic user interfaces for interacting with the framework are provided by *OSGi*.

Manifest File – the file, included in an *OSGi* *bundle*, that describes the properties and contents of that bundle, including the path of the *Activator* class and the dependencies of the bundle.

Platform – a collection of resources and applications upon which programs are executed

Protocol – a sequence of well-defined steps or messages that two or more parties use to communicate

Sensor – A device used by a smart home to collect data about its inhabitants or the physical world

Service – A reusable software artifact with a well-defined interface that provides known functionality to its users

Service Implementation – the collection of files containing the mechanism of the *service*

Service Interface – The file describing the fields and methods provided by a specific *service*

Service-Oriented Architecture (SOA) – A software platform for managing and executing services

Service Registry – The entity in *service-oriented architectures* responsible for maintaining the list of services currently available in the given platform

Smart Home – The term used for a home augmented with hardware devices, e.g. *sensors* and *actuators*, to provide the capabilities of monitoring and controlling of an inhabitant's physical environment, and software services/applications, to use the hardware devices for the inhabitant's benefit, use, and quality of life

Simple Object Access Protocol (SOAP) – the basic protocol used by web services to transmit service invocation requests and responses. It is *XML*-based over *HTTP*.

Simple Service Composition Language (SSCL) – The *XML*-based language used to specify the workflow of a composite service in the ISU smart home lab

Subclipse – The plug-in to the Eclipse *IDE* for version control of projects

Subversion (SVN) – The version control software used by the ISU smart home lab to keep track of the changes made to files over time

TortoiseSVN – A Windows-based graphical interface to *SVN*

Repository – The place where version controlled files and projects are stored

Universal Discovery, Description, and Integration (UDDI) – The standard XML-based *service registry* used to register and house web services

Virtual Machine – A software program that provides software interfaces of the hardware devices contained in the host system upon which it runs. These software interfaces are used by guest operating systems as though they were interacting with the real hardware

VMWare – The *virtual machine* platform used in the Smart Home lab

Web Services – Generally speaking a collection of *XML*-based standards relating to a decentralized *service-oriented architecture* that runs over standards *HTTP* protocol. More specifically, the actual services created and used in this platform. One of the two SOA platforms used in the ISU smart home lab

Web service container – The software entity responsible for "executing" a *web service*, i.e. receiving invocation request messages, executing the service, and returning the response message.

Web Services Definition Language (WSDL) – the *XML*-based language used to define the service interface of a *web service*

Extensible Markup Language (XML) - a platform-independent, self-describing language that is used as the basis for the *Web Services*