

5.1 Introduction

IoT systems comprise of multiple components and deployment tiers. In Chapter-1, we defined six IoT system levels. Each level is suited for different applications and has different component and deployment configurations. Designing IoT systems can be a complex and challenging task as these systems involve interactions between various components such as IoT devices and network resources, web services, analytics components, application and database servers. Due to a wide range of choices available for each of these components, IoT system designers may find it difficult to evaluate the available alternatives. IoT system designers often tend to design IoT systems keeping specific products/services in mind. Therefore, these designs are tied to specific product/service choices made. This leads to product, service or vendor lock-in, which while satisfactory to the dominant vendor, is unacceptable to the customer. For such systems, updating the system design to add new features or replacing a particular product/service choice for a component becomes very complex, and in many cases may require complete re-design of the system.

In this Chapter, we propose a generic design methodology for IoT system design which is independent of specific product, service or programming language. IoT systems designed with the proposed methodology have reduced design, testing and maintenance time, better interoperability and reduced complexity. With the proposed methodology, IoT system designers can compare various alternatives for the IoT system components. The methodology described in this Chapter is generally based on the IoT-A reference model [75], but is broad enough to embrace other industry efforts as well. Later chapters in this book describe the implementation aspects of various steps in the proposed methodology.

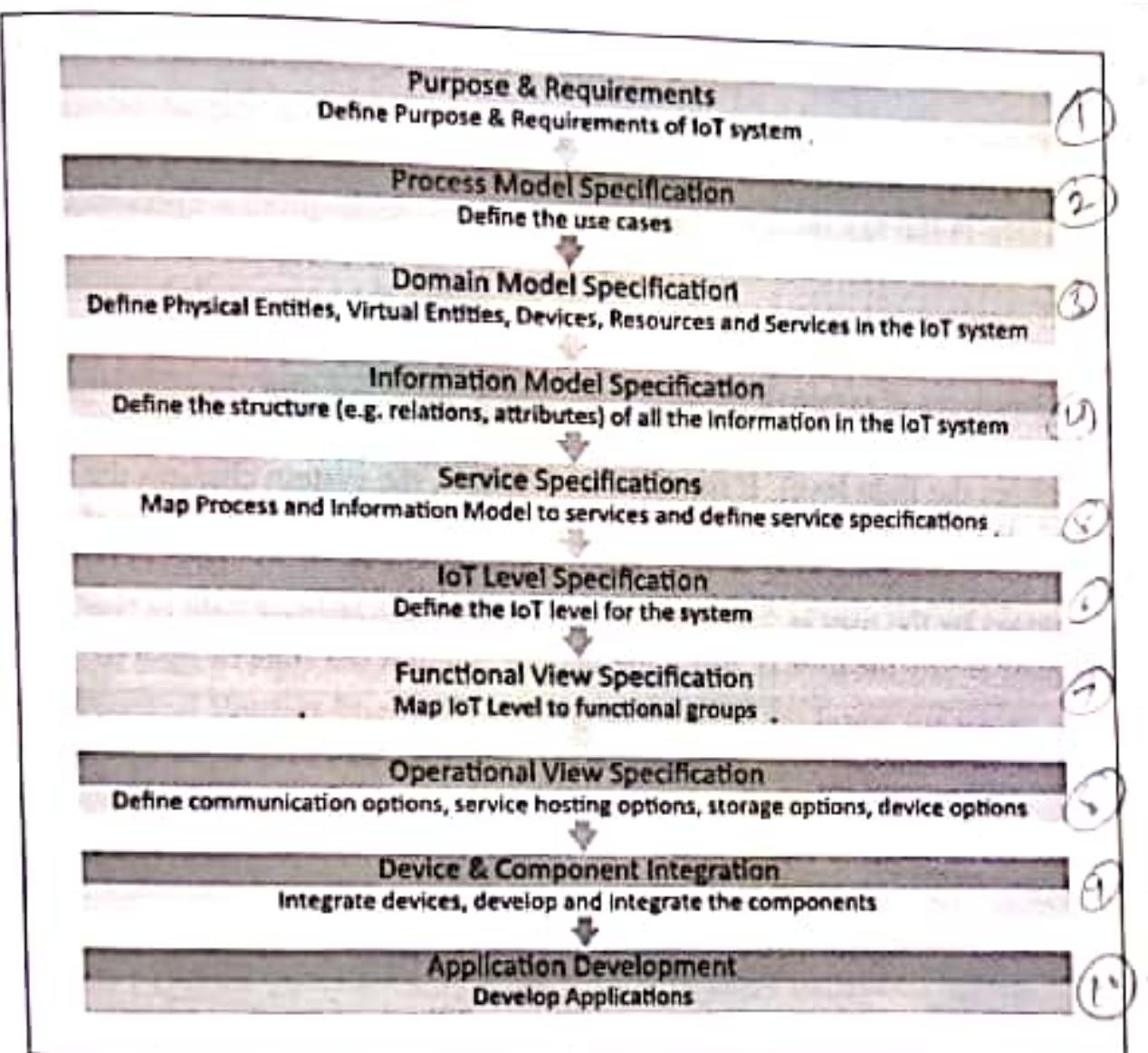
5.2 IoT Design Methodology

Figure 5.1 shows the steps involved in the IoT system design methodology. Each of these steps is explained in the sections that follow. To explain these steps, we use the example of a smart IoT-based home automation system.

5.2.1 Step 1: Purpose & Requirements Specification

The first step in IoT system design methodology is to define the purpose and requirements of the system. In this step, the system purpose, behavior and requirements (such as data collection requirements, data analysis requirements, system management requirements, data privacy and security requirements, user interface requirements, ...) are captured.

Applying this to our example of a smart home automation system, the purpose and requirements for the system may be described as follows:



✓ Figure 5.1: Steps involved in IoT system design methodology

- **Purpose :** A home automation system that allows controlling of the lights in a home remotely using a web application.
- **Behavior :** The home automation system should have auto and manual modes. In auto mode, the system measures the light level in the room and switches on the light when it gets dark. In manual mode, the system provides the option of manually and remotely switching on/off the light.
- **System Management Requirement :** The system should provide remote monitoring and control functions.
- **Data Analysis Requirement :** The system should perform local analysis of the data.
- **Application Deployment Requirement :** The application should be deployed locally on the device, but should be accessible remotely.

- **Security Requirement :** The system should have basic user authentication capability.

5.2.2 Step 2: Process Specification

The second step in the IoT design methodology is to define the process specification. In this step, the use cases of the IoT system are formally described based on and derived from the purpose and requirement specifications. Figure 5.2 shows the process diagram for the home automation system. The process diagram shows the two modes of the system - auto and manual. In a process diagram, the circle denotes the start of a process, diamond denotes a decision box and rectangle denotes a state or attribute. When the auto mode is chosen, the system monitors the light level. If the light level is low, the system changes the state of the light to "on". Whereas, if the light level is high, the system changes the state of the light to "off". When the manual mode is chosen, the system checks the light state set by the user. If the light state set by the user is "on", the system changes the state of light to "on". Whereas, if the light state set by the user is "off", the system changes the state of light to "off".

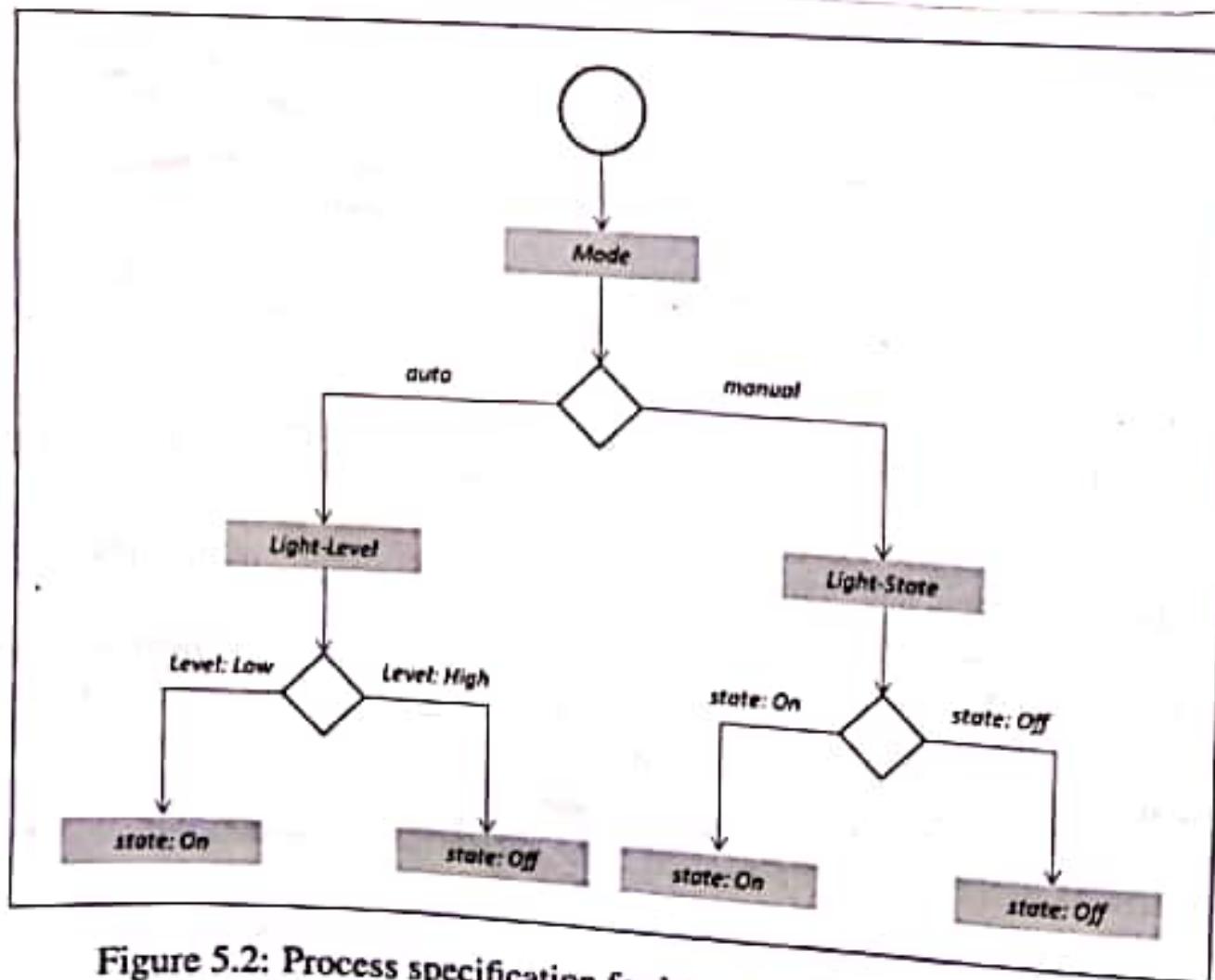


Figure 5.2: Process specification for home automation IoT system

5.2.3 Step 3: Domain Model Specification

The third step in the IoT design methodology is to define the Domain Model. The domain model describes the main concepts, entities and objects in the domain of IoT system to be designed. Domain model defines the attributes of the objects and relationships between objects. Domain model provides an abstract representation of the concepts, objects and entities in the IoT domain, independent of any specific technology or platform. With the domain model, the IoT system designers can get an understanding of the IoT domain for which the system is to be designed. Figure 5.3 shows the domain model for the home automation system example. The entities, objects and concepts defined in the domain model include:

- **Physical Entity :** Physical Entity is a discrete and identifiable entity in the physical environment (e.g. a room, a light, an appliance, a car, etc.). The IoT system provides information about the Physical Entity (using sensors) or performs actuation upon the Physical Entity (e.g., switching on a light). In the home automation example, there are two Physical Entities involved - one is the room in the home (of which the lighting conditions are to be monitored) and the other is the light appliance to be controlled.
- **Virtual Entity :** Virtual Entity is a representation of the Physical Entity in the digital world. For each Physical Entity, there is a Virtual Entity in the domain model. In the home automation example, there is one Virtual Entity for the room to be monitored, another for the appliance to be controlled.
- **Device :** Device provides a medium for interactions between Physical Entities and Virtual Entities. Devices are either attached to Physical Entities or placed near Physical Entities. Devices are used to gather information about Physical Entities (e.g., from sensors), perform actuation upon Physical Entities (e.g. using actuators) or used to identify Physical Entities (e.g., using tags). In the home automation example, the device is a single-board mini computer which has light sensor and actuator (relay switch) attached to it.
- **Resource :** Resources are software components which can be either "on-device" or "network-resources". On-device resources are hosted on the device and include software components that either provide information on or enable actuation upon the Physical Entity to which the device is attached. Network resources include the software components that are available in network (such as a database). In the home automation example, the on-device resource is the operating system that runs on the single-board mini computer.
- **Service :** Services provide an interface for interacting with the Physical Entity. Services access the resources hosted on the device or the network resources to obtain information about the Physical Entity or perform actuation upon the Physical Entity.

In the home automation example, there are three services: (1) a service that sets mode to auto or manual, or retrieves the current mode; (2) a service that sets the light appliance state to on/off, or retrieves the current light state; and (3) a controller service that runs as a native service on the device. When in auto mode, the controller service monitors the light level and switches the light on/off and updates the status in the status database. When in manual mode, the controller service retrieves the current state from the database and switches the light on/off. The process of deriving the services from the process specification and information model is described in the later sections.

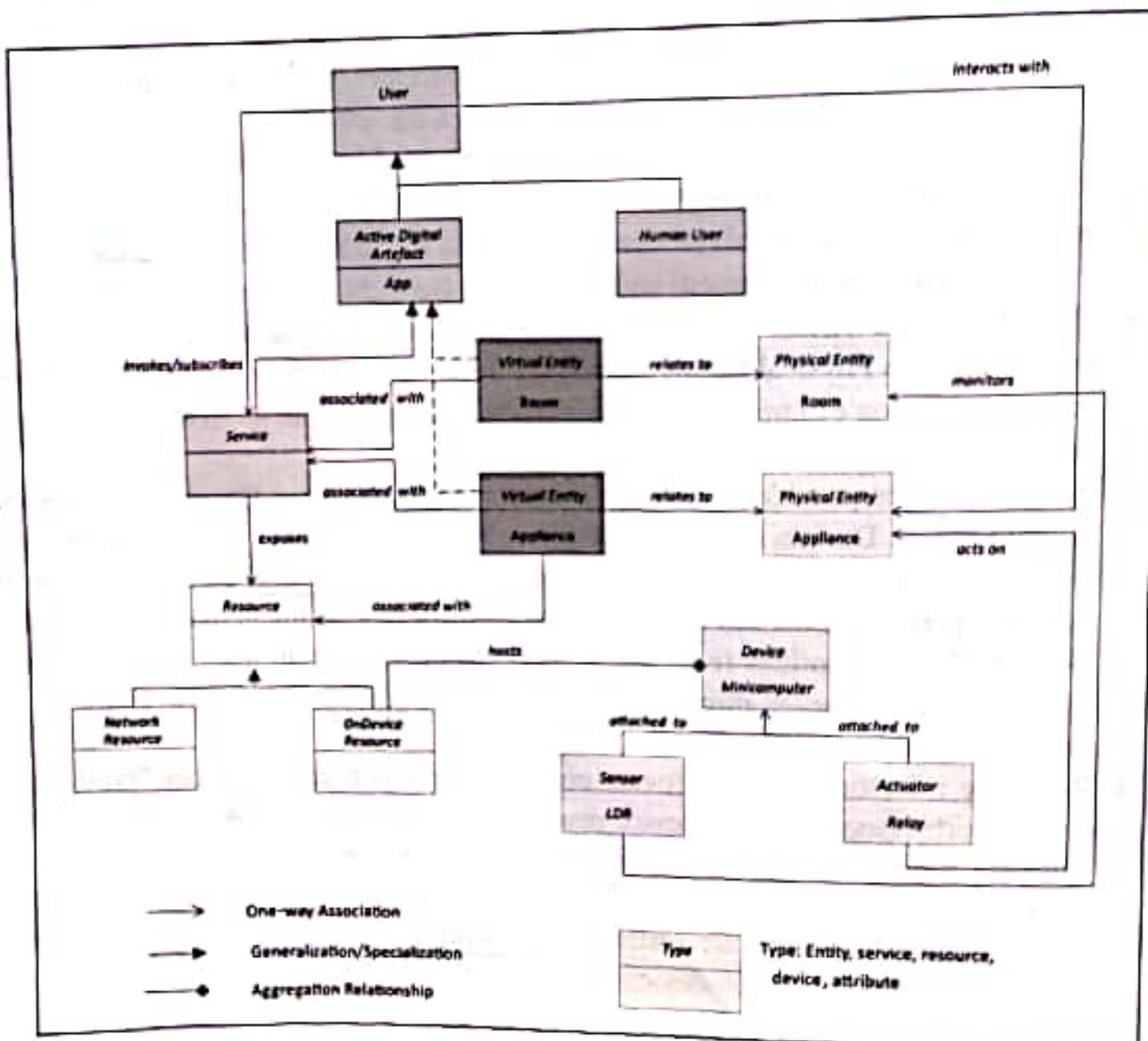


Figure 5.3: Domain model of the home automation IoT system

5.2.4 Step 4: Information Model Specification

The fourth step in the IoT design methodology is to define the Information Model. Information Model defines the structure of all the information in the IoT system, for example, attributes of Virtual Entities, relations, etc. Information model does not describe the specifics of how the information is represented or stored. To define the information model, we first list the Virtual Entities defined in the Domain Model. Information model adds more details to the Virtual Entities by defining their attributes and relations. In the home automation example, there are two Virtual Entities - a Virtual Entity for the light appliance (with attribute - light state) and a Virtual Entity for the room (with attribute - light level). Figure 5.4 shows the Information Model for the home automation system example.

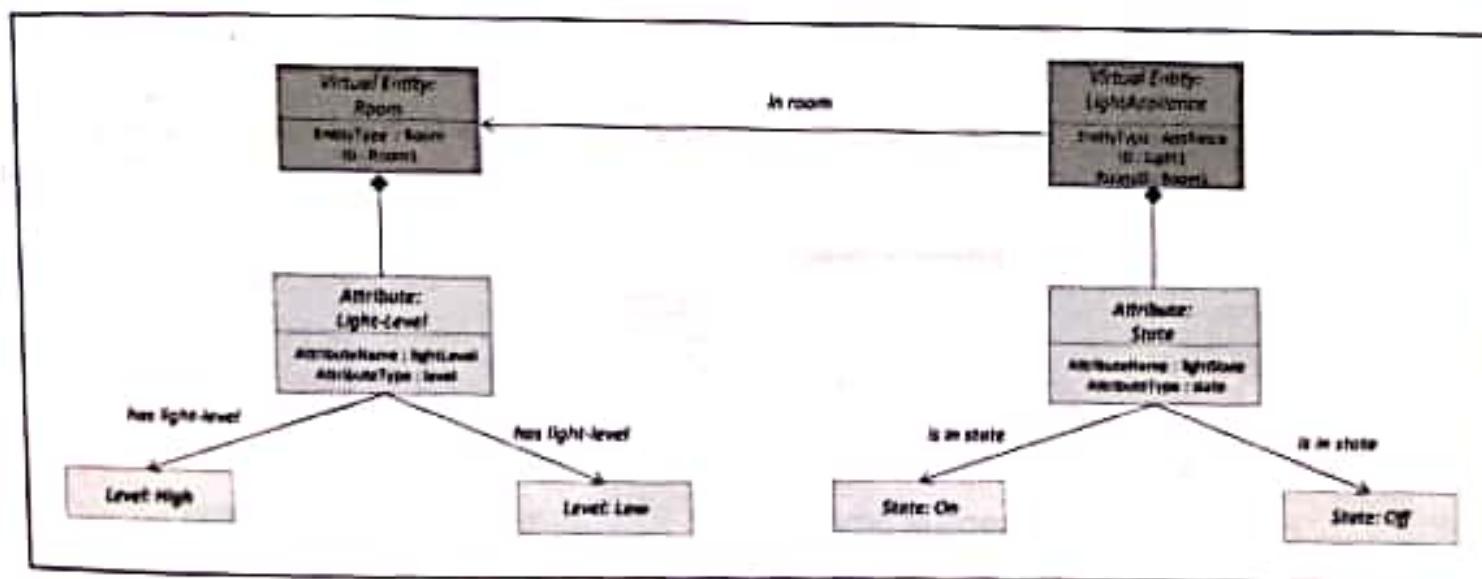


Figure 5.4: Information model of the home automation IoT system

5.2.5 Step 5: Service Specifications

The fifth step in the IoT design methodology is to define the service specifications. Service specifications define the services in the IoT system, service types, service inputs/output, service endpoints, service schedules, service preconditions and service effects.

You learned about the Process Specification and Information Model in the previous sections. Figure 5.5 shows an example of deriving the services from the process specification and information model for the home automation IoT system. From the process specification and information model, we identify the states and attributes. For each state and attribute we define a service. These services either change the state or attribute values or retrieve the current values. For example, the Mode service sets mode to auto or manual or retrieves the current mode. The State service sets the light appliance state to on/off or retrieves the current light state. The Controller service monitors the light level in auto mode and switches the

light on/off and updates the status in the status database. In manual mode, the controller service, retrieves the current state from the database and switches the light on/off.

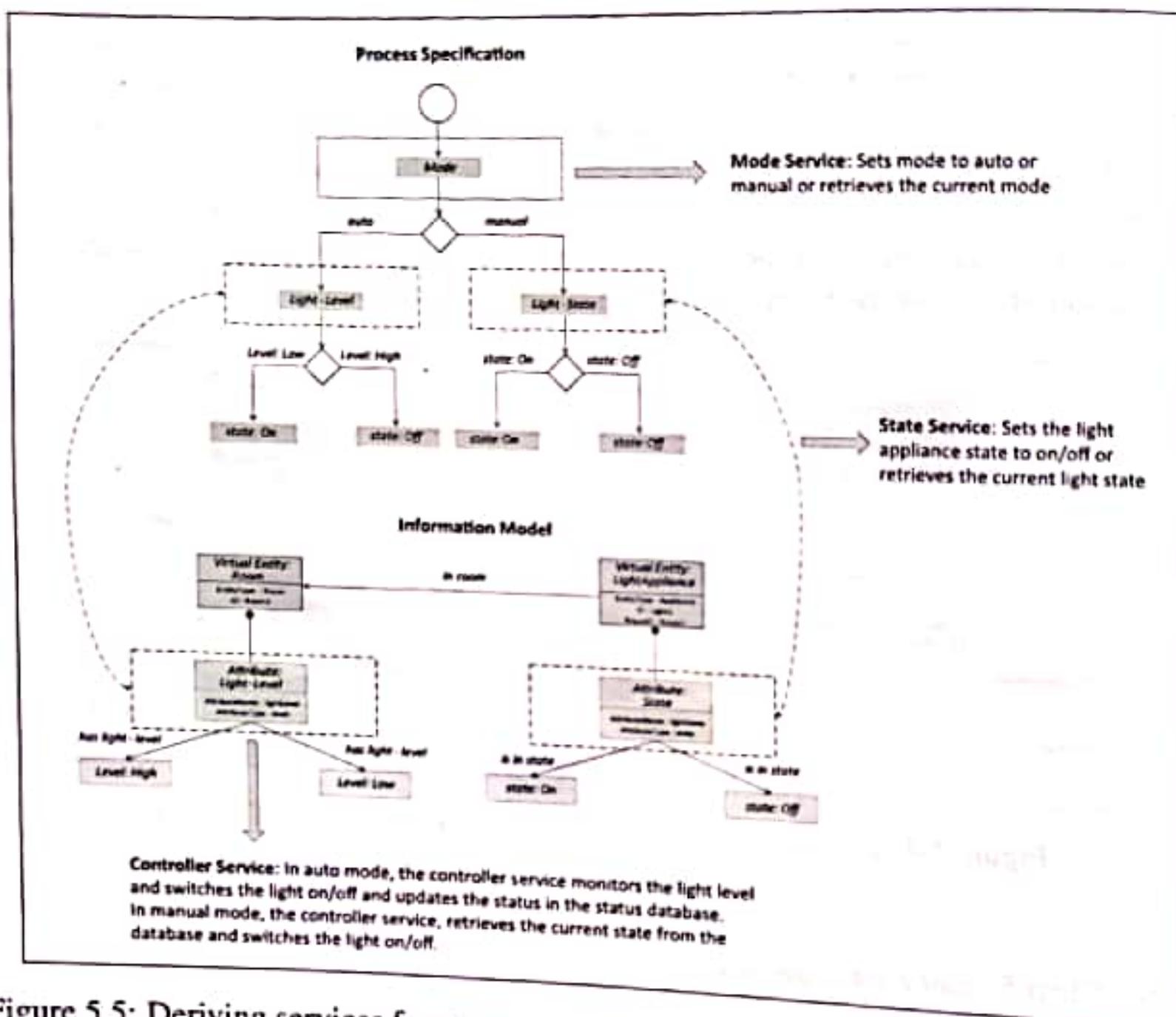


Figure 5.5: Deriving services from process specification and information model for home automation IoT system

Figures 5.6, 5.7 and 5.8 show specifications of the controller, mode and state services of the home automation system. The Mode service is a RESTful web service that sets mode to updated to/retrieved from the database. The State service is a RESTful web service that sets request). The state is updated to/retrieved from the status database. The Controller service runs as a native service on the device. When in auto mode, the controller service monitors

the light level and switches the light on/off and updates the status in the status database. When in manual mode, the controller service, retrieves the current state from the database and switches the light on/off.

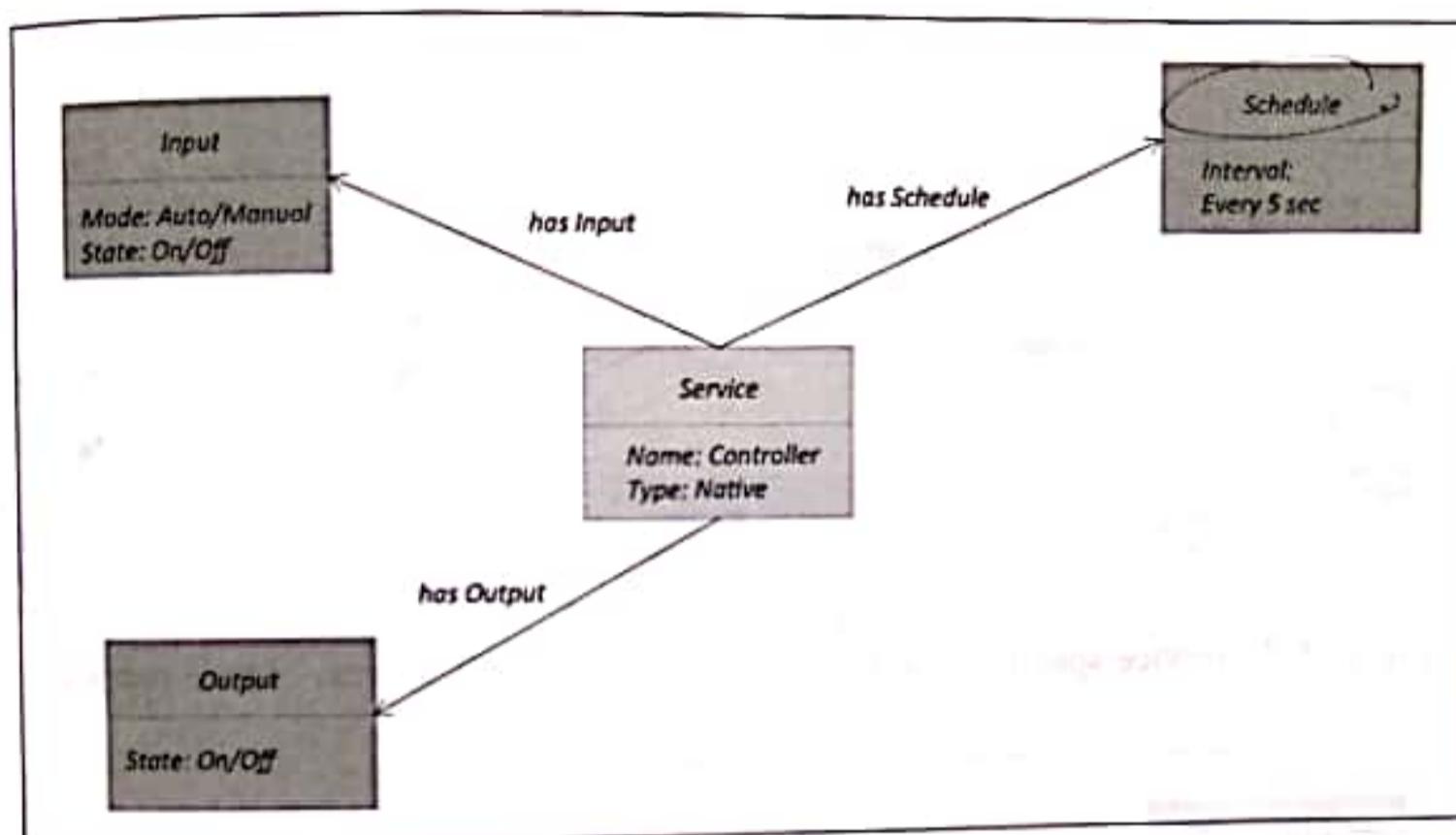


Figure 5.6: Controller service of the home automation IoT system

5.2.6 Step 6: IoT Level Specification

The sixth step in the IoT design methodology is to define the IoT level for the system. In Chapter-1, we defined five IoT deployment levels. Figure 5.9 shows the deployment level of the home automation IoT system, which is level-1.

5.2.7 Step 7: Functional View Specification

The seventh step in the IoT design methodology is to define the Functional View. The Functional View (FV) defines the functions of the IoT systems grouped into various Functional Groups (FGs). Each Functional Group either provides functionalities for interacting with instances of concepts defined in the Domain Model or provides information related to these concepts.

The Functional Groups (FG) included in a Functional View include:

- **Device** : The device FG contains devices for monitoring and control. In the home automation example, the device FG includes a single board mini-computer, a light

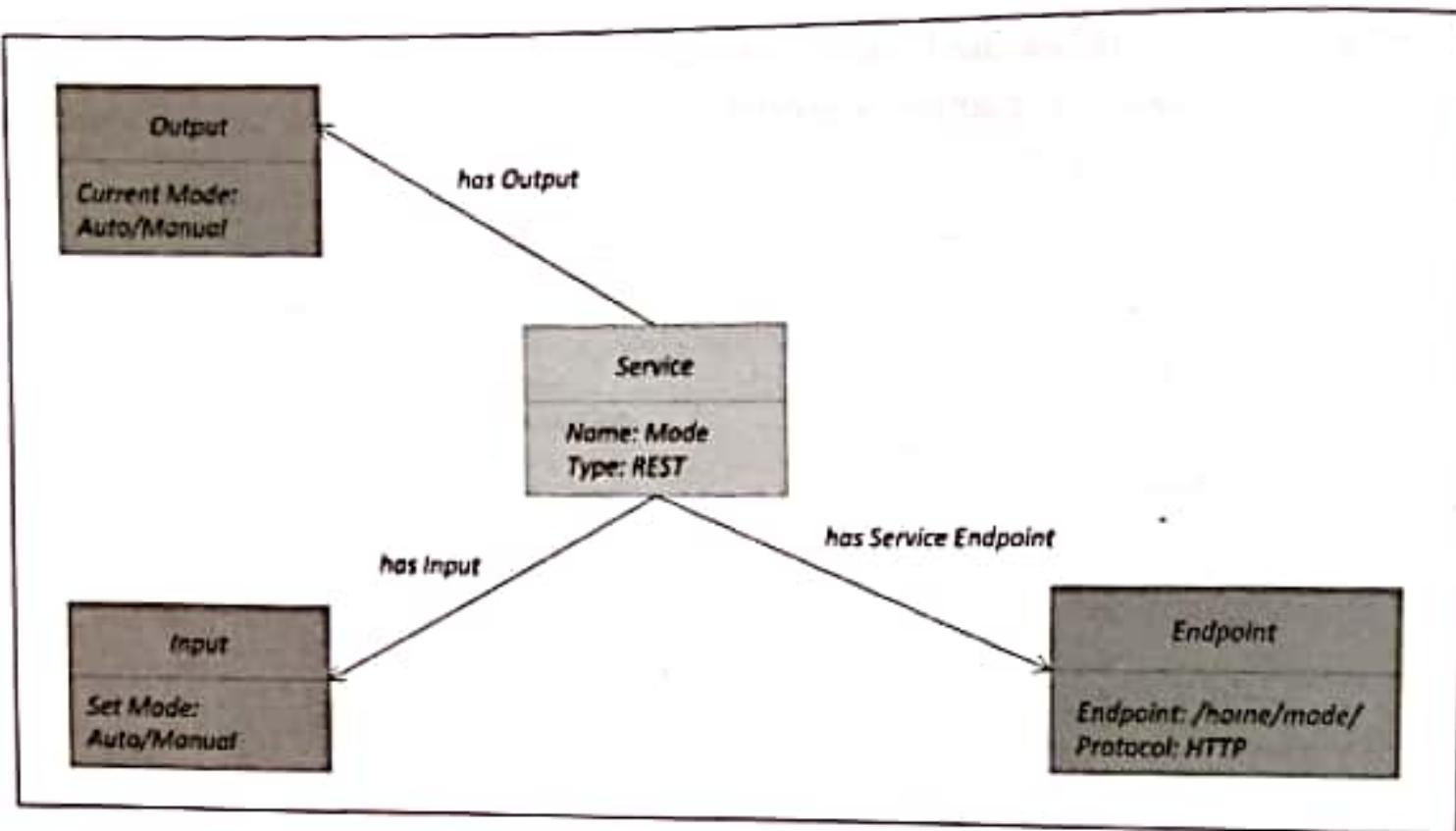


Figure 5.7: Service specification for home automation IoT system - mode service

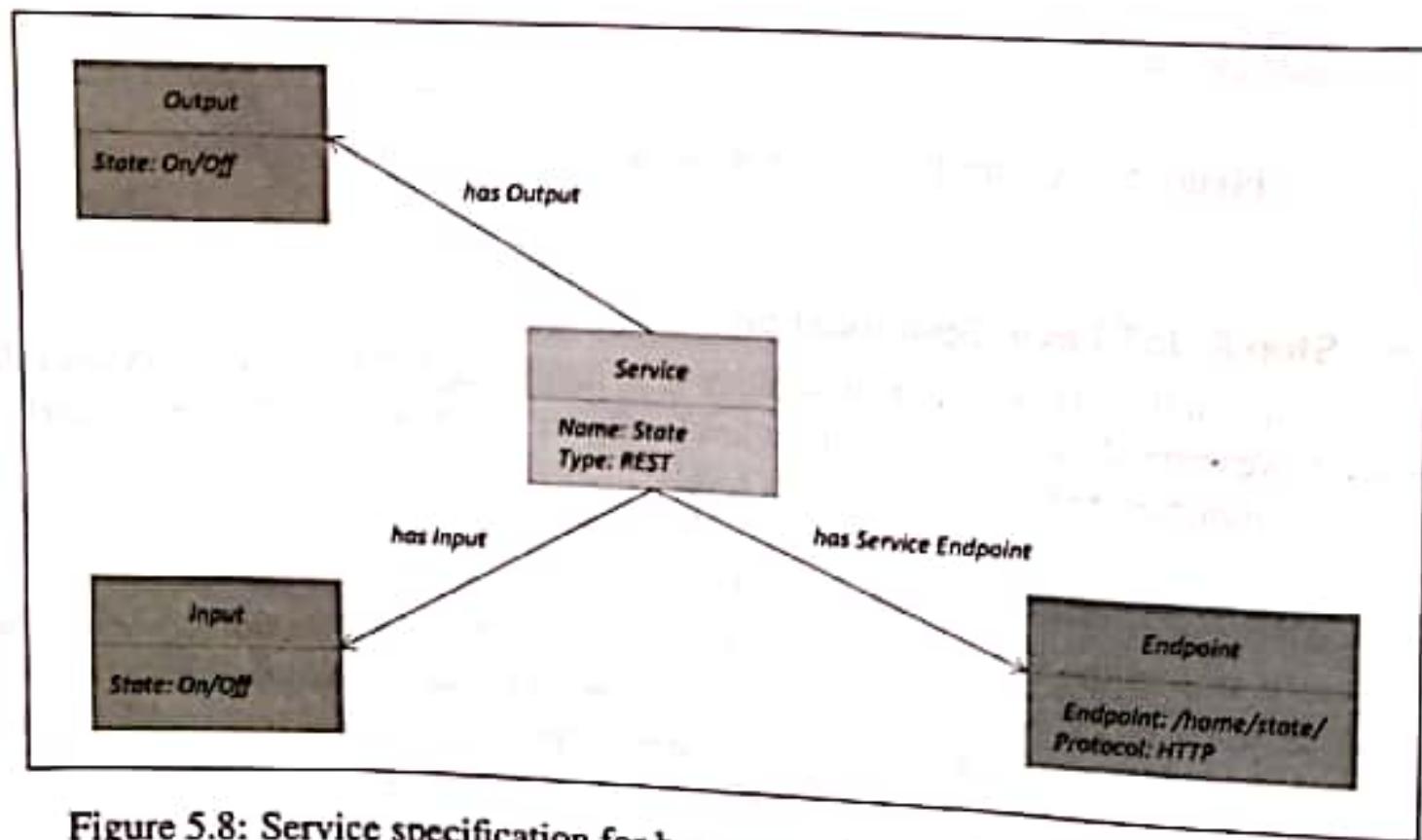


Figure 5.8: Service specification for home automation IoT system - state service

sensor and a relay switch (actuator).

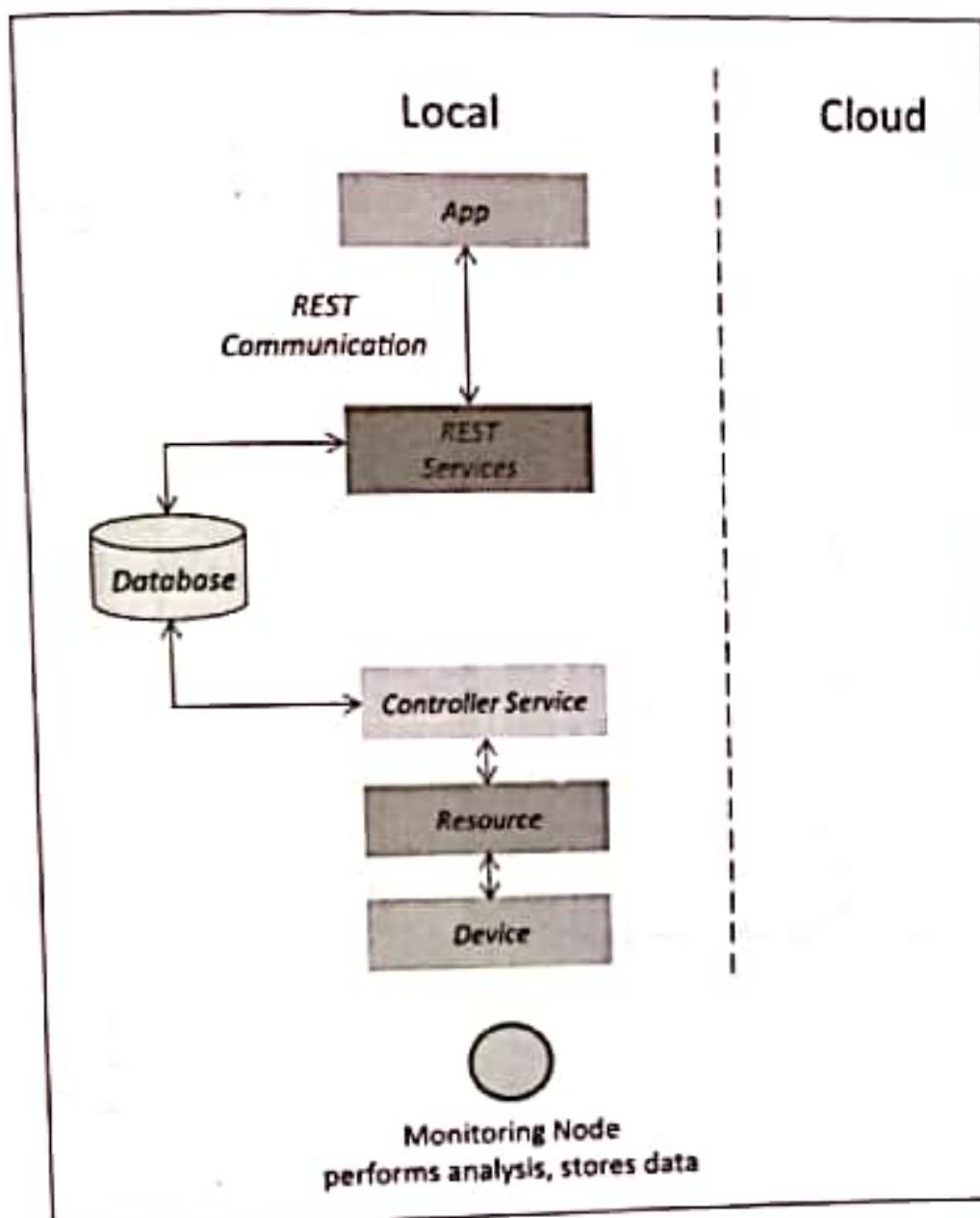


Figure 5.9: Deployment design of the home automation IoT system

- **Communication :** The communication FG handles the communication for the IoT system. The communication FG includes the communication protocols that form the backbone of IoT systems and enable network connectivity. You learned about various link, network, transport and application layer protocols in Chapter-1. The communication FG also includes the communication APIs (such as REST and WebSocket) that are used by the services and applications to exchange data over the network. In the home automation example the communication protocols include - 802.11 (link layer), IPv4/IPv6 (network layer), TCP (transport layer), and HTTP (application layer). The communication API used in the home automation examples is a REST-based API.
- **Services :** The service FG includes various services involved in the IoT system such

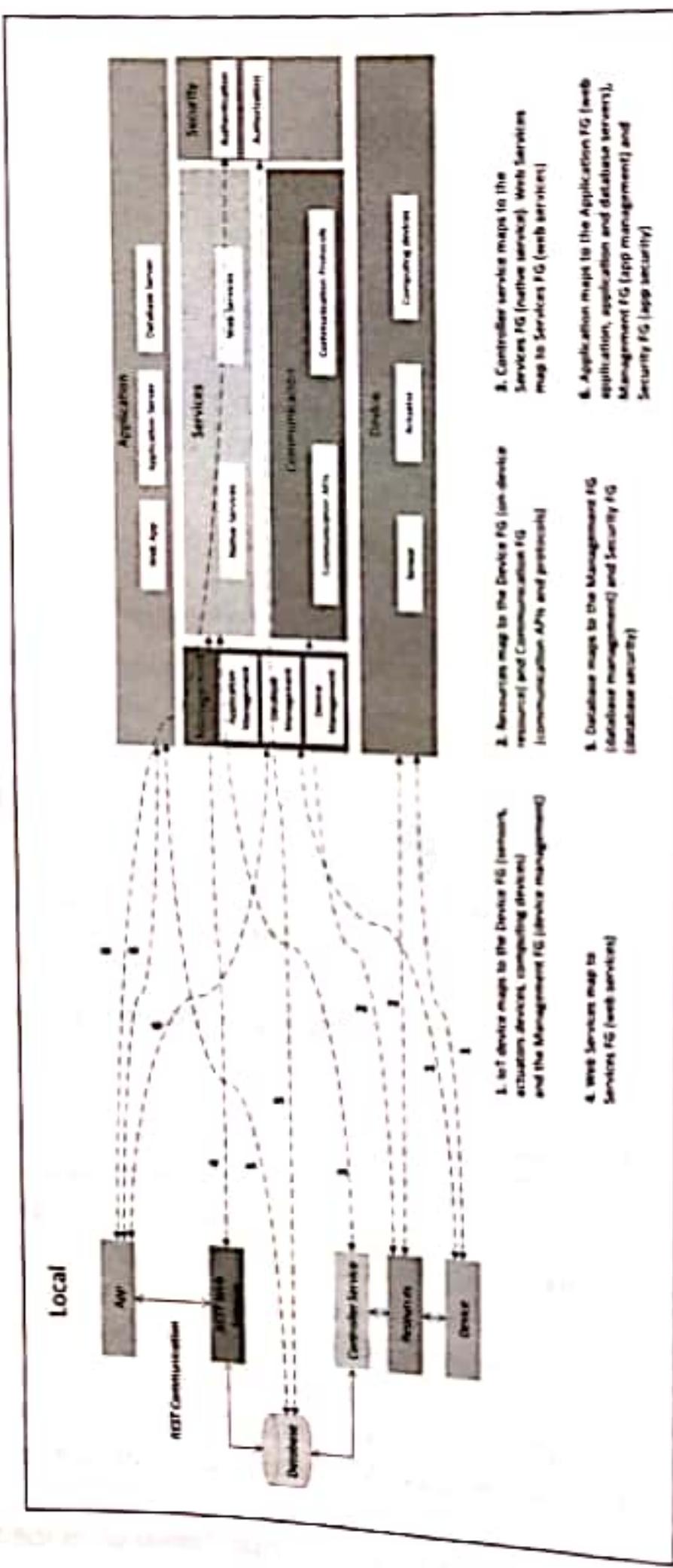


Figure 5.10: Mapping deployment level to functional groups for home automation IoT system

as services for device monitoring, device control services, data publishing services and services for device discovery. In the home automation example, there are two REST services (mode and state service) and one native service (controller service).

- **Management** : The management FG includes all functionalities that are needed to configure and manage the IoT system.
- **Security** : The security FG includes security mechanisms for the IoT system such as authentication, authorization, data security, etc.
- **Application** : The application FG includes applications that provide an interface to the users to control and monitor various aspects of the IoT system. Applications also allow users to view the system status and the processed data.

Figure 5.10 shows an example of mapping deployment level to functional groups for home automation IoT system.

IoT device maps to the Device FG (sensors, actuators devices, computing devices) and the Management FG (device management). Resources map to the Device FG (on-device resource) and Communication FG (communication APIs and protocols). Controller service maps to the Services FG (native service). Web Services map to Services FG . Database maps to the Management FG (database management) and Security FG (database security). Application maps to the Application FG (web application, application and database servers), Management FG (app management) and Security FG (app security).

5.2.8 Step 8: Operational View Specification

The eighth step in the IoT design methodology is to define the Operational View Specifications. In this step, various options pertaining to the IoT system deployment and operation are defined, such as, service hosting options, storage options, device options, application hosting options, etc.

Figure 5.11 shows an example of mapping functional groups to operational view specifications for home automation IoT system.

Operational View specifications for the home automation example are as follows:

- Devices: Computing device (Raspberry Pi), light dependent resistor (sensor), relay switch (actuator).
- Communication APIs: REST APIs
- Communication Protocols: Link Layer - 802.11, Network Layer - IPv4/IPv6, Transport - TCP, Application - HTTP.
- Services:
 1. Controller Service - Hosted on device, implemented in Python and run as a native service.
 2. Mode service - REST-ful web service, hosted on device, implemented with

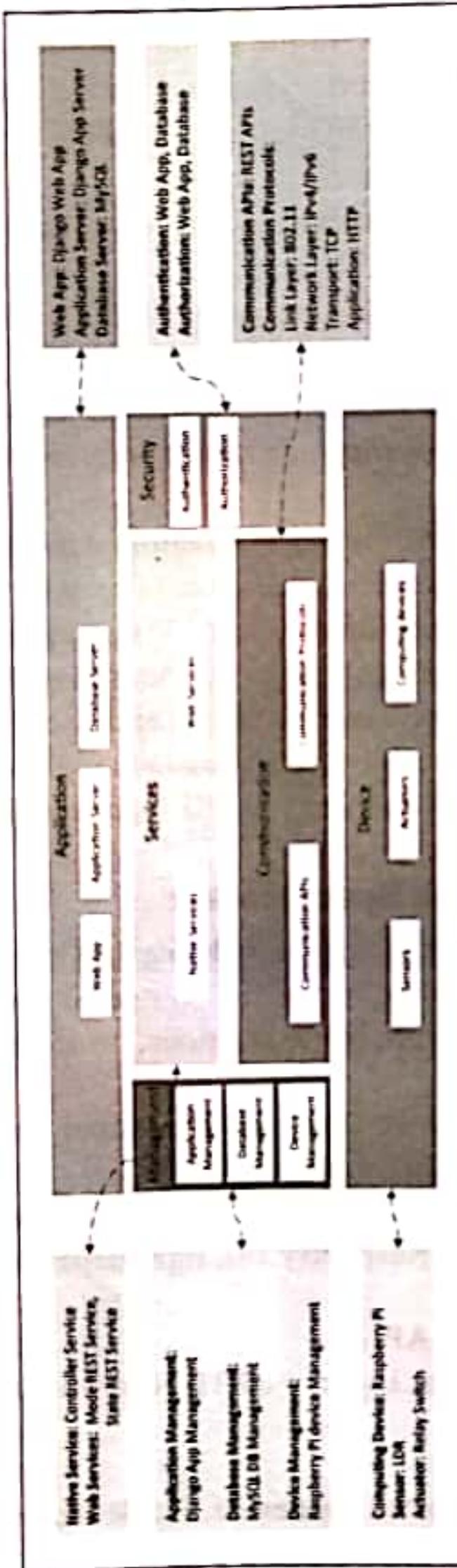


Figure 5.11: Mapping functional groups to operational view for home automation IoT system

Django-REST Framework.

3. State service - REST-ful web service, hosted on device, implemented with Django-REST Framework.

- Application:

Web Application - Django Web Application,

Application Server - Django App Server,

Database Server - MySQL.

- Security:

Authentication: Web App, Database

Authorization: Web App, Database

- Management:

Application Management - Django App Management

Database Management - MySQL DB Management,

Device Management - Raspberry Pi device Management.



5.2.9 Step 9: Device & Component Integration

The ninth step in the IoT design methodology is the integration of the devices and components. Figure 5.12 shows a schematic diagram of the home automation IoT system. The devices and components used in this example are Raspberry Pi mini computer, LDR sensor and relay switch actuator. A detailed description of Raspberry Pi board and how to interface sensors and actuators with the board is provided in later chapters.

5.2.10 Step 10: Application Development

(The final step in the IoT design methodology is to develop the IoT application. Figure 5.13 shows a screenshot of the home automation web application. The application has controls for the mode (auto on or auto off) and the light (on or off). In the auto mode, the IoT system controls the light appliance automatically based on the lighting conditions in the room. When auto mode is enabled the light control in the application is disabled and it reflects the current state of the light. When the auto mode is disabled, the light control is enabled and it is used for manually controlling the light.

5.3 Case Study on IoT System for Weather Monitoring

In this section we present a case study on design of an IoT system for weather monitoring using the IoT design methodology. The purpose of the weather monitoring system is to collect data on environmental conditions such as temperature, pressure, humidity and light

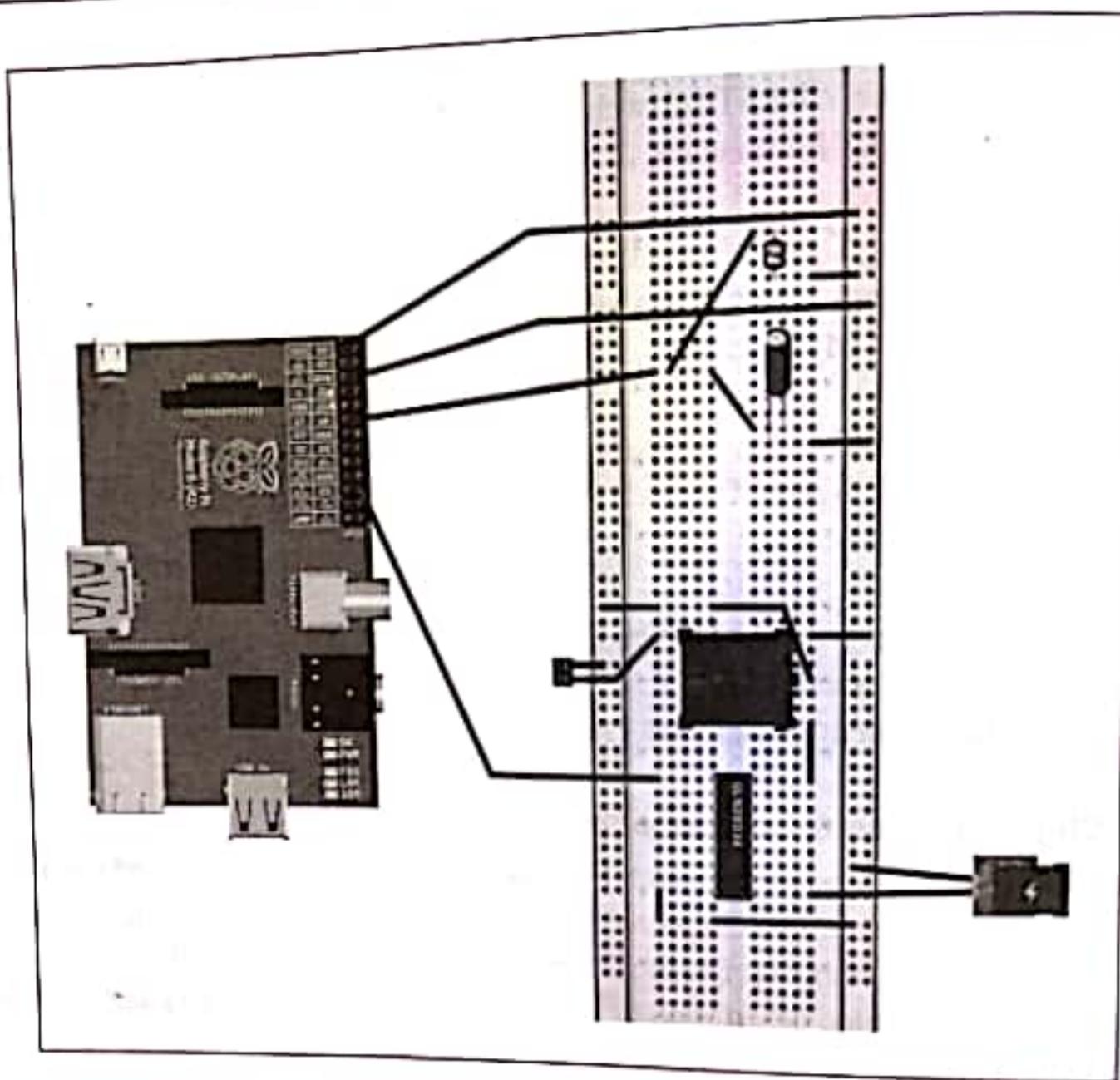


Figure 5.12: Schematic diagram of the home automation IoT system showing the device, sensor and actuator integrated

in an area using multiple end nodes. The end nodes send the data to the cloud where the data is aggregated and analyzed.

Figure 5.14 shows the process specification for the weather monitoring system. The process specification shows that the sensors are read after fixed intervals and the sensor measurements are stored.

Figure 5.15 shows the domain model for the weather monitoring system. In this domain model the physical entity is the environment which is being monitored. There is a virtual entity for the environment. Devices include temperature sensor, pressure sensor, humidity sensor, light sensor and single-board mini computer. Resources are software components which can be either on-device or network-resources. Services include the controller service that monitors the temperature, pressure, humidity and light and sends the readings to the

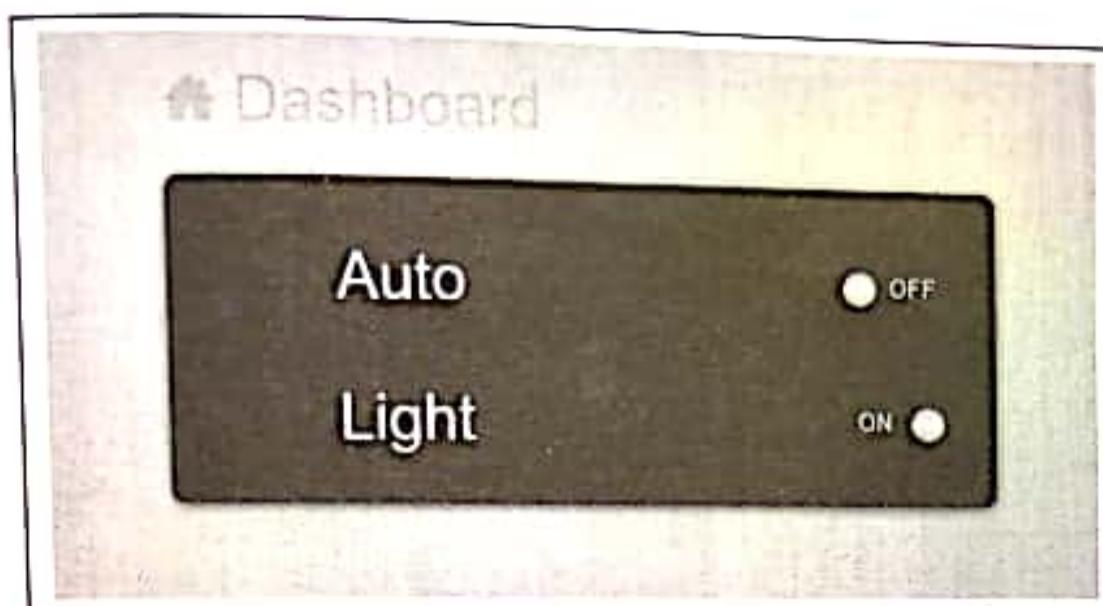


Figure 5.13: Home automation web application screenshot

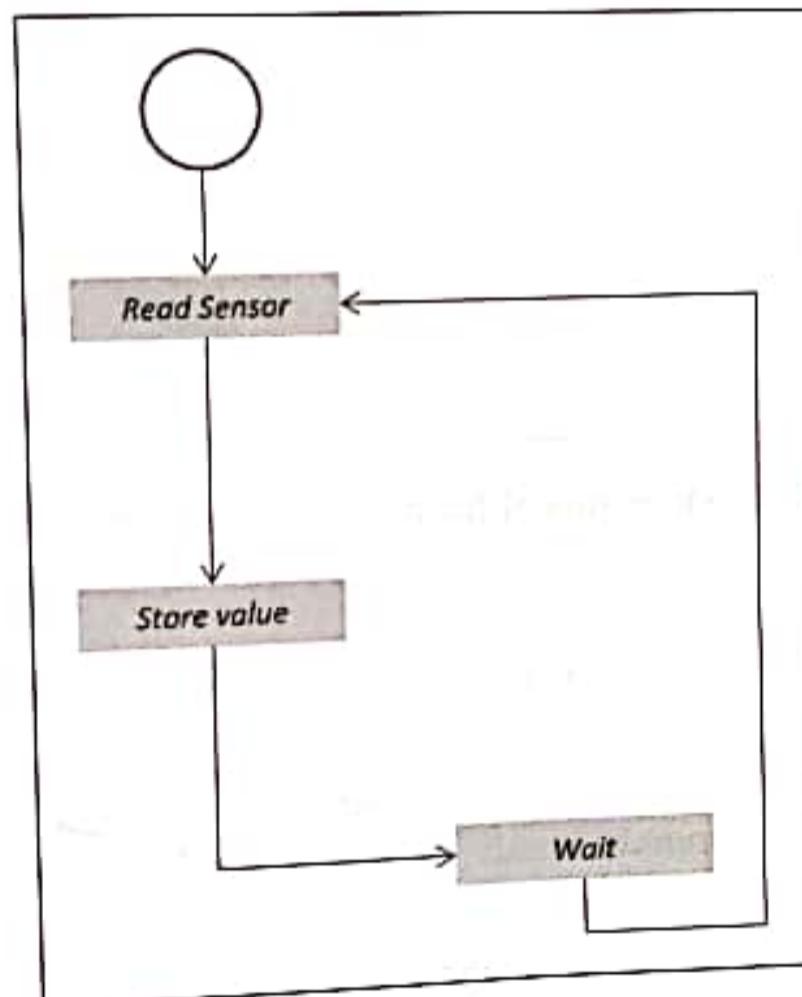


Figure 5.14: Process specification for weather monitoring IoT system

cloud.

Figure 5.16 shows the information model for the weather monitoring system. In this example, there is one virtual entity for the environment being sensed. The virtual entity has attributes - temperature, pressure, humidity and light. Figure 5.17 shows an example of

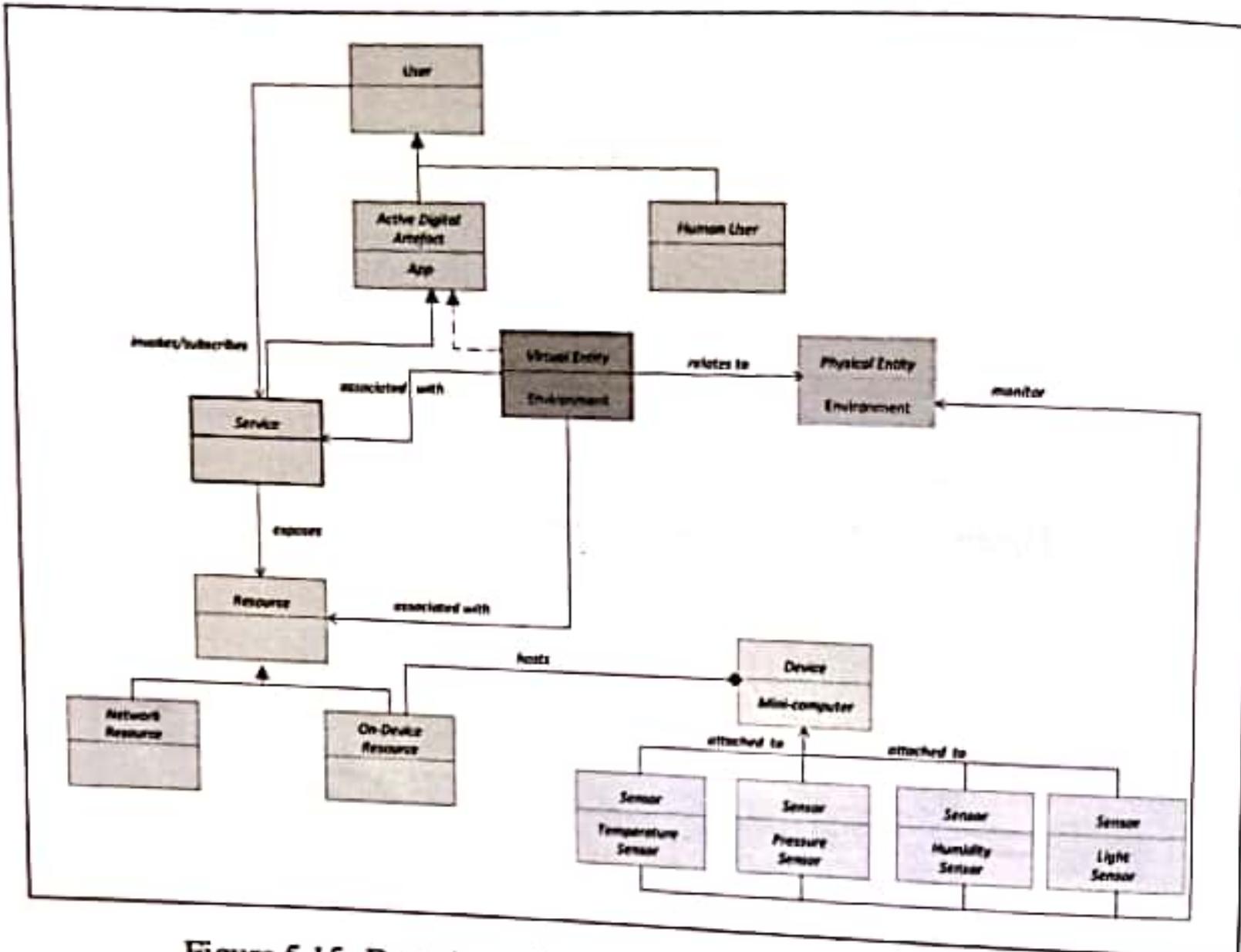


Figure 5.15: Domain model for weather monitoring IoT system

deriving the services from the process specification and information model for the weather monitoring system.

Figure 5.18 shows the specification of the controller service for the weather monitoring system. The controller service runs as a native service on the device and monitors temperature, pressure, humidity and light once every 15 seconds. The controller service calls the REST service to store these measurements in the cloud. In Chapter-8 we describe a Platform-as-a-Service called Xively that can be used for creating solutions for Internet of Things. An implementation of a controller service that calls the Xively REST API to store data in Xively cloud is described in Chapter-9.

Figure 5.19 shows the deployment design for the system. The system consists of multiple nodes placed in different locations for monitoring temperature, humidity and pressure in an area. The end nodes are equipped with various sensors (such as temperature, pressure, humidity and light). The end nodes send the data to the cloud and the data is stored in a cloud

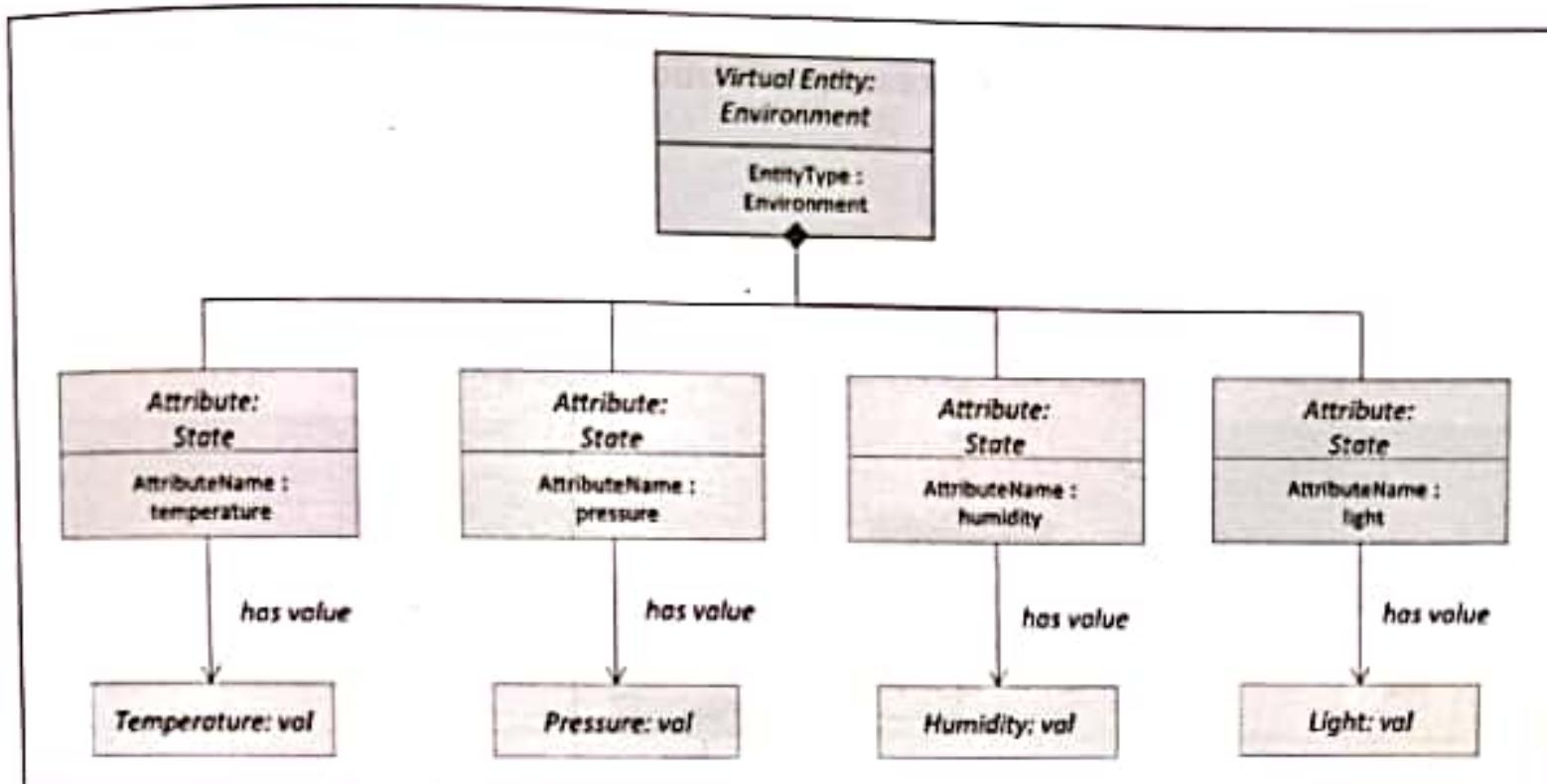


Figure 5.16: Information model for weather monitoring IoT system

database. The analysis of data is done in the cloud to aggregate the data and make predictions. A cloud-based application is used for visualizing the data. The centralized controller can send control commands to the end nodes, for example, to configure the monitoring interval on the end nodes.

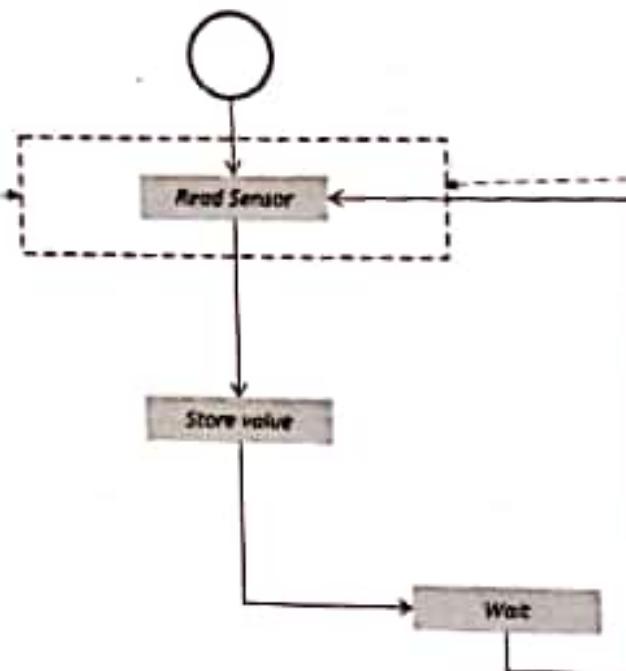
Figure 5.20 shows an example of mapping deployment level to functional groups for the weather monitoring system. Figure 5.21 shows an example of mapping functional groups to operational view specifications for the weather monitoring system.

Figure 5.22 shows a schematic diagram of the weather monitoring system. The devices and components used in this example are Raspberry Pi mini computer, temperature sensor, humidity sensor, pressure sensor and LDR sensor.

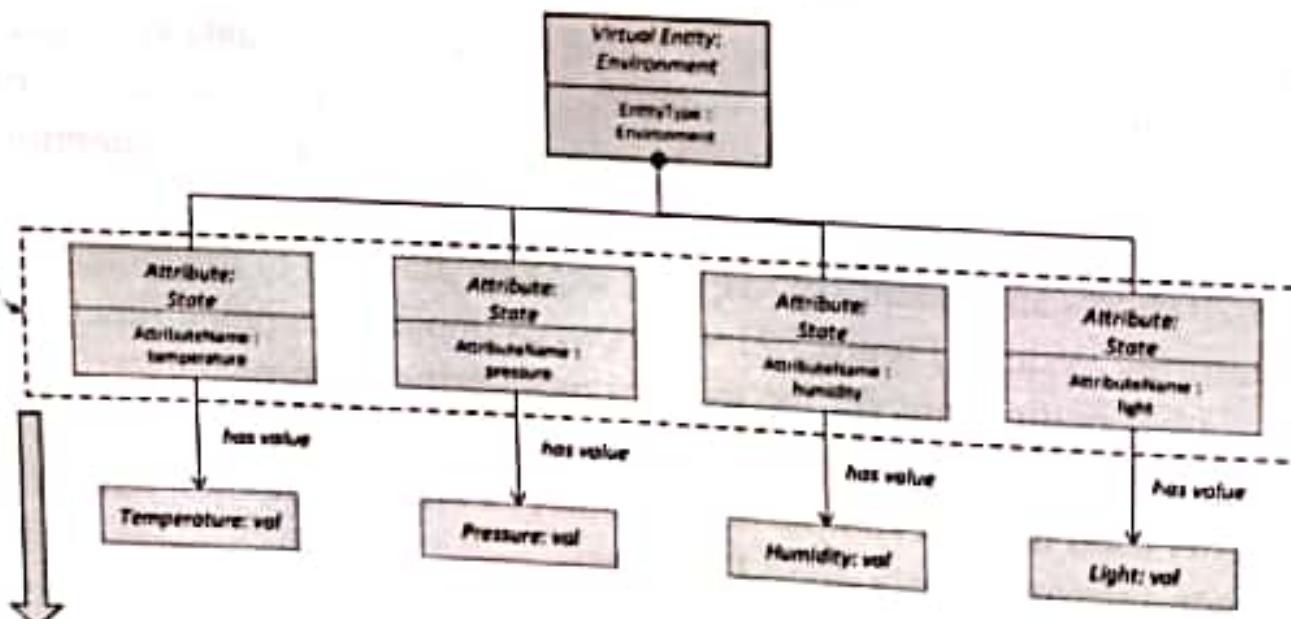
5.4 Motivation for Using Python

This book uses the Python language for all the examples, though the basic principles apply to other high level languages. In this section we explain the motivation for using Python for developing IoT systems. Python is a minimalistic language with English-like keywords and fewer syntactical constructions as compared to other languages. This makes Python easier to learn and understand. Moreover, Python code is compact as compared to other languages. Python is an interpreted language and does not require an explicit compilation step. The Python interpreter converts the Python code to the intermediate byte code, specific

Process Specification



Information Model



Controller Service: Runs as a native service on the device. Gets the current temperature, pressure, humidity and light readings and sends to the cloud database.

Figure 5.17: Deriving services from process specification and information model for weather monitoring IoT system

to the system. Python is supported on wide range of platforms, hence Python code is easily

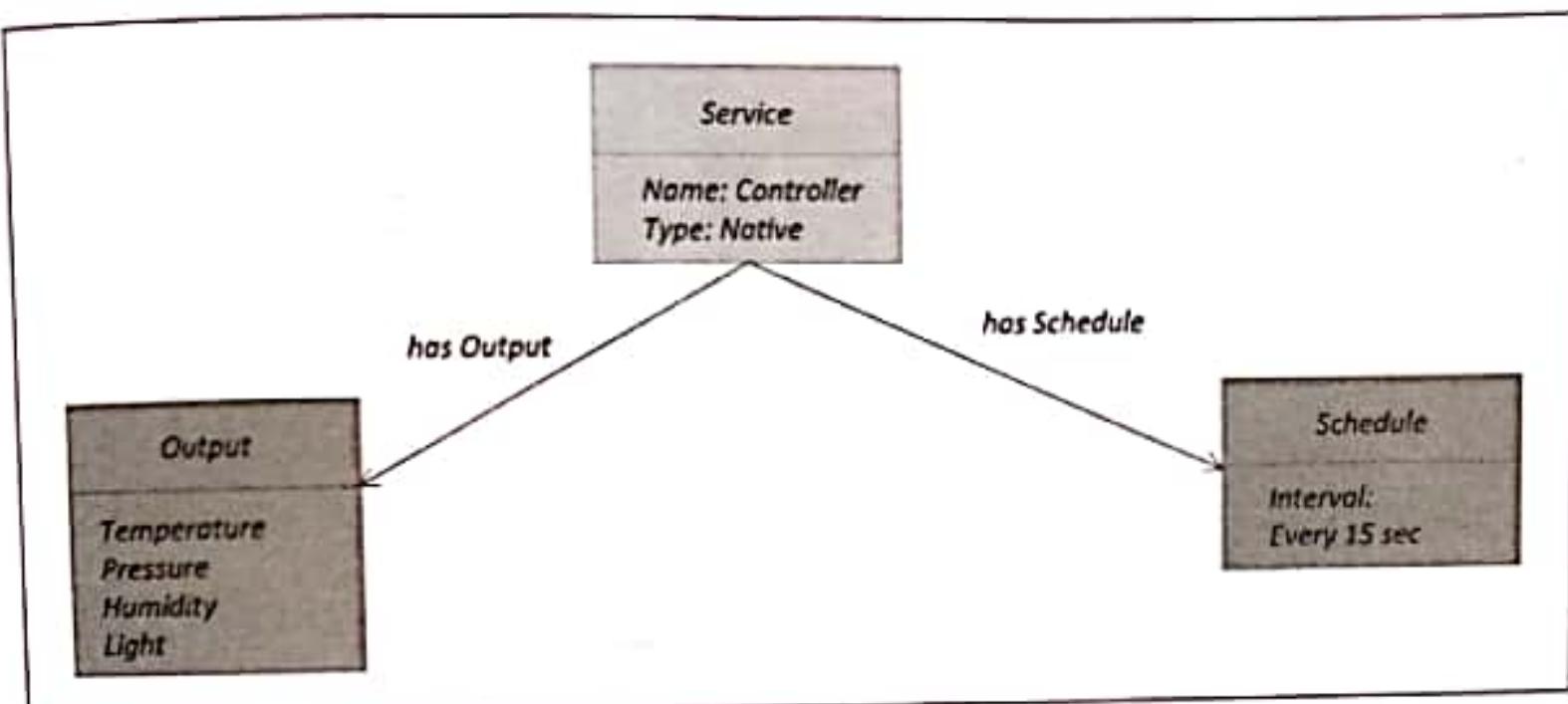


Figure 5.18: Controller service of the weather monitoring IoT system

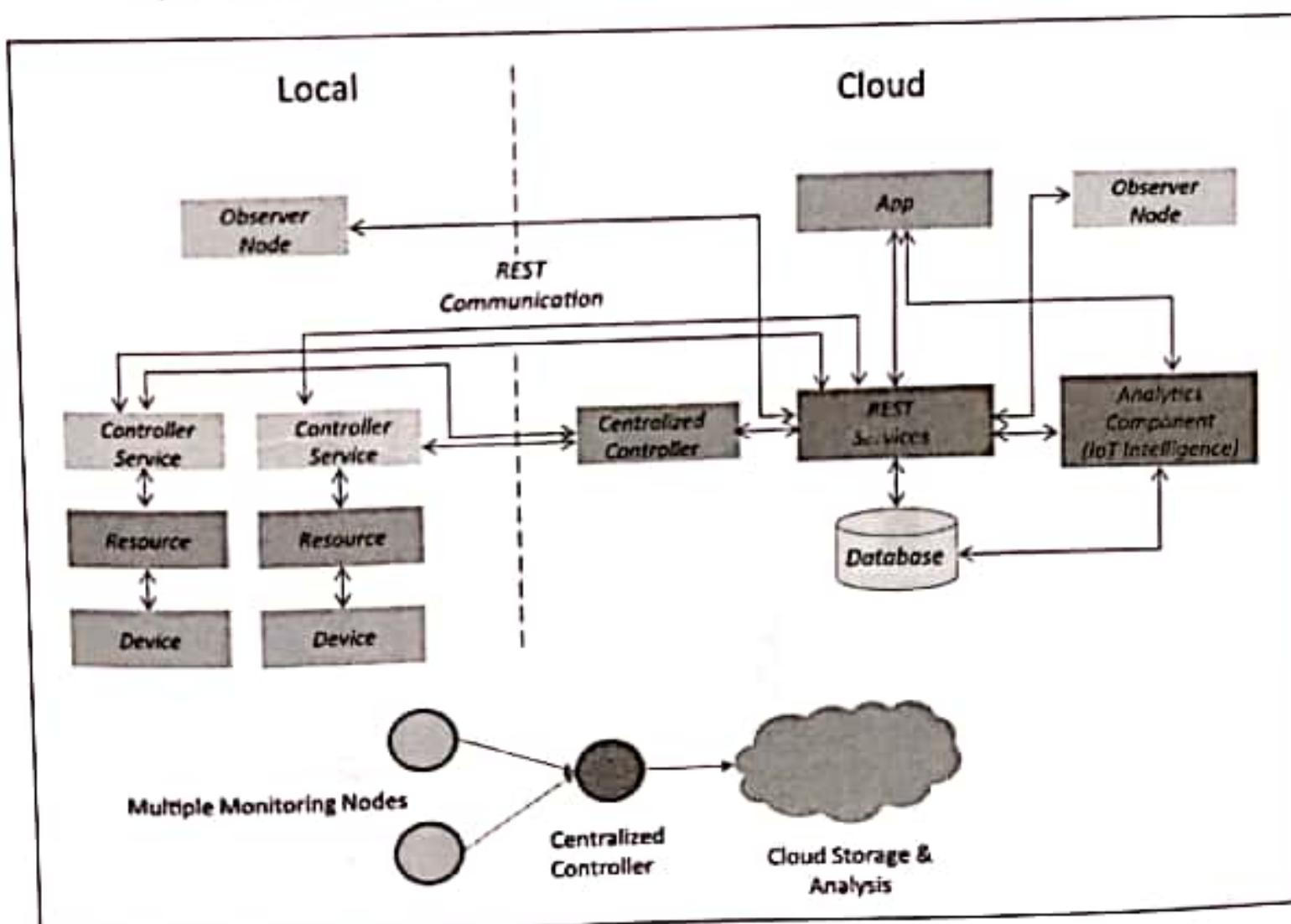


Figure 5.19: Deployment design of the weather monitoring IoT system

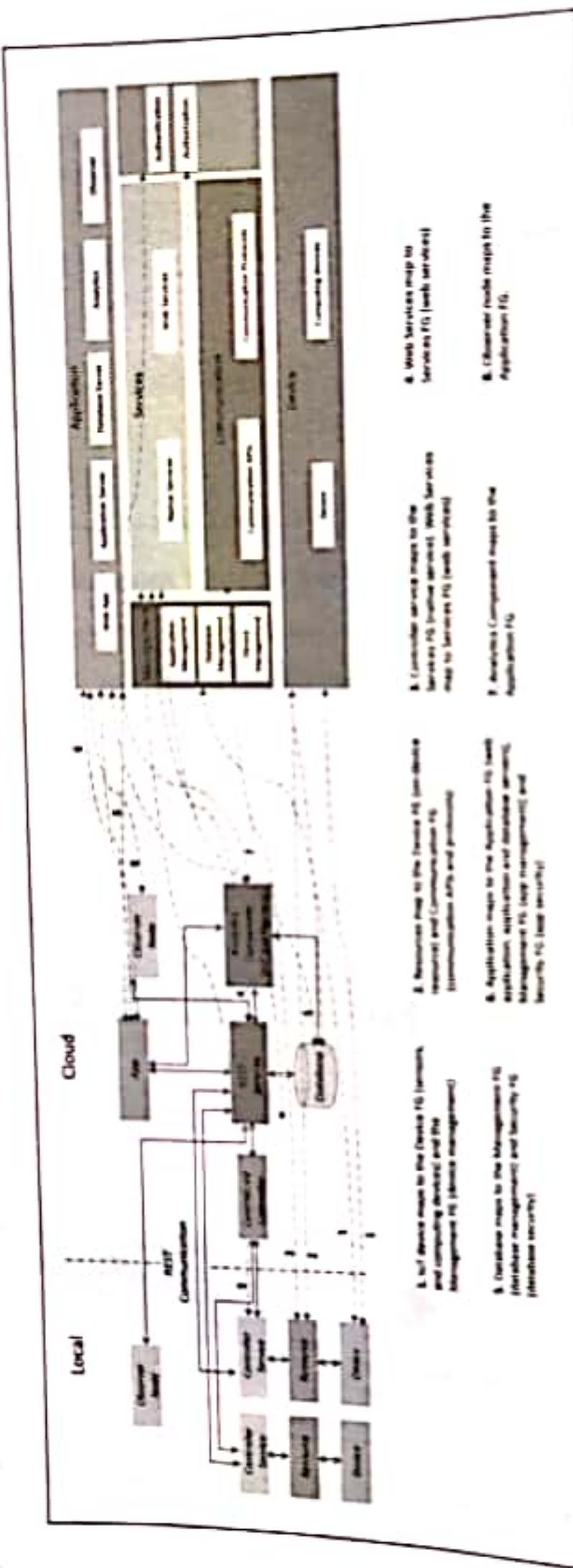


Figure 5.20: Mapping deployment level to functional groups for the weather monitoring IoT system

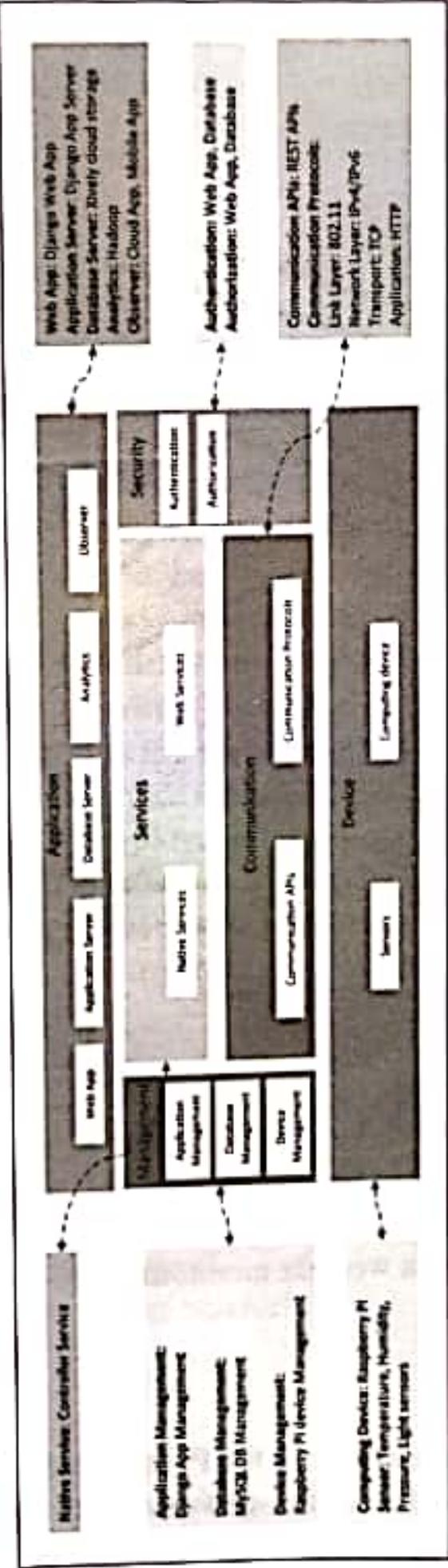
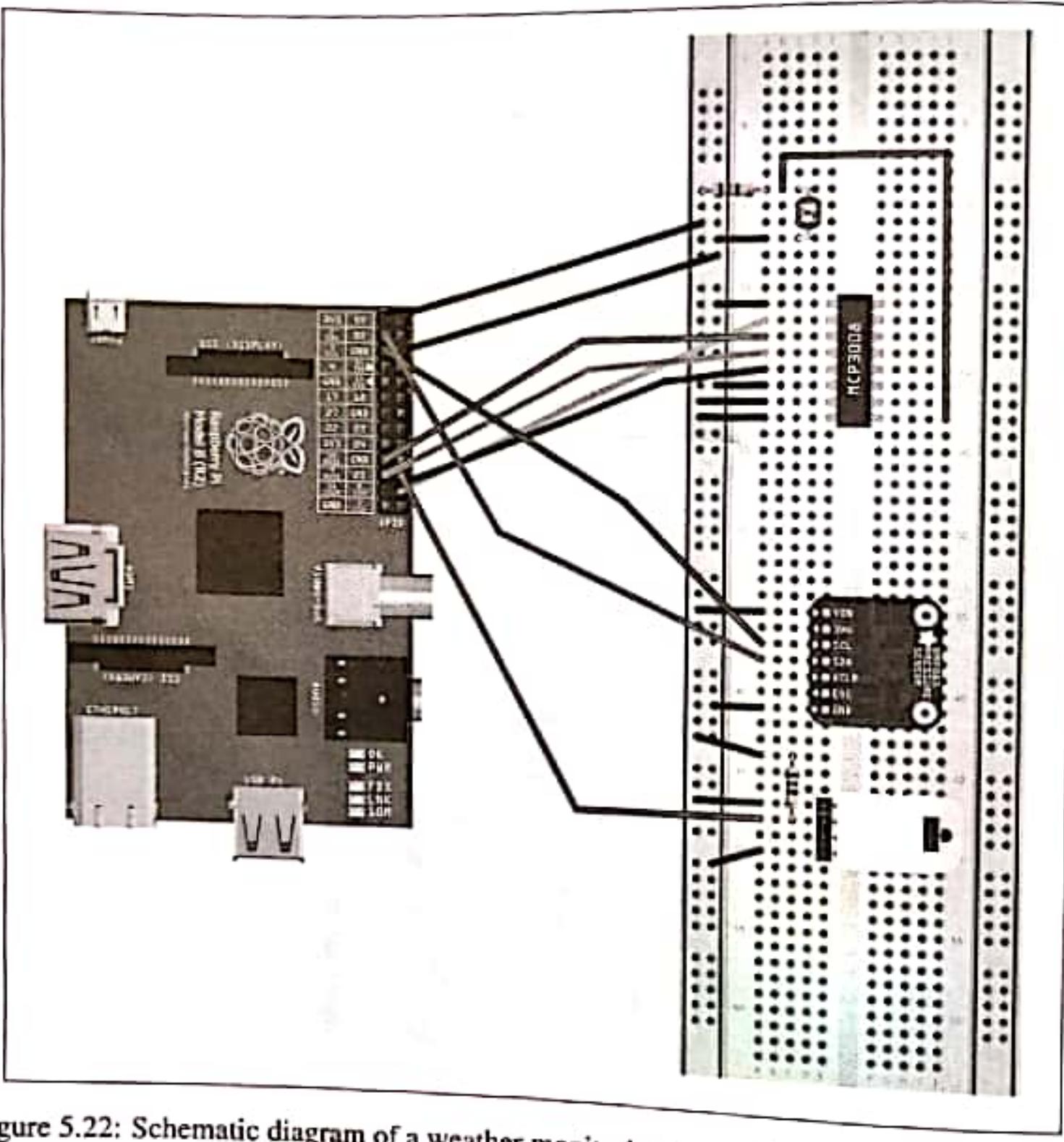


Figure 5.21: Mapping functional groups to operational view for the weather monitoring IoT system



web applications (e.g., Python web applications developed with Python web frameworks such as Django) and analytics components (e.g. machine learning components developed using Python libraries such as scikit-learn). In the next chapter you will learn the basics of Python language and all the related packages of interest that are used in the examples in this book.

Summary

In this chapter you learned about generic design methodology for IoT system design which is independent of specific product, service or programming language. The first step in IoT system design methodology is to define the purpose and requirements of the system. In the second step, the use cases of the IoT system are formally described based on the purpose and requirement specifications. The third step is to define the Domain Model which describes the main concepts, entities and objects in the domain of IoT system to be designed. The fourth step is to define the Information Model which defines the structure of all the information in the IoT system. The fifth step is to define the Functional View which defines the functions of the IoT systems grouped into various Functional Groups. The sixth step is to define the service specifications which define the services in the IoT system, service types, service inputs/output, service endpoints, service schedules, service preconditions and service effects. The seventh step is to define the Deployment & Operational View Specifications in which various options pertaining to the IoT system deployment and operation are defined. The eighth step is the integration of the devices and components. The final step in the IoT design methodology is to develop the IoT application.

Review Questions

1. What is the difference between a physical and virtual entity?
2. What is an IoT device?
3. What is the purpose of information model?
4. What are the various service types?
5. What is the need for a controller service?

6.1 Introduction

This book uses Python as the primary programming languages for examples. This chapter will help you in understanding the basics of Python programming and the Python packages that are used in examples in this book.

Python is a general-purpose high level programming language. Python 2.0 was released in the year 2000 and Python 3.0 was released in the year 2008. The 3.0 version is not backward compatible with earlier releases. The most recent release of Python is version 3.3. Currently, there is limited library support for the 3.x versions with operating systems such as Linux and Mac still using Python 2.x as default language. The exercises and examples in this book have been developed with Python version 2.7. The main characteristics of Python are:

1 Multi-paradigm programming language

Python supports more than one programming paradigms including object-oriented programming and structured programming

2 Interpreted Language

Python is an interpreted language and does not require an explicit compilation step. The Python interpreter executes the program source code directly, statement by statement, as a processor or scripting engine does.

3 Interactive Language

Python provides an interactive mode in which the user can submit commands at the Python prompt and interact with the interpreter directly.

The key benefits of Python are:

4 Easy-to-learn, read and maintain

Python is a minimalistic language with relatively few keywords, uses English keywords and has fewer syntactical constructions as compared to other languages. Reading Python programs is easy with pseudo-code like constructs. Python is easy to learn yet an extremely powerful language for a wide range of applications. Due to its simplicity, programs written in Python are generally easy to maintain.

5 Object and Procedure Oriented

Python supports both procedure-oriented programming and object-oriented programming. Procedure oriented paradigm allows programs to be written around procedures or functions that allow reuse of code. Procedure oriented paradigm allows programs to be written around objects that include both data and functionality.

6 Extendable

Python is an extendable language and allows integration of low-level modules written in languages such as C/C++. This is useful when you want to speed up a critical portion of a program.

7 Scalable

Due to the minimalistic nature of Python, it provides a manageable structure for large programs.

8 Portable

Since Python is an interpreted language, programmers do not have to worry about compilation, linking and loading of programs. Python programs can be directly executed from source code and copied from one machine to other without worrying about portability. The Python interpreter converts the source code to an intermediate form called byte codes and then translates this into the native language of your specific system and then runs it.

9 Broad Library Support

Python has a broad library support and works on various platforms such as Windows, Linux, Mac, etc. There are a large number of Python packages available for various applications such as machine learning, image processing, network programming, cryptography, etc.

6.2 Installing Python

Python is a highly portable language that works on various platforms such as Windows, Linux, Mac, etc. This section describes the Python installation steps for Windows and Linux:

Windows

Python binaries for Windows can be downloaded from <http://www.python.org/getit>. For the examples and exercise in this book, you would require Python 2.7 which can be directly downloaded from: <http://www.python.org/ftp/python/2.7.5/python-2.7.5.msi> Once the python binary is installed you can run the python shell at the command prompt using
> python

Linux

Box 6.1 provides the commands for installing Python on Ubuntu.

■ Box 6.1: Installing Python on Ubuntu Linux

```
G.I.T.Y
#Install Dependencies
sudo apt-get install build-essential
sudo apt-get install libreadline-gplv2-dev libncursesw5-dev
libssl-dev libsqlite3-dev tk-dev libgdbm-dev libc6-dev libbz2-dev

#Download Python
wget http://python.org/ftp/python/2.7.5/Python-2.7.5.tgz
tar -xvf Python-2.7.5.tgz
cd Python-2.7.5

#Install Python
./configure
make
sudo make install
```

6.3 Python Data Types & Data Structures

6.3.1 Numbers

Number data type is used to store numeric values. Numbers are immutable data types, therefore changing the value of a number data type results in a newly allocated object. Box 6.2 shows some examples of working with numbers.

■ Box 6.2: Working with Numbers in Python

```
#Integer
>>>a=5
>>>type(a)
<type 'int'>

#Floating Point
>>>b=2.5
>>>type(b)
<type 'float'>

#Long
>>>x=9898878787676L
>>>type(x)
<type 'long'>

#Complex
>>>y=2+5j
>>>y
```

```
(2+5j)
>>>type(y)
<type 'complex'>
>>>y.real
2
>>>y.imag
5

#Addition
>>>c=a+b
>>>c
7.5
>>>type(c)
<type 'float'>

#Subtraction
>>>d=a-b
>>>d
2.5
>>>type(d)
<type 'float'>

#Multiplication
>>>e=a*b
>>>e
12.5
>>>type(e)
<type 'float'>

#Division
>>>f=b/a
>>>f
0.5
>>>type(f)
<type 'float'>

#Power
>>>g=a**2
>>>g
25
```

6.3.2 Strings

A string is simply a list of characters in order. There are no limits to the number of characters you can have in a string. A string which has zero characters is called an empty string. Box 6.3

shows examples of working with strings.

■ Box 6.3: Working with Strings In Python

```
#Create string
>>>s="Hello World!"
>>>type(s)
<type 'str'>

#String concatenation
>>>t="This is sample program."
>>>r = s+t
>>>r
'Hello World!This is sample program.'

#Get length of string
>>>len(s)
12

#Convert string to integer
>>>x="100"
>>>type(s)
<type 'str'>
>>>y=int(x)
>>>y
100

#print string
>>>print s
Hello World!

#Formatting output
>>>print "The string (%s) has %d characters" % (s,len(s))
The string (Hello World!) has 12 characters

#Convert to upper/lower case
>>>s.upper()
'HELLO WORLD!'
>>>s.lower()
'hello world!'

#Accessing sub-strings
>>>s[0]
'H'
```

```
>>>s[6:]  
'World!'  
>>>s[6:-1]  
'World'  
  
#strip: Returns a copy of the string with  
#the leading and trailing characters removed.  
>>>s.strip("!")  
'Hello World'
```

6.3.3 Lists

List is a compound data type used to group together other values. List items need not all have the same type. A list contains items separated by commas and enclosed within square brackets. Box 6.4 shows examples of working with lists.

■ Box 6.4: Working with Lists In Python

```
>>>fruits=['apple','orange','banana','mango']  
>>>type(fruits)  
<type 'list'>  
>>>len(fruits)  
4  
  
>>>fruits[1]  
'orange'  
>>>fruits[1:3]  
['orange', 'banana']  
>>>fruits[1:]  
['orange', 'banana', 'mango']  
  
#Appending an item to a list  
>>>fruits.append('pear')  
>>>fruits  
['apple', 'orange', 'banana', 'mango', 'pear']  
  
#Removing an item from a list  
>>>fruits.remove('mango')  
>>>fruits  
['apple', 'orange', 'banana', 'pear']  
  
#Inserting an item to a list  
>>>fruits.insert(1,'mango')  
>>>fruits
```

```

['apple', 'mango', 'orange', 'banana', 'pear']

#Combining lists
>>>vegetables=['potato','carrot','onion','beans','radish']
>>>vegetables
['potato', 'carrot', 'onion', 'beans', 'radish']
>>>eatables=fruits+vegetables
>>>eatables
['apple', 'mango', 'orange', 'banana', 'pear',
'potato', 'carrot', 'onion', 'beans', 'radish']

#Mixed data types in a list
>>>mixed=['data',5,100.1,8287398L]
>>>type(mixed)
<type 'list'>
>>>type(mixed[0])
<type 'str'>
>>>type(mixed[1])
<type 'int'>
>>>type(mixed[2])
<type 'float'>
>>>type(mixed[3])
<type 'long'>

#It is possible to change individual elements of a list
>>>mixed[0]=mixed[0]+" items"
>>>mixed[1]=mixed[1]+1
>>>mixed[2]=mixed[2]+0.05
>>>mixed
['data items', 6, 100.14999999999999, 8287398L]

#Lists can be nested
>>>nested=[fruits,vegetables]
>>>nested
[['apple', 'mango', 'orange', 'banana', 'pear'],
['potato', 'carrot', 'onion', 'beans', 'radish']]

```

6.3.4 Tuples

A tuple is a sequence data type that is similar to the list. A tuple consists of a number of values separated by commas and enclosed within parentheses. Unlike lists, the elements of tuples cannot be changed, so tuples can be thought of as read-only lists. Box 6.5 shows examples of working with tuples.

■ Box 6.5: Working with Tuples in Python

```
>>>fruits=("apple","mango","banana","pineapple")
>>>fruits
('apple', 'mango', 'banana', 'pineapple')
>>>type(fruits)
<type 'tuple'>

#Get length of tuple
>>>len(fruits)
4

#Get an element from a tuple
>>>fruits[0]
'apple'
>>>fruits[:2]
('apple', 'mango')

#Combining tuples
>>>vegetables=('potato','carrot','onion','radish')
>>>eatables=fruits+vegetables
>>>eatables
('apple', 'mango', 'banana', 'pineapple',
 'potato', 'carrot', 'onion', 'radish')
```



6.3.5 Dictionaries

Dictionary is a mapping data type or a kind of hash table that maps keys to values. Keys in a dictionary can be of any data type, though numbers and strings are commonly used for keys. Values in a dictionary can be any data type or object. Box 6.6 shows examples on working with dictionaries.

■ Box 6.6: Working with Dictionaries in Python

```
>>>student={'name':'Mary','id':'8776','major':'CS'}
>>>student
{'major': 'CS', 'name': 'Mary', 'id': '8776'}
>>>type(student)
<type 'dict'>

#Get length of a dictionary
>>>len(student)
3
```

```

#Get the value of a key in dictionary
>>>student['name']
'Mary'

#Get all items in a dictionary
>>>student.items()
[('major', 'CS'), ('name', 'Mary'), ('id', '8776')]

#Get all keys in a dictionary
>>>student.keys()
['major', 'name', 'id']

#Get all values in a dictionary
>>>student.values()
['CS', 'Mary', '8776']

>>>student
{'major': 'CS', 'name': 'Mary', 'id': '8776'}

#A value in a dictionary can be another dictionary
>>>student1={'name':'David','id':'9876','major':'ECE'}
>>>students={'1': student,'2':student1}
>>>students
{'1': {'major': 'CS', 'name': 'Mary', 'id': '8776'},
 '2': {'major': 'ECE', 'name': 'David', 'id': '9876'}}

#Check if dictionary has a key
>>>student.has_key('name')
True
>>>student.has_key('grade')
False

```

6.3.6 Type Conversions

Box 6.7 shows examples of type conversions.

■ Box 6.7: Type conversion examples

```

#Convert to string
>>>a=10000
>>>str(a)
'10000'

```

```

#Convert to int
>>>b="2013"
>>>int(b)
2013

#Convert to float
>>>float(b)
2013.0

#Convert to long
>>>long(b)
2013L

#Convert to list
>>>s="aeiou"
>>>list(s)
['a', 'e', 'i', 'o', 'u']

#Convert to set
>>>x=['mango','apple','banana','mango','banana']
>>>set(x)
set(['mango', 'apple', 'banana'])

```

6.4 Control Flow

Lets us look at the control flow statements in Python.

6.4.1 if

The *if* statement in Python is similar to the *if* statement in other languages. Box 6.8 shows some examples of the *if* statement.

■ Box 6.8: if statement examples

```

>>>a = 25**5
>>>if a>10000:
    print "More"
else:
    print "Less"

More

>>>if a>10000:

```

```
if a<1000000:  
    print "Between 10k and 100k"  
else:  
    print "More than 100k"  
elif a==10000:  
    print "Equal to 10k"  
else:  
    print "Less than 10k"  
  
More than 100k  
  
>>>s="Hello World"  
>>>if "World" in s:  
    s=s+"!"  
    print s  
  
Hello World!  
  
>>>student={'name':'Mary','id':'8776'}  
>>>if not student.has_key('major'):  
    student['major']='CS'  
>>>student  
{'major': 'CS', 'name': 'Mary', 'id': '8776'}
```

6.4.2 for

The *for* statement in Python iterates over items of any sequence (list, string, etc.) in the order in which they appear in the sequence. This behavior is different from the *for* statement in other languages such as C in which an initialization, incrementing and stopping criteria are provided. Box 6.9 shows examples of the *for* statement.

■ Box 6.9: for statement examples

```
helloString = "Hello World"  
fruits=['apple','orange','banana','mango']  
student = {'name': 'Mary', 'id': '8776', 'major': 'CS'}  
  
#Looping over characters in a string  
for c in helloString:  
    print c  
  
#Looping over items in a list
```

```

i=0
for item in fruits:
    print "Fruit-%d: %s" % (i,item)
    i=i+1

#Looping over keys in a dictionary
for key in student:
    print "%s: %s" % (key,student[key])

```

6.4.3 while

The *while* statement in Python executes the statements within the *while* loop as long as the *while* condition is true. Box 6.10 shows a *while* statement example.

Box 6.10: while statement examples

```

#Prints even numbers upto 100
>>> i = 0
>>> while i<=100:
if i%2 == 0:
print i
i = i+1

```

6.4.4 range

The *range* statement in Python generates a list of numbers in arithmetic progression. Examples of *range* statement are shown in Box 6.11.

Box 6.11: range examples

```

#Generate a list of numbers from 0 - 9
>>>range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

#Generate a list of numbers from 10 - 100 with increments of 10
>>>range(10,110,10)
[10, 20, 30, 40, 50, 60, 70, 80, 90,100]

```

6.4.5 break/continue

The *break* and *continue* statements in Python are similar to the statements in C. The *break* statement breaks out of the for/while loop whereas the *continue* statement continues with the next iteration. Box 6.12 shows examples of *break* and *continue* usage.

Box 6.12: break/continue examples

```
#Break statement example
>>>y=1
>>>for x in range(4,256,4):
    y = y * x
    if y > 512:
        break
    print y

4
32
384

#Continue statement example
>>>fruits=['apple','orange','banana','mango']
>>>for item in fruits:
    if item == "banana":
        continue
    else:
        print item

apple
orange
mango
```

6.4.6 pass

The *pass* statement in Python is a null operation. The *pass* statement is used when a statement is required syntactically but you do not want any command or code to execute. Box 6.13 shows an example of *pass* statement.

Box 6.13: pass statement example

```
fruits=['apple','orange','banana','mango']
for item in fruits:
    if item == "banana":
        pass
    else:
        print item

apple
orange
```

mango

6.5 Functions

A function is a block of code that takes information in (in the form of parameters), does some computation, and returns a new piece of information based on the parameter information. A function in Python is a block of code that begins with the keyword *def* followed by the function name and parentheses. The function parameters are enclosed within the parenthesis. The code block within a function begins after a colon that comes after the parenthesis enclosing the parameters. The first statement of the function body can optionally be a documentation string or docstring. Box 6.14 shows an example of a function that computes the average grade given a dictionary containing student records.

■ Box 6.14: Example of a function in Python

```
students = { '1': {'name': 'Bob', 'grade': 2.5},  
            '2': {'name': 'Mary', 'grade': 3.5},  
            '3': {'name': 'David', 'grade': 4.2},  
            '4': {'name': 'John', 'grade': 4.1},  
            '5': {'name': 'Alex', 'grade': 3.8}}  
  
def averageGrade(students):  
    "This function computes the average grade"  
    sum = 0.0  
    for key in students:  
        sum = sum + students[key]['grade']  
    average = sum/len(students)  
    return average  
  
avg = averageGrade(students)  
print "The average grade is: %0.2f" % (avg)
```

Functions can have default values of the parameters. If a function with default values is called with fewer parameters or without any parameter, the default values of the parameters are used as shown in the example in Box 6.15.

■ Box 6.15: Example of function with default arguments

```
>>>def displayFruits(fruits=['apple','orange']):
```

```
print "There are %d fruits in the list" % (len(fruits))
for item in fruits:
    print item

#Using default arguments
>>>displayFruits()
apple
orange

>>>fruits = ['banana', 'pear', 'mango']
>>>displayFruits(fruits)
banana
pear
mango
```

(All parameters in the Python functions are passed by reference. Therefore, if a parameter is changed within a function the change also reflected back in the calling function. Box 6.16 shows an example of parameter passing by reference.

■ Box 6.16: Example of passing by reference

```
>>>def displayFruits(fruits):
    print "There are %d fruits in the list" % (len(fruits))
    for item in fruits:
        print item
    print "Adding one more fruit"
    fruits.append('mango')

>>>fruits = ['banana', 'pear', 'apple']
>>>displayFruits(fruits)
There are 3 fruits in the list
banana
pear
apple
Adding one more fruit

>>>print "There are %d fruits in the list" % (len(fruits))
There are 4 fruits in the list
```

Functions can also be called using keyword arguments that identify the arguments by the parameter name when the function is called. Box 6.17 shows examples of keyword arguments.

Box 6.17: Examples of keyword arguments

```

>>>def printStudentRecords(name, age=20, major='CS'):
    print "Name: " + name
    print "Age: " + str(age)
    print "Major: " + major

#This will give error as name is required argument
>>>printStudentRecords()
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
TypeError: printStudentRecords() takes at least 1 argument (0 given)

>>>printStudentRecords(name='Alex')
Name: Alex
Age: 20
Major: CS

>>>printStudentRecords(name='Bob', age=22, major='ECE')
Name: Bob
Age: 22
Major: ECE

>>>printStudentRecords(name='Alan', major='ECE')
Name: Alan
Age: 20
Major: ECE

#name is a formal argument.
#**kwargs is a keyword argument that receives all
#arguments except the formal argument as a dictionary.
>>>def student(name, **kwargs):
    print "Student Name: " + name
    for key in kwargs:
        print key + ': ' + kwargs[key]

>>>student(name='Bob', age='20', major = 'CS')
Student Name: Bob
age: 20
major: CS

```

Python functions can have variable length arguments. These variable length arguments are passed as a tuple to the function with an argument prefixed with asterix (*) as shown in

Box 6.18.

■ Box 6.18: Example of variable length arguments

```
def student(name, *varargs):
    print "Student Name: " + name
    for item in varargs:
        print item

>>>student('Nav')
Student Name: Nav

>>>student('Amy', 'Age: 24')
Student Name: Amy
Age: 24

>>>student('Bob', 'Age: 20', 'Major: CS')
Student Name: Bob
Age: 20
Major: CS
```

6.6 Modules

Python allows organizing of the program code into different modules which improves the code readability and management. A module is a Python file that defines some functionality in the form of functions or classes. Modules can be imported using the import keyword. Modules to be imported must be present in the search path. Box 6.19 shows the example of a student module that contains two functions and Box 6.20 shows an example of importing the student module and using it.

■ Box 6.19: Module student

```
def averageGrade(students):
    sum = 0.0
    for key in students:
        sum = sum + students[key]['grade']
    average = sum/len(students)
    return average

def printRecords(students):
    print "There are %d students" %(len(students))
```

```
i=1
for key in students:
    print "Student-%d: " % (i)
    print "Name: " + students[key]['name']
    print "Grade: " + str(students[key]['grade'])
    i = i+1
```

■ Box 6.20: Using module student

```
>>>import student

>>>students = {'1': {'name': 'Bob', 'grade': 2.5,
                     '2': {'name': 'Mary', 'grade': 3.5,
                     '3': {'name': 'David', 'grade': 4.2,
                     '4': {'name': 'John', 'grade': 4.1,
                     '5': {'name': 'Alex', 'grade': 3.8

>>>student.printRecords(students)
There are 5 students
Student-1:
Name: Bob
Grade: 2.5

Student-2:
Name: David
Grade: 4.2

Student-3:
Name: Mary
Grade: 3.5

Student-4:
Name: Alex
Grade: 3.8

Student-5:
Name: John
Grade: 4.1

>>>avg = student.averageGrade(students)
>>>print "The average grade is: %.2f" % (avg)
3.62
```

The import keyword followed by the module name imports all the functions in the module. If you want to use only a specific function it is recommended to import only that function using the keyword from as shown in the example in Box 6.21.

■ Box 6.21: Importing a specific function from a module

```
>>>from student import averageGrade

>>>students = {'1': 'name': 'Bob', 'grade': 2.5,
   '2': 'name': 'Mary', 'grade': 3.5,
   '3': 'name': 'David', 'grade': 4.2,
   '4': 'name': 'John', 'grade': 4.1,
   '5': 'name': 'Alex', 'grade': 3.8

>>>avg = averageGrade(students)
>>>print "The average grade is: %0.2f" % (avg)
3.62
```

Python comes with a number of standard modules such as system related modules (sys), OS related module (os), mathematical modules (math, fractions, etc.), Internet related modules (email, json, etc), etc. The complete list of standard modules is available in the Python documentation [87]. Box 6.22 shows an example of listing all names defined in a module using the built-in dir function.

■ Box 6.22: Listing all names defined in a module

```
>>>import email

>>>dir (email)
['Charset', 'Encoders', 'Errors', 'FeedParser', 'Generator', 'Header',
 'Iterators', 'LazyImporter', 'MIMEAudio', 'MIMEBase', 'MIMEImage',
 'MIMEMessage', 'MIMEMultipart', 'MIMENonMultipart', 'MIMEText',
 'Message', 'Parser', 'Utils', '_LOWERNAMES', '_MIMENAMES',
 '__all__', '__builtins__', '__doc__', '__file__',
 '__name__', '__package__', '__path__', '__version__',
 '__name__', 'base64MIME', 'email', 'importer', 'message_from_file',
 'message_from_string', 'mime', 'quopriMIME', 'sys']
```

6.7 Packages

6.7 Packages

Python package is hierarchical file structure that consists of modules and subpackages. Packages allow better organization of modules related to a single application environment. For example, Box 6.23 shows the listing of the skimage package that provides image processing algorithms. The package is organized into a root directory (skimage) with sub-directories (color, draw, etc) which are sub-packages within the skimage package. Each directory contains a special file named `__init__.py` which tells Python to treat directories as packages. This file can either be an empty file or contain some initialization code for the package.

■ Box 6.23: skimage package listing

```
skimage/                                Top level package
    __init__.py                           Treat directory as a package
    color/                                 color subpackage
        __init__.py
        colorconv.py
        colorlabel.py
        rgb_colors.py
    draw/                                  draw subpackage
        __init__.py
        draw.py
        setup.py
    exposure/                             exposure subpackage
        __init__.py
        _adapthist.py
        exposure.py
    feature/                             feature subpackage
        __init__.py
        _brief.py
        _daisy.py
    ...
    ...
```

6.8 File Handling

Python allows reading and writing to files using the file object. The `open(filename, mode)` function is used to get a file object. The mode can be `read (r)`, `write (w)`, `append (a)`, `read and write (r+ or w+)`, `read-binary (rb)`, `write-binary (wb)`, etc. Box 6.24 shows an example of reading an entire file with `read` function. After the file contents have been read the `close` function is called which closes the file object.

■ Box 6.24: Example of reading an entire file

```
>>>fp = open('file.txt','r')
>>>content = fp.read()
>>>print content
Python supports more than one programming paradigms
including object-oriented programming and structured
programming
Python is an interpreted language and does
not require an explicit compilation
step. >>>fp.close()
```

Box 6.25 shows an example of reading line by line from a file using the readline function.

■ Box 6.25: Example of reading line by line

```
>>>fp.close()
>>>fp = open('file.txt','r')
>>>print "Line-1: " + fp.readline()
Line-1: Python supports more than one programming paradigms
including object-oriented programming and structured
programming

>>>print "Line-2: " + fp.readline()
Line-2: Python is an interpreted language and does not
require an explicit compilation step.

>>>fp.close()
```

Box 6.26 shows an example of reading lines of a file in a loop using the readlines function.

■ Box 6.26: Example of reading lines in a loop

```
>>>fp = open('file.txt','r')
>>>lines = fp.readlines()
>>>for line in lines:
    print line

Python supports more than one programming paradigms including
object-oriented programming and structured programming
```

Python is an interpreted language and does not require an explicit compilation step.

Box 6.27 shows an example of reading a certain number of bytes from a file using the `read(size)` function.

■ **Box 6.27: Example of reading a certain number of bytes**

```
>>>fp = open('file.txt','r')
>>>fp.read(10)
'Python sup'
>>>fp.close()
```

Box 6.28 shows an example of getting the current position of read using the `tell` function.

■ **Box 6.28: Example of getting the current position of read**

```
>>>fp = open('file.txt','r')
>>>fp.read(10)
'Python sup'
>>>currentpos = fp.tell
>>>print currentpos
<built-in method tell of file object at 0x0000000002391390>
>>>fp.close()
```

Box 6.29 shows an example of seeking to a certain position in a file using the `seek` function.

■ **Box 6.29: Example of seeking to a certain position**

```
>>>fp = open('file.txt','r')
>>>fp.seek(10,0)
>>>content = fp.read(10)
>>>print content
Ports more
>>>fp.close()
```

Box 6.30 shows an example of writing a file using the `write` function.

■ Box 6.30: Example of writing to a file

```
>>>fo = open('file1.txt','w')
>>>content='This is an example of writing to a file in Python.'
>>>fo.write(content)
>>>fo.close()
```

6.9 Date/Time Operations

Python provides several functions for date and time access and conversions. The datetime module allows manipulating date and time in several ways. Box 6.31 shows examples of manipulating with date.

■ Box 6.31: Examples of manipulating with date

```
>>>from datetime import date
>>>now = date.today()
>>>print "Date: " + now.strftime("%m-%d-%y")
Date: 07-24-13
>>>print "Day of Week: " + now.strftime("%A")
Day of Week: Wednesday
>>>print "Month: " + now.strftime("%B")
Month: July
>>>
>>>then = date(2013, 6, 7)
>>>timediff = now - then
>>>timediff.days
47
```

The time module in Python provides various time-related functions. Box 6.32 shows examples of manipulating with time.

■ Box 6.32: Examples of manipulating with time

```
>>>import time
>>>nowtime = time.time()
>>>time.localtime(nowtime)
time.struct_time(tm_year=2013, tm_mon=7, tm_mday=24, tm_
sec=51, tm_wday=2, tm_yday=205, tm_isdst=0)
>>>time.asctime(time.localtime(nowtime)) *
```

'Wed Jul 24 16:14:51 2013'

```
>>>time.strftime("The date is %d-%m-%y.  
Today is a %A. It is %H hours, %M minutes and %S seconds now."  
'The date is 24-07-13. Today is a Wednesday. It is 16 hours,  
15 minutes and 14 seconds now.'
```

6.10 Classes

Python is an Object-Oriented Programming (OOP) language. Python provides all the standard features of Object Oriented Programming such as classes, class variables, class methods, inheritance, function overloading, and operator overloading. Let us briefly look at these OOP concepts:

Class

A class is simply a representation of a type of object and user-defined prototype for an object that is composed of three things: a name, attributes, and operations/methods.

Instance/Object

Object is an instance of the data structure defined by a class.

Inheritance

Inheritance is the process of forming a new class from an existing class or base class.

Function overloading

Function overloading is a form of polymorphism that allows a function to have different meanings, depending on its context.

Operator overloading

Operator overloading is a form of polymorphism that allows assignment of more than one function to a particular operator.

Function overriding

Function overriding allows a child class to provide a specific implementation of a function that is already provided by the base class. Child class implementation of the overridden function has the same name, parameters and return type as the function in the base class.

Box 6.33 shows an example of a Class. The variable *studentCount* is a class variable that is shared by all instances of the class *Student* and is accessed by *Student.studentCount*. The variables *name*, *id* and *grades* are instance variables which are specific to each instance of the class. There is a special method by the name *__init__()* which is the class constructor.

The class constructor initializes a new instance when it is created. The function `__del__()` is the class destructor.

■ Box 6.33: Examples of a class

```
>>>class Student:  
    studentCount = 0  
    def __init__(self, name, id):  
        print "Constructor called"  
        self.name = name  
        self.id = id  
        Student.studentCount = Student.studentCount + 1  
        self.grades = {}  
  
    def __del__(self):  
        print "Destructor called"  
  
    def getStudentCount(self):  
        return Student.studentCount  
  
    def addGrade(self, key, value):  
        self.grades[key] = value  
  
    def getGrade(self, key):  
        return self.grades[key]  
  
    def printGrades(self):  
        for key in self.grades:  
            print key + ": " + self.grades[key]  
  
>>>s = Student('Steve', '98928')  
Constructor called  
>>>s.addGrade('Math', '90')  
>>>s.addGrade('Physics', '85')  
>>>s.printGrades()  
Physics: 85  
Math: 90  
>>>mathgrade = s.getGrade('Math')  
>>>print mathgrade  
90  
>>>count = s.getStudentCount()  
>>>print count  
1  
>>>del s
```

Box 6.34 shows an example of class inheritance. In this example *Shape* is the base class and *Circle* is the derived class. The class *Circle* inherits the attributes of the *Shape* class. The child class *Circle* overrides the methods and attributes of the base class (eg. *draw()* function defined in the base class *Shape* is overridden in child class *Circle*). It is possible to hide some class attributes by naming them with a *double underscore* prefix. For example, *_label* attribute is hidden and cannot be directly accessed using the object (*circ._label* gives an error). To hide the attributes with double underscore prefix, Python changes their names internally and prefixes the class name (e.g. *_label* is changed to *_Circle_label*).

■ Box 6.34: Examples of class inheritance

```

>>>class Shape:
    def __init__(self):
        print "Base class constructor"
        self.color = 'Green'
        self.lineWeight = 10.0
    def draw(self):
        print "Draw - to be implemented"
    def setColor(self, c):
        self.color = c
    def getColor(self):
        return self.color
    def setLineWeight(self, lwt):
        self.lineWeight = lwt
    def getLineWeight(self):
        return self.lineWeight

>>>class Circle(Shape):
    def __init__(self, c, r):
        print "Child class constructor"
        self.center = c
        self.radius = r
        self.color = 'Green'
        self.lineWeight = 10.0
        self.__label = 'Hidden circle label'
    def setCenter(self, c):
        self.center = c
    def getCenter(self):
        return self.center
    def setRadius(self, r):

```

```
    self.radius = r
def getRadius(self):
    return self.radius
def draw(self):
    print "Draw Circle (overridden function)"

>>>class Point:
    def __init__(self, x, y):
        self.xCoordinate = x
        self.yCoordinate = y

    def setXCoordinate(self,x):
        self.xCoordinate = x
    def getXCoordinate(self):
        return self.xCoordinate
    def setYCoordinate(self,y):
        self.yCoordinate = y
    def getYCoordinate(self):
        return self.yCoordinate

>>>p = Point(2,4)
>>>circ = Circle(p,7)
Child class constructor
>>>circ.getColor()
'Green'
>>>circ.setColor('Red')
>>>circ.getColor()
'Red'

>>>circ.getLineWidth()
10.0
>>>circ.getCenter().getXCoordinate()
2
>>>circ.getCenter().getYCoordinate()
4

>>>circ.draw()
Draw Circle (overridden function)
>>>circ.radius
7
>>>circ.__label
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
AttributeError: Circle instance has no attribute '__label'
```

```
>>>circ._Circle_label  
'Hidden circle label'
```

6.11 Python Packages of Interest for IoT

6.11.1 JSON

JavaScript Object Notation (JSON) is an easy to read and write data-interchange format.
JSON is used as an alternative to XML and is is easy for machines to parse and generate.
JSON is built on two structures - a collection of name-value pairs (e.g. a Python dictionary)
and ordered lists of values (e.g., a Python list).

JSON format is often used for serializing and transmitting structured data over a network connection, for example, transmitting data between a server and web application. Box 6.35 shows an example of a Twitter tweet object encoded as JSON.

■ Box 6.35: JSON Example - A Twitter tweet object

```
{  
    "created_at":  
        "Sat Jun 01 11:39:43 +0000 2013",  
    "id":340794787059875841,  
    "text":"What a bright and sunny day today!",  
    "truncated":false,  
    "in_reply_to_status_id":null,  
    "user":{  
        "id":383825039,  
        "name":"Harry",  
        "followers_count":316,  
        "friends_count":298,  
        "listed_count":0,  
        "created_at":"Sun Oct 02 15:51:16 +0000 2011",  
        "favorites_count":251,  
        "statuses_count":1707,  
        :  
        "notifications":null  
    },  
    "geo":{  
        "type":"Point",  
        "coordinates":[26.92782727,75.78908449]  
    },  
    "coordinates":{
```

```
    "type": "Point",
    "coordinates": [75.78908449, 26.92782727]
  },
  "place": null,
  "contributors": null,
  "retweet_count": 0,
  "favorite_count": 0,
  "entities": {
    "hashtags": [],
    "symbols": [],
    "urls": [],
    "user_mentions": []
  },
  "favorited": false,
  "retweeted": false,
  "filter_level": "medium",
  "lang": "nl"
}
```

Exchange of information encoded as JSON involves encoding and decoding steps. The Python JSON package [109] provides functions for encoding and decoding JSON. Box 6.36 shows an example of JSON encoding and decoding.

■ Box 6.36: Encoding & Decoding JSON In Python

```
>>> import json

>>> message = {
    "created": "Wed Jun 31 2013",
    "id": "001",
    "text": "This is a test message."
}

>>> json.dumps(message)
'{"text": "This is a test message.", "id": "001",
"created": "Wed Jun 31 2013"}'

>>> decodedMsg = json.loads('{"text": "This is a
test message.", "id": "001", "created": "Wed Jun 31 2013"}')

>>> decodedMsg['created']
u'Wed Jun 31 2013'
>>> decodedMsg['text']
```

```
u'This is a test message.'
```

6.11.2 XML

XML (Extensible Markup Language) is a data format for structured document interchange. Box 6.37 shows an example of an XML file. In this section you will learn how to parse, read and write XML with Python. The Python *minidom* library provides a minimal implementation of the Document Object Model interface and has an API similar to that in other languages. Box 6.38 shows a Python program for parsing an XML file. Box 6.39 shows a Python program for creating an XML file.

■ Box 6.37: XML example

```
<?xml version="1.0"?>
<catalog>
< plant id='1' >
<common>Bloodroot</common>
<botanical>Sanguinaria canadensis</botanical>
<zone>4</zone>
<light>Mostly Shady</light>
<price> 2.44 </price>
<availability>031599</availability>
</plant>
<plant id='2' >
<common>Columbine</common>
<botanical>Aquilegia canadensis</botanical>
<zone>3</zone>
<light>Mostly Shady</light>
<price> 9.37</price >
<availability>030699</availability>
</plant>
<plant id='3' >
<common>Marsh Marigold</common>
<botanical>Caltha palustris</botanical>
<zone>4</zone>
<light>Mostly Sunny</light>
<price> 6.81</price>
<availability>051799</availability>
</plant>
</catalog>
```

■ Box 6.38: Parsing an XML file in Python

```
from xml.dom.minidom import parse
dom = parse("test.xml")
for node in dom.getElementsByTagName('plant'):
    id=node.getAttribute('id')
    print "Plant ID:", id
    common=node.getElementsByTagName('common')[0]
        .childNodes[0].nodeValue
    print "Common:", common
    botanical=node.getElementsByTagName('botanical')[0]
        .childNodes[0].nodeValue
    print "Botanical:", botanical
    zone=node.getElementsByTagName('zone')[0]
        .childNodes[0].nodeValue
    print "Zone:", zone
```

■ Box 6.39: Creating an XML file with Python

```
#Python example to create the following XML:
#<?xml version="1.0" ?> <Class> <Student>
#<Name>Alex</Name> <Major>ECE</Major> </Student > </Class>
from xml.dom.minidom import Document
doc = Document()

# create base element
base = doc.createElement('Class')
doc.appendChild(base)

# create an entry element
entry = doc.createElement('Student')
base.appendChild(entry)

# create an element and append to entry element
name = doc.createElement('Name')
nameContent = doc.createTextNode('Alex')
name.appendChild(nameContent)
entry.appendChild(name)

# create an element and append to entry element
```

```
major = doc.createElement('Major')
majorContent = doc.createTextNode('ECE')
major.appendChild(majorContent)
entry.appendChild(major)

fp = open('foo.xml','w')
doc.writexml()
fp.close()
```

6.11.3 HTTPLib & URLLib

HTTPLib2 and URLLib2 are Python libraries used in network/internet programming [111, 112]. HTTPLib2 is an HTTP client library and URLLib2 is a library for fetching URLs.

Box 6.40 shows an example of an HTTP GET request using the HTTPLib. The variable *resp* contains the response headers and *content* contains the content retrieved from the URL.

■ Box 6.40: HTTP GET request example using HTTPLib

```
>>> import httplib2
>>> h = httplib2.Http()
>>> resp, content = h.request("http://example.com", "GET")
>>> resp
{'status': '200', 'content-length': '1270', 'content-location':
 'http://example.com', 'x-cache': 'HIT', 'accept-ranges':
 'bytes', 'server': 'ECS
(cpm/F858)', 'last-modified': 'Thu,
25 Apr 2013 16:13:23 GMT', 'etag':
 '"780602-4f6-4db31b2978ec0"', 'date': 'Wed, 31 Jul 2013 12:36:05 GMT',
 'content-type': 'text/html; charset=UTF-8'}

>>> content
'<!doctype html>\n<html>\n<head>\n<title>Example Domain</title>\n\n
<meta charset="utf-8" />\n
:
```

Box 6.41 shows an HTTP request example using URLLib2. A *request* object is created by calling *urllib2.Request* with the URL to fetch as input parameter. Then *urllib2.urlopen* is called with the *request* object which returns the *response* object for the requested URL. The *response* object is read by calling *read* function.

■ Box 6.41: HTTP request example using URLLib2

```
>>> import urllib2
>>>
>>> req = urllib2.Request('http://example.com')
>>> response = urllib2.urlopen(req)
>>> response.read()
'<!doctype html>\n<html>\n<head>\n<title>Example Domain</title>\n\n<meta charset="utf-8" />\n
```

Box 6.42 shows an example of an HTTP POST request. The data in the POST body is encoded using the *urlencode* function from urllib.

■ Box 6.42: HTTP POST example using HTTPLib2

```
>>> import httplib2
>>> import urllib
>>> h = httplib2.Http()
>>> data = {'title': 'Cloud computing'}
>>> resp, content =
h.request("http://www.htmlcodetutorial.com/cgi-bin/mycgi.pl", "POST",
urllib.urlencode(data))
>>> resp
{'status': '200', 'transfer-encoding': 'chunked',
'server': 'Apache/2.0.64 (Unix) mod_ssl/2.0.64 OpenSSL/0.9.7a
mod_auth_passthrough/2.1 mod_bwlimited/1.4 FrontPage/5.0.2.2635
PHP/5.3.10', 'connection': 'close', 'date': 'Wed, 31 Jul 2013
12:41:20 GMT', 'content-type': 'text/html; charset=ISO-8859-1'} .
>>> content
'<HTML>\n<HEAD>\n<TITLE>Idocs Guide to
HTML: My CGI</TITLE>\n</HEAD>
:
```

Box 6.43 shows an example of sending data to a URL using URLLib2 (e.g. an HTML form submission). This example is similar to the HTTP POST example in Box 6.42 and uses URLLib2 request object instead of HTTPLib2.

■ Box 6.43: Example of sending data to a URL

```
>>> import urllib
>>> import urllib2
>>>
>>> url = 'http://www.htmlcodetutorial.com/cgi-bin/mycgi.pl'
>>> values = {'title' : 'Cloud Computing',
... 'language' : 'Python' }
>>>
>>> data = urllib.urlencode(values)
>>> req = urllib2.Request(url, data)
>>> response = urllib2.urlopen(req)
>>> the_page = response.read()
>>> the_page
'<HTML>\n<HEAD>\n<TITLE>Idocs Guide to
HTML: My CGI</TITLE>\n</HEAD>
:'
```

6.11.4 SMTPLib

Simple Mail Transfer Protocol (SMTP) is a protocol which handles sending email and routing e-mail between mail servers. The Python `smtplib` module provides an SMTP client session object that can be used to send email [113].

Box 6.44 shows a Python example of sending email from a Gmail account. The string `message` contains the email message to be sent. To send email from a Gmail account the Gmail SMTP server is specified in the `server` string.

To send an email, first a connection is established with the SMTP server by calling `smtplib.SMTP` with the SMTP server name and port. The user name and password provided are then used to login into the server. The email is then sent by calling `server.sendmail` function with the from address, to address list and message as input parameters.

■ Box 6.44: Python example of sending email

```
import smtplib

from_email = '<enter-gmail-address>'
recipients_list = ['<enter-sender-email>']
cc_list = []
subject = 'Hello'
message = 'This is a test message.'
```

```
username = '<enter-gmail-username>'  
password = '<enter-gmail-password>'  
server = 'smtp.gmail.com:587'  
  
def sendemail(from_addr, to_addr_list, cc_addr_list,  
subject, message,  
login, password,  
smtpserver):  
  
    header = 'From: %s\n' % from_addr  
    header += 'To: %s\n' % ','.join(to_addr_list)  
    header += 'Cc: %s\n' % ','.join(cc_addr_list)  
    header += 'Subject: %s\n\n' % subject  
    message = header + message  
  
    server = smtplib.SMTP(smtpserver)  
    server.starttls()  
    server.login(login, password)  
    problems = server.sendmail(from_addr, to_addr_list, message)  
    server.quit()  
  
#Send email  
sendemail(from_email, recipients_list, cc_list, subject,  
message, username, password, server)
```

Summary

In this chapter you learned the essentials of the Python programming language. Python is a general-purpose, high level programming language that supports more than one programming paradigms including object-oriented programming and structured programming. Python is an interpreted language and does not require an explicit compilation step. Python provides an interactive mode in which the user can submit commands at the Python prompt and interact with the interpreter directly. Python supports both procedure-oriented programming and object-oriented programming. Python programs can be directly executed from source code and copied from one machine to another without worrying about portability. Python Data Types & Data Structures include Numbers, Strings, Lists, Tuples and Dictionaries. Control flow statements in Python include *if*, *for*, *while*, *break*, *continue*, *range* and *pass*. A function in Python is a block of code that begins with the keyword *def* followed by the function name and parentheses. Python allows organizing the program code into different modules which improves the code readability and makes it easy to manage. Python packages allow better organization of modules related to a single application environment. Python provides all the standard features of Object Oriented Programming such as classes, class