MOBILE AND PERVASIVE COMPUTING

ASSESSMENT 3

MANGALASHRI R 2013103510

Chapter 1(Ubiquitous Computing: Basics and Vision)

Debate the benefits for Ubiquitous Systems: to support a strong notion of autonomy, to support a strong notion of intelligence.

The term 'ubiquitous', meaning appearing or existing everywhere, combined with computing to form the term Ubiquitous Computing (UbiCom) is used to describe ICT (Information and Communication Technology) systems that enable information and tasks to be made available everywhere, and to support intuitive human usage, appearing invisible to the user.

Autonomy

Autonomy refers to the property of a system that enables a system to control its own actions independently. An autonomous system may still be interfaced with other systems and environments. However, it controls its own actions.

- Autonomous systems are **self-governing** and are capable of their own independent decisions and actions.
- Autonomous systems may be **goal or policy oriented**: they operate primarily to adhere to a policy or to achieve a goal. There are several different types of autonomous system.
  - ➢ On the Internet, an autonomous system is a system which is governed by a router policy for one or more networks, controlled by a common network administrator on behalf of a single administrative entity.
  - ➢ A software agent system is often characterised as an autonomous system.

  Autonomous systems can be designed so that these goals can be assigned to them dynamically, perhaps by users.
- Rather than users needing to interact and control each low level task interaction, users only need to interact to **specify high level tasks or goals**. The system itself will then automatically plan the set of low level tasks needed and schedule them automatically, reducing the complexity for the user.
- The system can also **re-plan** in case a particular plan or schedule of tasks to achieve goals cannot be reached. Note the planning problem is often solved using artificial intelligence (AI).

- **Reducing Human Interaction**

  Much of the ubiquitous system interaction cannot be entirely human centred even if computers become less obtrusive to interact with, because:
  - ➢ Human interaction can quickly become a bottleneck to operate a complex system. Systems can be designed to rely on humans being in the control loop. The bottleneck can happen at each step, if the user is required to validate or understand that task step.
  - ➢ It may not be feasible to make some or much machine interaction intelligible to some humans in specific situations.
  - ➢ This may overload the cognitive and haptic (touch) capabilities of humans, in part because of the sheer number of decisions and amount of information that occur.
  - ➢ This original vision needs to be revisited and extended to cover networks of devices that can interact intelligently, for the benefit of people, but without human intervention. These types of systems are called automated systems.

- **Easing System Maintenance Versus Self-Maintaining Systems**

  Building, maintaining and interlinking individual systems to be larger, more open, more heterogeneous and complex systems is more challenging. Some systems can be relatively simply interlinked at the network layer. However, this does not mean that these can be so easily interlinked at the service layer, e.g., interlinking two independent heterogeneous data sources, defined using different data schemas, so that data from both can be aggregated. Such maintenance requires a lot of additional design in order to develop mapping and mediating data models. Complex system interaction, even for automated systems, reintroduces humans in order to manage and maintain the system.

Rather than design systems to focus on pure automation but which end up requiring manual intervention, systems need to be designed to operate more autonomously, to operate in a self-governed way to achieve operational goals. Autonomous systems are related to both context aware systems and intelligence as follows. System autonomy can improve when a system can determine the state of its environment, when it can **create and maintain an intelligent behavioural model of its environment and itself**, and when it can adapt its actions to this model and to the context. For example, a printer can estimate the expected time before the printer toner runs out based upon current usage patterns and notify someone to replace the toner.

Note that autonomous behaviour may not necessarily always act in ways that human users expect and understand, e.g., self-upgrading may make some services unresponsive while these management processes are occurring. Users may require further explanation and mediated support because of perceived differences between the system image (how the system actually works) and users' mental model of the system.

From a software engineering system perspective, autonomous systems are similar to **functionally independent systems in which systems are designed to be self-contained, single minded, functional, systems with high cohesion and that are relatively independent of other systems (low coupling)**. Such systems are easier to design to support composition, defined as atomic modules that can be combined into larger, more complex, composite modules.

Intelligence

It is possible for UbiCom systems to be context aware, to be autonomous and for systems to adapt their behaviour in dynamic environments in significant ways, without using any artificial intelligence in the system. Systems could simply use a directory service and simple event condition action rules to identify available resources and to select from them, e.g., to discover local resources such as the nearest printer. There are several ways to characterise intelligent systems. Intelligence can enable systems to **act more proactively and dynamically** in order to support the following behaviours in UbiCom systems:

- ***Modelling of its physical environment:*** an intelligent system (IS) can attune its behaviour to act more effectively by taking into account a model of how its environment changes when deciding how it should act.
- ***Modelling and mimicking its human environment:*** it is useful for a IS to have a model of a human in order to better support iHCI. IS could enable humans to be able to delegate high level goals to the system rather than interact with it through specifying the low level tasks needed to complete the goal.
- ***Handling incompleteness:*** Systems may also be incomplete because environments are open to change and because system components may fail. AI planning can support re planning to present alternative plans. Part of the system may only be partially observable. Incomplete knowledge of a system's environment can be supplemented by AI type reasoning about the model of its environment in order to deduce what it cannot see is happening.
- ***Handling non deterministic behaviour:*** UbiCom systems can operate in open, service dynamic environments. Actions and goals of users may not be completely determined. System design may need to assume that their environment is a semi deterministic environment (also referred to as a volatile system environment) and be designed to handle this. Intelligent systems use explicit models to handle uncertainty.
- ***Semantic and knowledge based behaviour:*** UbiCom systems are also likely to operate in open and heterogeneous environments. Types of intelligent systems define powerful models to support interoperability between heterogeneous systems and their components, e.g., semantic based interaction.

Chapter 2 (Applications: Past and Present)

Discuss how to design a UbiCom system to support Abowd and Mynatt's (2000) design principles for supporting daily informal activities.

Just as pushing the availability of computing away from the traditional desktop fundamentally changes the relationship between humans and computers, providing continuous interaction moves computing from a localized tool to a constant presence. Motivations for everyday computing stem from wanting to support the informal and unstructured activities typical of much of our everyday lives. These activities are continuous in time, a constant ebb and flow of action that has no clear starting or ending point. Familiar examples are orchestrating tasks, communicating with family and friends, and managing information.

Designing for everyday computing requires addressing these features of informal, daily activities:

- ***They rarely have a clear beginning or end:*** Either as a fundamental activity, such as communication, or as a long-term endeavour, such as research in human-computer interaction, these activities have no point of closure. Information from the past is often recycled. Although new names may appear in an address book or new items on a to-do list, the basic activities of communication or information management do not cease. A basic tenet in HCI is designing for closure. Given a goal, such as spell-checking a document, the steps necessary to accomplish that goal should be intuitively ordered with the load on short-term memory held to a reasonable limit. The dialogue is constrained so that the goal is accomplished before the user begins the next endeavour. When designing for an activity, principles such as providing visibility of the current state, freedom in dialogue, and overall simplicity in features play a prominent role.
- ***Interruption is expected:*** Thinking of these activities as continuous, albeit possibly operating in the background, is a useful conceptualization. One side-effect is that resumption of an activity does not start at a consistent point, but is related to the state prior to interruption. Interaction must be modelled as a sequence of steps that will, at some point, be resumed and built upon. In addition to representing past interaction, the interface can remind the user of actions left uncompleted.
- ***Multiple activities operate concurrently:*** Since these activities are continuous, the need for context-shifting among multiple activities is assumed. Application interfaces can allow the user to monitor a background activity, assisting the user in knowing when he or she should resume that activity. Resumption may be opportunistic, based on the availability of other people, or on the recent arrival of needed information. For example, users may want to resume an activity based on the number of related events that have transpired, such as reading messages in a newsgroup only after a reasonable number of messages have been previously posted. To design for background awareness, interfaces should support multiple levels of "intrusiveness" in conveying monitoring information that matches the relative urgency and importance of events.
- ***Time is an important discriminator:*** Time is a fundamental human measuring stick although it is rarely represented in computer interfaces. Whether the last conversation with a family member was last week or five minutes ago is relevant when interpreting an incoming call from that person. When searching for a paper on a desk, whether it was last seen yesterday or last month informs the search. There are numerous ways to incorporate time into human-computer interfaces. As we try to regain our working state, interfaces can represent past events contingent on the length of time (minutes, hours, days) since the last interaction. As applications interpret real-world events, such as deciding how to handle an incoming phone call or to react to the arrival at the local grocery store, they can utilize timing information to tailor their interaction.
- ***Associative models of information are needed:*** Hierarchical models of information are a good match for well-defined tasks, while models of information for activities are principally

associative, since information is often reused on multiple occasions, from multiple perspectives. For example, assume you have been saving email from colleagues, friends, and family for a long time. When dealing with current mail, you may attempt to organize it into a hierarchy of folders on various topics. Over time, this organization has likely changed, resulting in a morass of messages that can be searched with varying degrees of success. Likewise, interfaces for to-do lists are often failures given the difficulty in organizing items in well-defined lists. Associative and context-rich models of organization support activities by allowing the user to reacquire the information from numerous points of view. These views are inherent in the need to resume an activity in many ways, for many reasons. For example, users may want to retrieve information based on current context such as when someone enters their office or when they arrive at the grocery store. They may also remember information relative to other current information, e.g., a document last edited some weeks ago or the document that a colleague circulated about some similar topic.

- ***Design a continuously present computer interface:*** There are multiple models for how to portray computers that are ubiquitous, although none of these models are wholly satisfying. Computational systems that continue to operate in the background, perhaps learning from past activity and acting opportunistically, are typically represented as anthropomorphized agents. However it is doubtful that every interface should be based on dialogue with a talking head or human-oriented personality. Research in wearables explores continually worn interfaces, but these are limited by the current input and display technologies and are typically rudimentary text-based interfaces.
- ***Presenting information at different levels of the periphery of human attention:*** Despite increasing interest in tangible media and peripheral awareness, especially in computer-supported collaborate work (CSCW) and wearable computing, current interfaces typically present a generic peripheral backdrop with no mechanism for the user, or the background task, to move the peripheral information into the foreground of attention. Design should be aimed at creating peripheral interfaces that can operate at different levels of the user's periphery.
- ***Connecting events in the physical and virtual worlds:*** People operate in two disconnected spaces: the virtual space of email, documents, and Web pages and the physical space of face-to-face interactions, books, and paper files. Yet human activity is coordinated across these two spaces. There is much work left to be done to understand how to combine information from these spaces to better match how people conceptualize their own endeavours.
- ***Modifying traditional HCI methods to support designing for informal, peripheral, and opportunistic behaviour:*** There is no one methodology for understanding the role of computers in our everyday lives. However, combining information from methods as different as laboratory experiments and ethnographic observations is far from simple.

As computing becomes more ubiquitously available, it is imperative that the tools offered reflect their role in longer-term activities. Although principles in everyday computing can be applied to desktop interfaces, these design challenges are most relevant given a continuously changing user context. In mobile scenarios, users shift between activities while the computing resources available to them also vary for different environments. Even in an office setting, various tools and objects play multiple roles for different activities. For example, use of a computer-augmented whiteboard varies based on contextual information such as people present. Different physical objects such as a paper file or an ambient display can provide entry points and background information for activities. This distribution of interaction in the physical world is implicit in the notion of everyday computing and thus clearly relevant to research in ubiquitous computing.

Chapter 3 (Smart Devices and Services)

Argue for system designs that separate coordination or control from computation. Then discuss pros and cons of object-oriented versus event-driven versus blackboard repository type coordination.

Event-Driven Architectures (EDA)

Garlan and Shaw have identified and classified several different interaction mechanisms for distributed ICT systems. One common way to decouple coordination from computation is an event driven system which supports very loose coupled control or coordination between event generators or producers, and event receivers or consumers, e.g., clicking buttons on a User Interface (UI) produces events that trigger associated actions in services. This is also known as **publish and subscribe interaction**. One or more nodes publish events while others subscribe to being notified when specified events occur. An event is some input such as a message or procedure call that is of interest. An event may be significant because it may cause a significant change in state, e.g., a flat tyre triggers a vehicle driver to slow down. An event may cause some predefined threshold to be crossed, e.g., after travelling a certain number of miles, a vehicle must be serviced to maintain it in a roadworthy state. An event may be time based, e.g., at a certain time record a certain audio video program. External events can trigger services. Services may in turn trigger additional internal events, e.g., the wheel brake pads are too worn and need to be replaced. Event driven architectures are an important interaction to support service oriented architectures. These are referred to as event driven service oriented architectures.

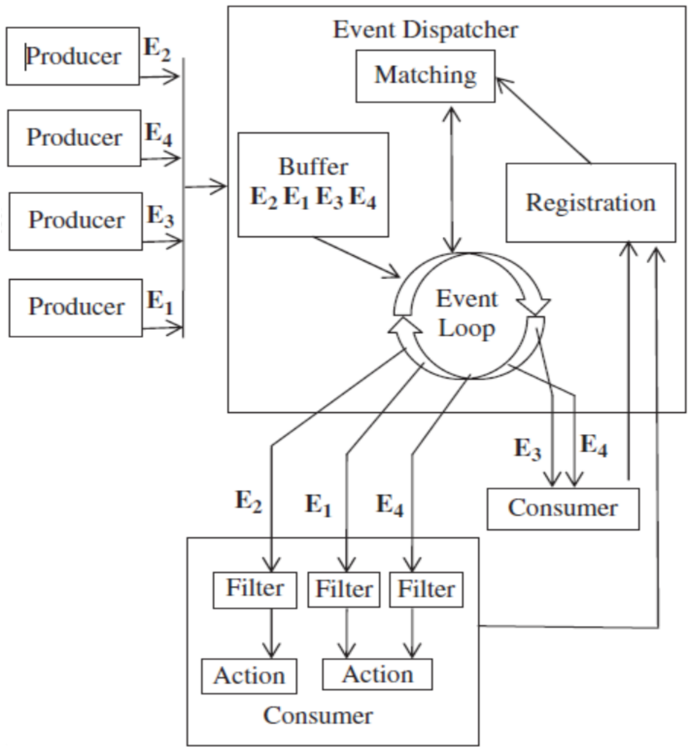

Figure 1 Event-Driven Interaction

Figure 1 shows a basic EDA. Multiple input events are input into an event buffer such as a first in first out (FIFO) or a first in last out (FILO) buffer. Events are matched with consumers that have previously registered an interest in those events. Matching can be simple, e.g., an event is matched with any registered consumers and sent to them, or more complex. In more complex design, events can be consumed using an ECA (Event Condition Action) paradigm, i.e., events are used to trigger actions when certain conditions are met, e.g., if time equals T1, start recording programme on channel

Y. These actions could in turn also trigger new events. Events could be filtered in the event producer rather than in a centralised event dispatcher that is shared between multiple event producers.

Shared Data Repository

In a repository style of interaction, two participants communicate by leaving messages for others via some shared intermediary. Hence, a shared repository system consists of two types of components: a central data structure represents the current state, and a collection of independent components operates on the central data store. There are two major sub types of coordination depending whether transactions in an input stream trigger the selection of executing processes, e.g., a database repository, or if the current state of the central data structure is the main trigger of selecting processes to execute, e.g., a blackboard repository. This represents and stores data that is created and used by other components. Repositories are similar to EDA systems in the way they support independence and allow volatile producers and consumers to be tolerated. The main difference between the EDA and shared data repository is the persistent storage of the input data and data management, e.g., consistency management. Examples of shared data repository include electronic bulletin boards, knowledge based blackboard systems including tuplespaces and relational data type databases.

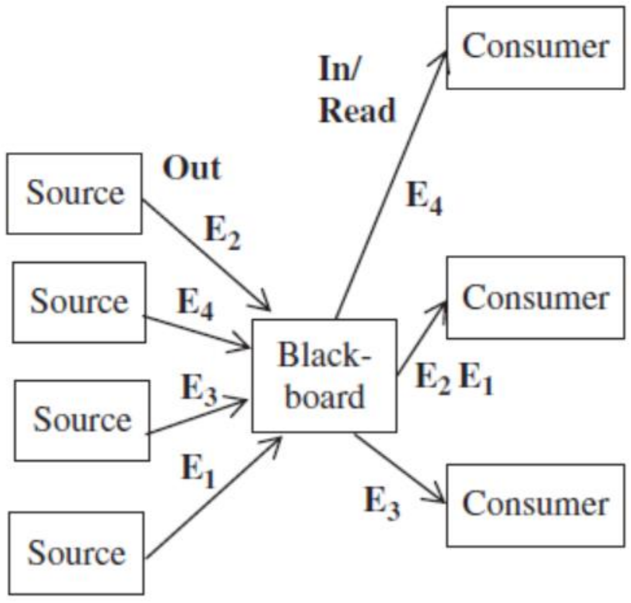

Figure 2 Shared Data Repository

Object-oriented Architecture

In Object-oriented design, the technology-independent concepts in the analysis model are mapped onto implementing classes, constraints are identified, and interfaces are designed, resulting in a model for the solution domain. In a nutshell, a detailed description is constructed specifying how the system is to be built on concrete technologies.
The stages for object–oriented design can be identified as:

- Definition of the context of the system
- Designing system architecture
- Identification of the objects in the system
- Construction of design models
- Specification of object interfaces

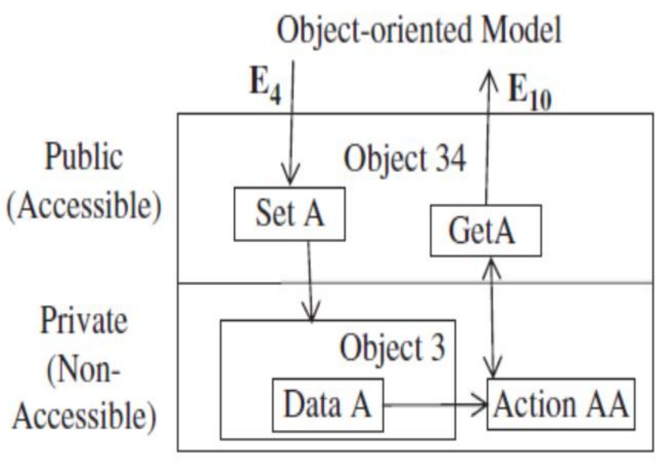

Figure 3 Object-oriented Architecture

Process-based designs

- Interfaces are patterns of actions whose transformational meaning is largely ignored.
- Richer composition mechanisms.
- Interaction by action handshake.

Service-based designs

- Interfaces as sets of ports through which data flows.
- Interaction is anonymous and handled by complex connectors.
- Clear separation between computation and coordination.

Object-oriented Coordination

Pros

- Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients.
- Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

Cons

- In order for one object to interact with another (via procedure call) it must know the identity of that other object. The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it. In a module oriented language this manifests itself as the need to change the "import" list of every module that uses the changed module.
- There can be side-effect problems: if A uses object B and C also uses B, then C's effects on B look like unexpected side effects to A, and vice versa.

Event-driven Coordination

Pros

- ***Portability:*** by providing computation language independent mechanisms for coordination.
- ***Heterogeneity:*** enabling devices and applications to coordinate with each other even when these mechanisms are implemented in different hardware or languages.
- ***Flexibility:*** enabling different coordination mechanisms and computations mechanisms to be mixed and matched.

- It provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system.
- Its invocation eases system evolution. Components may be replaced by other components without affecting the interfaces of other components in the system.

Cons

- Dealing with event floods and asynchronous coordination. In event floods, a few highly significant events can be lost in a great volume of non-significant events that cannot be processed in time. Event floods can be dealt with by prioritising events, enabling events to expire if not acted upon within a specific time frame and by using event filters. Event coordination may be needed by applications when events can arrive in any order.
- EDA generally have no persistence.
- There is no inherent way to reuse recent events.
- It is more difficult to keep things running in a failure situation; receivers may also need to track new senders to decide whether or not to subscribe to them.

Blackboard Repository

Pros

- They support independence and allow volatile producers and consumers to be tolerated.
- Persistent storage of the input data and data management, e.g., consistency management.
- Extending the data space is easy.

Cons

- Suitable only for problems with non-deterministic solution strategy known.
- All components have access to a shared data store.
- Changing the data space structure may be hard.

Chapter 4 (Smart Mobile Devices Networks and Cards)

What is OSGI? Discuss how OSGI can be used in a multi-vendor device discovery environment.

The **Open Services Gateway Initiative (OSGi**) defines and promotes open specifications, such as a core platform specification, for the delivery of managed services into networked environments such as homes and automobiles. The core OSGi platform specification defines the service framework to include a minimal component model, management services for components, and a service registry. OSGi in turn uses underlying Java VM and OS services.

Multi-vendor devices offer an increasing number of services and end-user applications that base their value on the ability to exploit the information originating from the surrounding environment by means of an increasing number of embedded sensors, e.g. GPS, compass, RFID reader and camera. However, usually such devices are not able to exchange information because of a lack of shared data storage and common information exchange methods. The integration of Smart-M3, an information interoperability platform with OSGi will provide a shared interoperable infrastructure, where multi-vendor devices can exchange information, enriched with the service discovery, usability facilities and development tools of OSGi.

Service interoperability with OSGi

In OSGi the application emerges from a set of dynamic modules called bundles that collaborate with each other. Dependencies between bundles and associated consistency are automatically managed. The OSGi service oriented component model enables networked services to dynamically discover other services and work together to achieve the desired functionality. It is also

possible to receive events notification when a service emerges or changes its properties. Services can be linked to devices. If a device is no more available, the services it offers are automatically unregistered. OSGi has a component-based architecture that enables device-side applications to load required components from the management server at run time. In this sense, application components can be customized on demand, e.g. if a new sensor is available or a new service appears, the specific module can be loaded and used smoothly.
The OSGi framework provides the following benefits:

- The modular approach based on bundles reduces the complexity, in terms of bundles development and in terms of system architecture.
- OSGi framework is simple.
- OSGi is a dynamic framework, where bundles can be updated on the fly and the associated services come and go dynamically.
- The framework is adaptive, because bundles can find out what capabilities are available on the system through a service registry and can adapt consequently the functionality they can provide.
- OSGi based solutions are easy to deploy, because the standard specifies how components are installed and managed.
- It provides a new security model that leverages and hardens the Java fine grained security model but improves the usability by introducing a simple way to specify the security details of the bundles.
- It is not intrusive because it can run potentially in any existing facility and can be easily ported to almost any software environment.
- It is available since 1998 and has been extensively used in several application contexts (automotive, mobile and fixed telephony, industrial automation, gateways & routers, private branch exchanges, etc.).
- It is supported in many development environments (IBM Websphere, SpringSource Application Server, Oracle Weblogic, Sun's GlassFish, Eclipse, and Redhat's JBoss) and by key companies (Oracle, IBM, Samsung, Nokia, IONA, Motorola, NTT, Siemens, Hitachi, Ericsson, etc.).

OSGi specifications do not concern with interoperability at the information level therefore the introduction of elements that allow information interoperability would represent an important improvement for OSGi platform.

Information interoperability with Smart-M3

Smart-M3 is an open source architecture supporting a tuple-space and agent-based model of computation upon a semantic web substrate providing integration of different kinds of devices at the information level and thereby facilitating interoperability between applications and devices. It provides cross domain search extent of information. As an example, Smart-M3 allows an application developer who works on a specific mobile platform to access simultaneously and in uniform way contextual information of a car, home, office, football stadium, etc. Smart-M3 stores information in RDF (Resource Description Language) format, which is a W3C standard. First, RDF provides the ability to join data from vocabularies belonging to different business domains, without having to negotiate structural differences between the vocabularies.
The main concepts on which Smart M3 is based are the following:

- Smart Space (SS): a named search extent of information.
- Semantic Information Broker (SIB): an entity for storing, sharing and governing the information of one smart space as RDF triples, i.e. ontology.
- Knowledge Processor (KP): any entity contributing to produce (insert, remove, update) and/or consume (query, subscribe) information in a SS. Common functionalities to access and use SS information are made available to KP by Knowledge Processor Interfaces (KPls), i.e. KPls hide SSAP details.

- Smart Space Application (SSA): a subset of KPs that use one (or more) SS as resource to perform the desired functionality.

Smart-M3 architecture allows KPs running in different business domains to share interoperable information.

Integrated service and information interoperability solution

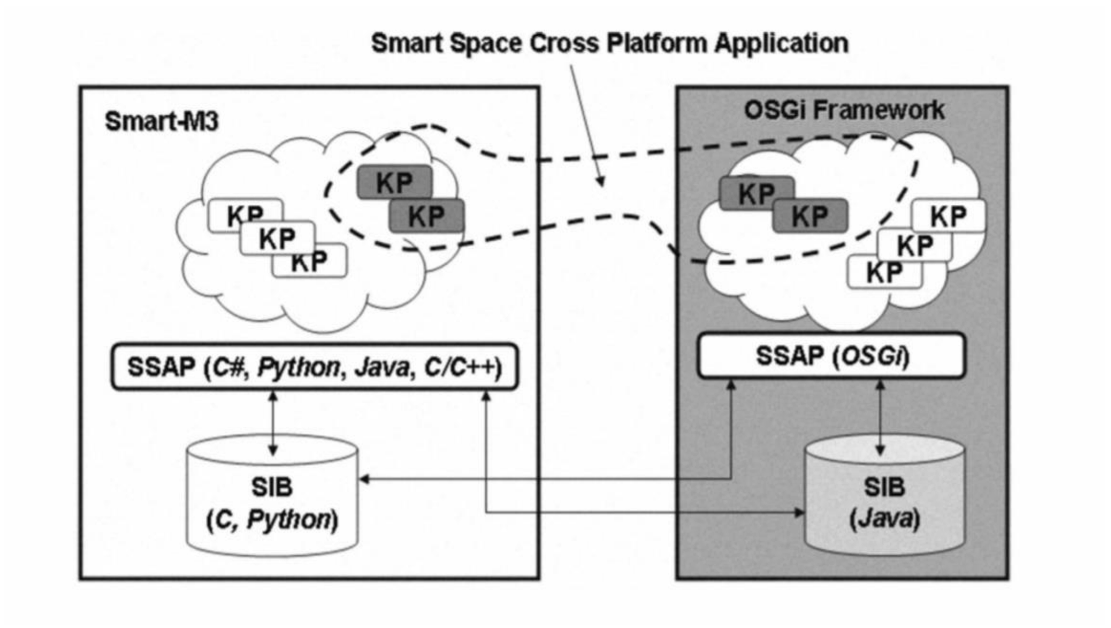

Figure 4 Integrated Smart-M3/OSGi Solution

The functionalities of the information interoperability level are based on the interaction between the actors involved in a smart environment, i.e. KPs and SIBs, and on the common access protocol, the SSAP. Porting these elements into OSGi, as OSGi bundles, introduces the information interoperability level into OSGi framework allowing OSGi Smart Environments applications to exchange information and use services in a unified and simple way.

In Figure 4, on the left side - the Smart-M3 KPs written in Python, Java, C#, C/C++ and - on the right side - the same modules implemented in Java as bundles running on the OSGi framework. Through the SSAP these modules can communicate and exchange information in both directions(Smart-M3 to OSGi, OSGi to Smart-M3).

The OSGi Alliance creates specifications and test suites that allow interoperability between different implementations.

For example, future generations of smart phones will contain an OSGi Framework. Complex applications or libraries written for a Nokia phone will then run unmodified on a Motorola phone.

Multi-vendor interoperability gives the platform operator as well as the manufacturer a choice. The operators can buy components from different vendors and run them on the same service platforms. The resulting competition increases quality and lowers prices.