Pers Ubiquit Comput (2014) 18:1099–1116
DOI 10.1007/s00779-013-0722-7

ORIGINAL ARTICLE

A pluggable middleware architecture for developing context-aware mobile applications

Nearchos Paspallis · George A. Papadopoulos

Received: 30 November 2012 / Accepted: 3 July 2013 / Published online: 8 October 2013

Abstract The proliferation of powerful smartphone devices provides a great opportunity for context-aware mobile applications becoming mainstream. However, we argue that conventional software development techniques suffer because of the added complexity required for collecting and managing context information. This paper presents a component-based middleware architecture which facilitates the development and deployment of context-aware applications via reusable components. The main contribution of this approach is the combination of a development methodology with the middleware architecture, which together bring significant value to developers of context-aware applications. Further contributions include the following: The methodology utilizes separation of concerns, thus decreasing the developmental cost and improving the productivity. The design and implementation of context-aware applications are also eased via the use of reusable components, called context plug-ins. Finally, the middleware architecture facilitates the deployment and management of the context plug-ins in a resource-aware manner. The proposed methodology and middleware architecture are evaluated both quantitatively and qualitatively.

Keywords Context-awareness · Middleware · Reusability · Pluggability · Modularity · Mobile devices · Separation-of-concerns

N. Paspallis (✉)
UCLan Cyprus, 7080 Pyla, Cyprus
e-mail: npaspallis@uclan.ac.uk

G. A. Papadopoulos
University of Cyprus, 2109 Aglantzia, Cyprus

1 Introduction

With a global population penetration exceeding 67 % as of 2010, the mobile phone is considered the most prolific technology to date [1]. An increasing subset of them are smartphones—powerful mobile devices, capable of running third-party applications—which are themselves credited as one of the greatest technologies of the last decade [28]. The advanced capabilities of these devices have also increased user expectations; Improvement is especially sought in areas such as context-awareness and self-adaptation. However, enabling this kind of sophisticated functionality on devices that are typically characterized by limited resources and capabilities is a challenging engineering task.

This paper presents a pluggable and modular middleware architecture which enables individual context providers (such as a location sensor) and context consumers (such as an interactive tourist guide) to be independently developed and dynamically bound via a middleware layer. The architecture allows attaching new peripherals to the device at runtime (e.g., a Bluetooth-based GPS receiver) without requiring that the dependent applications are restarted to take advantage of the additional—or richer—context information. Furthermore, by separating the roles of context providers and context consumers and also by adopting component orientation [32], we facilitate code reuse in the form of reusable context plug-ins. Additionally, the proposed middleware architecture features modularity, which allows the developers to quickly and easily configure the middleware according to the needs of the deployed applications and the capabilities of the deployment platform. Finally, the middleware behaves in a resource-aware manner by dynamically activating and deactivating context plug-ins on demand, and thus minimizing the run-time resource consumption (e.g., battery drain and memory use).

The rest of this paper is organized as follows. Section 2 introduces our view on context-awareness and lists the requirements for the proposed middleware architecture. The pluggable and modular aspects of the architecture are described in detail in Sect. 3, along with some implementation issues. The architecture is further illustrated via an extensive case study application, described in Sect. 4, followed by a qualitative and quantitative evaluation in Sect. 5. The paper closes with conclusions and pointers to future work in Sect. 6.

2 Requirements

The term of context-awareness was first studied [33] and defined [31] just a few years after Weiser originally documented his vision of *ubiquitous computing* [35]. At that point, context was defined as "the location of use, nearby people, hosts and accessible devices as well as changes to these things over time." At that point, context-awareness was often viewed as merely location-awareness, and the first context-aware applications were predominantly location-aware (e.g., the Active Badge [33], the ParcTab [34], and the CyberGuide [19] projects). As more researchers got engaged in the study of context-aware applications though, its definition evolved [3]. One of the most commonly referenced works defines context as "any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves" [6].

In relation to ubiquitous computing, context-awareness is considered to be one of the main enabling technologies. For instance, ubiquitous computing envisions a future where the interaction with the users is minimized and raised to a subconscious level, whenever possible. However, in order for computing systems to achieve this degree of invisibility, they need to keep track of as much of the contextual information as possible: available resources, possible means of *User Interfacing*, the user preferences and profile, as well as past user choices, user state, and the physical environment.

This section presents a list of requirements, as identified in the literature, which is relevant to the decisions that led to the design of the middleware architecture described in the following sections.

2.1 Functional requirements

The functional requirements are discussed first, as they are more focused on the properties, which are specific to context-awareness.

Application-specific context acquisition, analysis and detection This requirement is about providing a uniform and platform-independent interface for applications to express their *need* for different context data without knowing how that data are acquired [30, 36]. A consistent and well-defined context access interface is an important requirement for easing the development via separation-of-concerns.

Context histories As context information is dynamically generated and processed, new values continuously supersede previous ones. For example, when a new reading of a GPS sensor is performed, the newly reported location is used to replace the previous one. However, in many cases, access to historical context values is needed as well. Dey et al. [7] argue that maintaining a context history is an important requirement in order to enable arbitrary context-aware applications.

Support for multiple concurrent context-aware applications As modern mobile devices have become more powerful, they are commonly used to run multiple applications in parallel. Providing support for concurrent context-aware applications is thus a natural requirement. Instead of employing multiple instances of the same context sensors (e.g., sensors, widgets, components etc), it is preferable that applications share the context information as it is generated by a single source [10, 21].

Transparent distribution of context data In many cases, the context information gathered and modeled in one device is relevant to a wider scope. Furthermore, the distribution of context information should be as transparent as possible in order to ease the development of distributed context-aware applications [7]. There are many reasons why this is important: From a functional perspective, in some cases, the application logic itself requires context distribution. For example, a communication application might reveal the callee's status (e.g., available, busy, or unavailable), before a call is even attempted. From an extra-functional perspective, context distribution is also important as it can enable richer context information (from multiple, distributed sensors) and better resource utilization (sharing of common sensors installed only in a subset of the devices requiring the corresponding context information). For instance, a higher-precision GPS receiver embedded in a car is preferred to a smartphone's WiFi-based location sensor when the user is driving.

Privacy of context information Privacy is one of the most often cited criticisms of ubiquitous computing and is considered one of the main barriers to its long-term success [15]. As context frameworks have access to sensitive user information (e.g., static information, like a user's profile, and dynamic information, like a user's location), they should strive to protect context data. This requirement is particularly important in cases of context distribution and

context storage. It should be noted that while privacy is a common concern in most computing systems, it requires special attention in mobile and pervasive environments as they introduce challenges not present in traditional desktop computing.

Traceability and control of context-aware behavior As context-aware systems typically employ self-adaptation logic, which can be quite complex, there should be some provision for allowing the developers (and the users) to inspect the information flow—and whenever possible also manipulate it—in order to provide users with adequate understanding and control of the system and to facilitate debugging [13]. Traceability significantly improves user acceptance, as it was found that users are more tolerant of—even imperfect—context-aware applications when they understand that their autonomic behavior is rational [12, 25].

2.2 Extra-functional requirements

The extra-functional requirements are not specific to context-aware systems, but nevertheless, they discuss important aspects that must be taken into consideration.

Code reuse By implementing the context-aware related functionality (i.e., context sensing, processing, storing, etc) outside the application, multiple applications can reuse the same code, either one at a time or simultaneously. The requirement for code reuse in the form of reusable software components was identified as early as in [7].

Modularity As different applications and different deployment platforms have different requirements and capabilities respectively, a modular architecture is needed to ensure that those and only those features are used, which are explicitly needed in each case. A modular architecture can enable this by allowing different variants of the system to be deployed on varying platforms as needed. They also provide the advantage of more resource-efficient customization according to the dynamic needs of the deployed applications.

Separation-of-concerns Separation-of-concerns is a requirement that enables a more disciplined and efficient method for developing context-aware applications [7, 10]. For instance in [22] and in [23], the concerns of context-awareness and self-adaptiveness are treated independently of the standard business logic of the application. As the developers are allowed to focus on individual concerns of a developed context-aware system at a time, the overall development and maintenance tasks are rendered easier, faster, and more cost efficient.

Dynamic behavior Many context-aware frameworks use specialized context sensing components, which are used as wrappers around hardware sensors or OS libraries (e.g., context widgets [5] and context plug-ins [24]). Arguably, when these are deployed on mobile devices which are characterized by limited battery resources, it should be possible to dynamically activate or deactivate them as needed. This should be done according to the actual runtime needs of the deployed applications, while at the same time minimizing resource consumption.

Resource efficiency The requirement for resource efficiency refers to the ability of context-aware frameworks to operate on lightweight devices while imposing only a small footprint on resources such as CPU, memory, and battery. This requirement is particularly important for small, mobile devices which are characterized by limited resources because of space, shape, and weight constraints.

A additional number of extra-functional requirements were also identified in the literature, such as *ease of building*, *platform independence*, *lightweight architecture*, *support for mobility*, *ease of deployment and configuration*, *uniform development support*, *evolution*, *scalability*, *adoption of existing patterns and standards* and *fault tolerance*. While these requirements are not explicitly discussed in this paper, they were taken into consideration during the design and the evaluation of the development methodology and the middleware architecture [20].

3 Middleware architecture

This section presents the middleware architecture which enables the deployment and management of context-aware applications. These applications consist of context providers and consumers that are independently developed and dynamically composed—via a middleware layer—to form context-aware applications. The proposed architecture allows for dynamically adding and/or removing context providers (such as a Bluetooth-based GPS sensor) at runtime, without requiring that the dependent applications (such as an interactive walking tour application) are restarted in order to take advantage of the additional—or richer—context information. Furthermore, by separating the roles of context providers and consumers, and also by building on top of a component-oriented architecture [32], code reuse is facilitated in the form of reusable context plug-ins. Also, by monitoring the plug-ins' metadata—which encode their *provided* and *required* context types—the middleware intelligently and autonomously activates and deactivates context plug-ins as needed, and thus optimizing their resource consumption (e.g., battery drain, memory use, etc.). Finally, the architecture of the middleware itself is modular, allowing for alternative variants of it which can better match the requirements and properties of the target domain.

The single most important functional requirement of the middleware architecture is quite straightforward,

providing application-specific access to context information. To this end, the architecture was designed to provide support for *collecting*, *storing*, *organizing*, and *accessing* context information. Additional requirements, such as interoperability, inference of inter-dependencies of context, and support for context distribution, are also important. These functional requirements have led to the implementation of a comprehensive context model and an elaborate *Context Query Language* (CQL), which, however, are only briefly described here (more details are available in [9, 26, 27]).

3.1 Pluggability

In order to accommodate the requirements detected in Sect. 2, a basic guideline was adopted: The developers must be able to "independently develop and deploy context providers and context consumers." In other words, the concern of developing context-aware applications is separated into the concerns of developing *context producing* and *context consuming* components. This has led to the adoption of a *pluggable* architecture, where the context providers are designed and developed as *dynamically pluggable components* which are capable of independent deployment and activation. These components are referred to as *context plug-ins*. On the other side, the context-aware applications act as context clients and are only loosely coupled with the context providers.

To support such separation-of-concerns, a middleware-based system is defined. This system acts as an intelligent context hub: It collects, stores, processes context data, and makes it available to context clients. It should be noted that in this system's architecture, the context providers and the context consumers are *loosely coupled*, interacting with each other only through the middleware based on their context dependencies. This enables developers to build their own context plug-ins or reuse existing ones, as per the proposed developmental methodology.

While some context plug-ins can be pure context providers (i.e., context sensor plug-ins), others may also require input of—typically—primitive context data and thus be considered as both context providers and consumers (i.e., context reasoner plug-ins).

The context types provided and—optionally—required by each context plug-in are explicitly defined at design-time and read by the middleware during installation. This information is used to resolve the context dependencies of the plug-ins. Furthermore, based on the context types that the deployed context-aware applications need, the middleware dynamically activates an appropriate subset of context plug-ins. The corresponding mechanisms are described in Sects. 3.1.2 and 3.1.3.

3.1.1 Context plug-ins

In our model, context plug-ins are software components which are defined and deployed independently (as per Szyperski's definition [32]). These components provide a contractually specified interface, which allows them to interact with a central entity in the middleware (i.e., the *context manager*).

While the actual context sensing and context reasoning are a core responsibility of the plug-in developer, synthesis of plug-ins and lifecycle control is automatically handled by the middleware. To enable these, the plug-ins need to

- Realize an interface that allows them to communicate context data to and from the middleware, and
- Specify the provided (mandatory) and required (optional) context types as static metadata.

Concerning the former, the interface is heavily dependent on the actual implementation and the underlying platform. In the two Java-based implementations discussed in this paper, i.e., OSGi (*Open Service Gateway initiative*) and Android (see Sect. 3.3), this is realized via a standard Java interface (i.e., the `IContextPlugin`).

Concerning the latter, the semantics (i.e., syntax of the metadata) of the context dependencies are standardized as per the adopted context model [26, 27], but the place of annotation is again dependent on the underlying platform (e.g., the metadata is stored in the file `MANIFEST.MF` in the case of OSGi, and in the file `AndroidManifest.xml` in the case of Android).

The context plug-in lifecycle is defined as a simple automaton with three states:

- *Installed* where all static dependencies (e.g., software libraries and/or hardware components) of a plug-in are resolved,
- *Resolved* where a plugin is Installed *and* all its context dependencies are resolved (note that, by definition, context sensors have no context dependencies, and thus they are always resolved),
- *Active* where a plug-in is Resolved and active and generating context data (note that a plug-in might be Resolved, but not activated because its provided context data are not needed at the time).

The context plug-ins define their provided context types in addition to their required ones. In this respect, context sensors are automatically considered resolved, while context reasoners are considered resolved *only when their context dependencies are satisfied*. For instance, a context reasoner plug-in, which produces context type A and requires context type B, cannot be resolved unless there is at least one plug-in installed—and *resolved*—offering

context type B. Similarly, a plug-in offering a context type C is activated only when there is at least one active context consumer in need of that context type. As changes to the required and provided context types are dynamic, transitions back from `Active` to `Resolved` and possibly to `Installed` are also feasible. When a plug-in's state transitions to `Active`, then a special `activate` method is automatically invoked by the middleware. Similarly, when its state transitions out of the `Active` state, the `deactivate` method is invoked (i.e., *Inversion-of-Control* pattern).

The rationale for these context plug-in-specific states is the following: When a context plug-in is installed and resolved (in OSGi terms) or simply installed (in Android terms), it should only be activated when that would be useful. But, if the plug-in is not resolved (in terms of context dependencies), then activating it would not yield any context data as the required input would never be provided (or worse, it could yield erroneous data). Similarly, when no context client requires the context types provided by a specific plug-in, then activating it would not have any impact to the system, except for resource consumption. Thus, by introducing these specialized states, it is possible to optimize the resources consumed by the context plug-ins by activating those, and only those plug-ins, which are absolutely necessary.

In practice, it is assumed that the developers implement their plug-ins so that when they are activated, they allocate their required resources, and when they are deactivated, they de-allocate them. For example, if the context plug-in is used to generate "location" context types via input from a GPS device, then activating the plug-in would result in turning the GPS sensor on and deactivating it would result in turning it back off. Consequently, by only activating the location plug-in when an actual context-aware application needs location context information, it is possible to conserve significant resources (in this case, battery).

After a plug-in is installed (and assuming its context dependencies are resolved and is activated), it is registered by the context middleware which tracks its *provided* and *required* context types. Additionally, when context clients connect to the context middleware and inquire context data, the latter also records data about these clients and their required context types.

At run-time, the required and provided context types are processed by the *context manager*, which uses this information to resolve and activate the plug-ins as needed. In particular, the resolution mechanism is triggered when the provided context changes, i.e., every time a new plug-in is installed, or an existing one is uninstalled. The activation mechanism, on the other hand, is triggered when the required context changes, i.e., when a new context client is activated or an existing one deactivated. The result of this process is to activate those—and only those—context plug-ins that are resolved and offer context types needed by active context clients.

A two-phase approach is followed: The plug-ins are first resolved and then activated as needed. The corresponding algorithms used in these mechanisms are presented in the following two subsections.

3.1.2 Resolution mechanism

The resolution mechanism is responsible for marking plug-ins as *resolved* or *unresolved*. Resolved are those plug-ins which either have no context dependencies, or their dependencies are offered by some other resolved plug-ins. It should be noted that when a plug-in is marked as resolved, it is implied that it can be instantaneously activated and start producing events of its provided context type. As it is evident from this definition, an iterative algorithm is a straightforward approach for realizing the resolution mechanism. This mechanism is depicted in Algorithm 1.

The resolution mechanism defines three data structures: First, the `all plug-ins` are a set, which includes all the installed plug-ins. As it was mentioned earlier, the execution of this algorithm is triggered by changes to this set. The second data structure, `resolved`, is a subset of the `all plug-ins` set and contains the plug-ins which are marked as resolved. Finally, the last data structure, `provided`, is a map which associates each context type to a list of context plug-ins. Each of these lists includes those, and only those, plug-ins that are known to provide the corresponding context type and are known to be resolved.

Regarding the algorithm, it consists of two phases: In the first phase, it makes sure that each resolved context plug-in is marked as accordingly (i.e., it is included in the [resolved] set). To achieve this, it repeatedly checks each unresolved plug-in to see if there is a change in its dependencies. If it is found to be resolved, it is marked as such. Otherwise, when a complete iteration of the unresolved plug-ins is completed without a change, it is assumed that all resolved plug-ins have already been marked. This part of the algorithm always terminates, because there is only a limited number of unresolved plug-ins, and at least one of them is shifted to the

resolved set in each iteration—with the exception of the last one.

Algorithm 1 Resolution mechanism algorithm

Basic data-structures
[all plug-ins] - set containing all installed context plug-ins
[resolved] - subset of the above; contains only resolved plug-ins
[provided] - map of context types C_T to sets of resolved plug-ins providing C_T
Triggered by
Changes to the [all plug-ins] set
Algorithm

```
1: {first ensure that all resolved plug-ins are in [resolved]}
2: changes-detected ⇐ true
3: while changes-detected do
4:    changes-detected ⇐ false
5:    for all p in [all plug-ins] - [resolved] do
6:       if reqCtxtTypes(p) ⊆ [provided].keys then
7:          [resolved] ⇐ [resolved] ∪ {p}
8:          [provided] ⇐ [provided] ∪ {prvCtxtTyps(p) → p}
9:          changes-detected ⇐ true
10:      end if
11:   end for
12: end while
13: {next ensure that no unresolved plug-ins are in [resolved]}
14: changes-detected ⇐ true
15: while changes-detected do
16:   changes-detected ⇐ false
17:   for all p in [resolved] do
18:      if not reqCtxtTypes(p) ⊆ [provided].keys then
19:         [resolved] ⇐ [resolved] - {p}
20:         [provided] ⇐ [provided] - {prvCtxtTypes(p) → p}
21:         changes-detected ⇐ true
22:      end if
23:   end for
24: end while
```

The second phase achieves a symmetric goal: It ensures that all plug-ins, included in the resolved set, are still resolved. To achieve this goal, an iterative process is used again, where each plug-in that is marked as resolved is checked against its dependencies. If it is found to be unresolved, it is unmarked (i.e., it is deleted from the resolved set), and the process repeats until a full iteration is completed without changes. This part of the algorithm also always terminates, because there is only a limited number of plug-ins in the resolved set, and at least one of them is deleted from that in each loop.

3.1.3 Activation mechanism

While the resolution mechanism is triggered by changes to the availability of context plug-ins, the activation mechanism is triggered by changes to the context needs. As context-aware applications start and stop, they register and unregister for context notification with the middleware. For this purpose, a mapping for each needed context type is maintained, pointing to a list of registered clients. Context clients, which require periodic, synchronous access to context, are required to register their context needs with the middleware. Asynchronous subscribers to context data are also registered, implicitly while placing their queries. This approach allows the middleware to be aware of the actual context needs of all the applications and thus be able to intelligently activate and deactivate the plug-ins on demand. The details of this mechanism are presented in Algorithm 2.

This algorithm leverages four data structures: First, provided is a map of all provided context types to a list of resolved plug-ins providing them (i.e., this data structure is identical to the corresponding data structure used in the resolution mechanism). The second data structure (needed) is a similar one, mapping—however—needed context types to the corresponding requestors. In this case, the keys are context types, and the values are lists of objects, corresponding to the context clients. Next, the resolved is a set containing all plug-ins that are marked as resolved, and finally, active is a set containing all plug-ins that are marked as being active. Naturally, the active set is a subset of resolved as that was defined in the resolution mechanism. Updating these data structures is the main goal of this mechanism.

After an initialization phase where all unresolved plug-ins are removed from the set of active plug-ins, the activation algorithm spans two phases, just like the resolution one. In the first phase, all the plug-ins to be activated are selected by iterating the needed context types and *selecting* from the ones providing the corresponding context type. After the appropriate plug-ins are selected for the given context type, the "needed" data structure is also updated with any new context types possibly needed by the newly activated plug-ins. This loop is repeated until no changes are detected.

It should be noted that the *select* operation defined in line 8 of Algorithm 2 can be as basic as selecting *all* the plug-ins providing the corresponding context type C_T, or, it can be quite complex featuring intelligent selection of a *subset* of the plug-ins based on their resource consumption and their *Quality of Context* (QoC) properties. In [24], we implemented, tested, and evaluated the simple selection method, showing how the resource consumption is improved. Providing more elaborate implementations for the selection operator is a promising future direction, with a potential for further resource utilization, that is currently being investigated.

Algorithm 2 Activation mechanism algorithm

Basic data-structures

[provided] - map of context types C_T to sets of resolved plug-ins providing C_T

[needed] - map of all needed context types to lists of corresponding clients

[resolved] - contains only those plug-ins that are resolved

[active] - the set of all plug-ins that need to be active

Triggered by

Changes to the [needed].keys context types set

Algorithm

```
1: {first ensure that unresolved plug-ins are not in [active]}
2: [active] ⇐ [active] ∩ [resolved]
3: {next make sure that the needed plug-ins are in [active]}
4: changes-detected ⇐ true
5: while changes-detected do
6:    changes-detected ⇐ false
7:    for all C_T in [needed].keys do
8:       for all p in select(provided [C_T]) do
9:          [active] ⇐ [active] ∪ {p}
10:         [needed] ⇐ [needed] ∪ {reqCtxtTypes(p) → p}
11:         changes-detected ⇐ true
12:      end for
13:   end for
14: end while
15: {last make sure that no unneeded plug-ins are in [active]}
16: changes-detected ⇐ true
17: while changes-detected do
18:   changes-detected ⇐ false
19:   for all p in [active] do
20:      if provCtxtTypes(p) ∩ [needed].keys == ∅ then
21:         [active] ⇐ [active] - {p}
22:         [needed] ⇐ [needed] - {reqCtxtTypes(p) → p}
23:         changes-detected ⇐ true
24:      end if
25:   end for
26: end while
```

In the second phase of the algorithm, each plug-in that is marked as active is checked against the needed context types. If none of its provided context types is needed by the active context clients, then it is unmarked, and any context requirements it possibly has are removed from the `needed` map. This process is repeated until no new changes are detected.

At the end of this algorithm, the states of the plug-ins are checked, and their lifecycle is adjusted accordingly. For instance, any active plug-ins that were unmarked are deactivated and any inactive ones that were marked are activated. From a technical point-of-view, the changes in the plug-ins' lifecycle are enabled via corresponding methods defined in the plug-ins, realizing the *Inversion-of-Control* paradigm.

3.2 Modularity

The context middleware is a component-based framework itself, consisting of a main, central component—the *Context Manager*—and a few secondary ones, including the *Context Model*, the *Context Repository*, and the *Context Query Processor*. The internal structure of the context middleware architecture is illustrated in Fig. 1.

3.2.1 Main services

As the main component in the architecture, the context manager enables the core functionality of the context middleware: It *receives*, *stores*, *processes*, and, eventually,

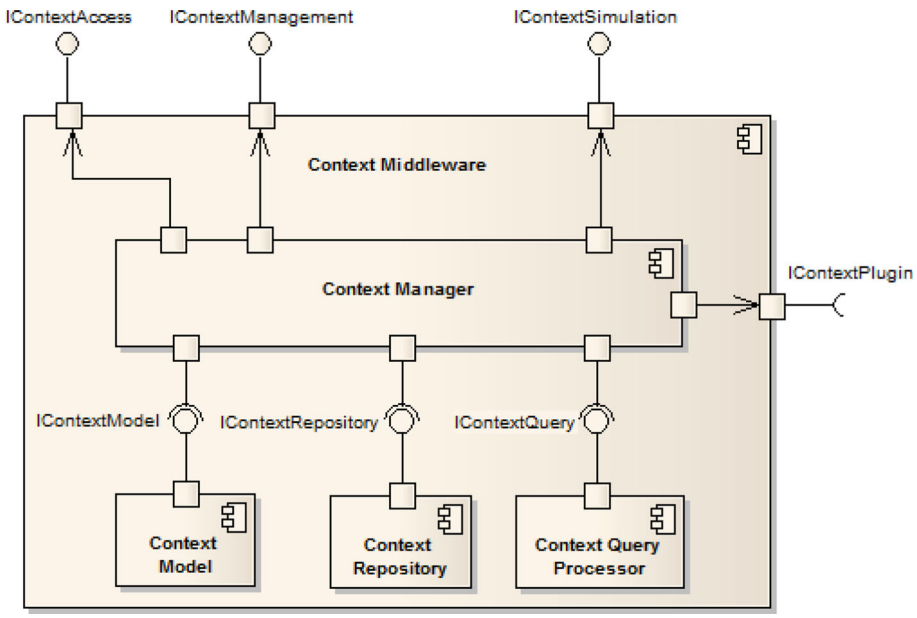

Fig. 1 The modular architecture of the context middleware

forwards context information from context providers to context consumers. It externalizes—via delegation—three main services:

The `IContextAccess` is the basic service enabling access to context information. The queries can be either synchronous or asynchronous. In the first case, simple context types—or conditional context queries—are used to specify the desired context data. Similarly, in the latter case, either a context type is specified so that the inquirer is notified whenever a change is sensed, or a more complex CQL-based query [9] is used to specify the conditions under which the inquirer should be notified.

The `IContextManagement` service aims at facilitating the extension of the context middleware. For this reason, it provides the functionality for monitoring the *required* context types, which vary as a result of the inquiries placed by the deployed applications, as well as the *provided* context types, which are reported by the installed context plug-ins. This is central, for instance, for adding support for context distribution.

The `IContextSimulation` is a utility service, used for enabling testing and debugging. In particular, this service allows external components to get access to the flow of all context information, intercept it, and even simulate arbitrary context events (with assistance from the context management service). This was used, for instance, for building a scenario engine for testing context-aware applications [18].

Finally, the `IContextPlugin` is the only externally needed service, which is used to enable the dynamic binding of context plug-ins to the middleware (see Sect. 3.2.3).

On the other hand, the context manager also features one optional and two mandatory *internal* services:

The `IContextQuery` service defines support for processing queries expressed in the CQL. However, simple context-aware applications are often realized using the plain context access API only, which provide access to requested context types filtered only by timestamp. For this reason, this service is defined as optional, and thus allowing for the formation of lighter configurations when needed (e.g., for resource-constrained devices).

The `IContextModel` service is mandatory and provides the required functionality for enabling transformation operations, as well as for identifying relationships between context entities, both at run-time. While the default implementation is realized using ontologies (offering a powerful, yet resource-demanding context model), simpler hardcoded implementations are also available (offering less flexibility and adaptability but with a much lighter resource footprint).

The `IContextRepository` service is also mandatory, and it provides the functionality needed for storing historical context values. The context data are initially provided (and accessed) in the form of objects, which are then serialized so they can be stored. The actual implementation of the context repository can include the use of an underlying DBMS system and feature a cache system as needed. This service facilitates the customization of the middleware to the targeted platform. For instance, in the Android port of the middleware, the natively built SQLite database is utilized for realizing the repository.

3.2.2 Core functionality

The core functionality of the context manager includes the binding with context providers and the servicing of context clients. Context plug-ins, for instance, are bound to `IContextAccess` and `IContextManagement`, as well as to the `IContextPlugin` service. Through the first two, the plug-ins can access additional context data and report new context information. The latter is used by the middleware for controlling when the plug-ins are started or stopped.

When a new context-aware application is started, it naturally registers its interest for certain context types. It does so by using the `IContextAccess` service, either implicitly (i.e., by registering for asynchronous notification) or explicitly (i.e., by specifying context types of interest). This information is used by the context manager, which continuously and dynamically decides which plug-ins must be activated (see Sect. 3.1.3).

The context manager also uses a repository for the purpose of storing context data for long-term use. To facilitate various implementations of such a repository, the `IContextRepository` internal service is defined. By default, all context data are stored in the repository and become subject to querying. Certain policies control the garbage collection of outdated data.

3.2.3 Architectural variability

Besides its core functionality, the middleware architecture also facilitates extending its functionality for context visualization, simulation, testing, and distribution. Furthermore, alternative realizations can be provided for realizing the roles of the context model, the repository, and query processor. An example illustrating this variability is shown in Fig. 2.

This example illustrates two alternative realizations of the `IContextModel`, two more realizations of the `IContextRepository`, and a single optional realization of the `IContextQuery` interface.

For instance, the context model can be backed by an ontology, requiring a more elaborate codebase for handling XML and OWL-based documents. But, when simpler applications are used and/or a more lightweight platform is targeted, a simplified, object-based implementation of the model can be a better match (albeit harder to maintain).

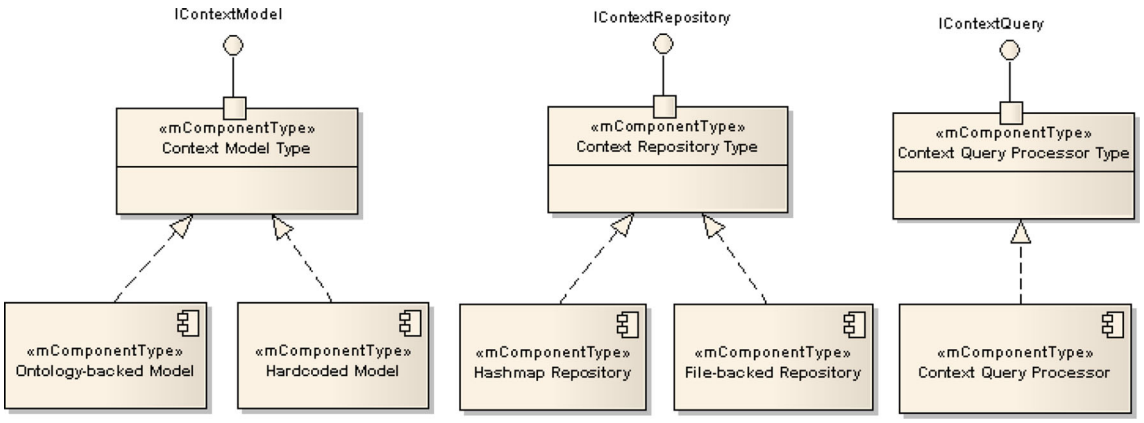

Fig. 2 Variability of the context middleware architecture

Similarly, the context repository can be customized to the targeted platform. For instance, in embedded devices, a small codebase footprint is essential, rendering a custom, object-based repository the favored solution. On the other hand, platforms such as Android feature built-in databases, making them ideal for building the repository on them.

Finally, the fact that the CQP is an optional component—along with the interchangeability of the context model and the repository—increases the modularity of the system: Different configurations of the framework can be easily produced and finetuned to the targeted platforms and their resource profiles. For instance, when only time-based context queries are needed, then the CQP bundle can be omitted altogether, and a lightweight implementation of the context cache can be used. On the other hand, when more demanding context queries are needed and the deployment system is sufficiently resourceful, the *context query processor* can be included along with a more elaborate implementation of the context repository, such as one based on a full-scale *Database Management System* (DBMS).

Besides the modularity inside the context middleware boundaries, the offered services also facilitate extending it with additional functionality. For instance, the base architecture can be augmented with context distribution functionality The prototype implementation of this architecture, discussed in the next subsection, features such optional components, extending its capabilities with a SIP-based context distribution system as well as with a swing-based context simulator.

3.3 Implementation

3.3.1 OGSGi-based implementation

The proposed pluggable and modular architecture was originally implemented on *OSGi Alliance*'s Java-based service platform (commonly referred to as OSGi) [8, 10]. The adoption of a standardized framework results in a smoother learning curve for new adopters and was selected to make the framework more appealing to the developers.

OSGi supports a simple—yet powerful—component lifecycle, which allows for dynamic installation, update, resolution and activation of components. This is illustrated in Fig. 3, which depicts both the original OSGi component lifecycle, along with the extensions, namely `C_INSTALLED`, `C_RESOLVED`, and `C_ACTIVE`, added to accommodate the mechanisms described in Sects. 3.1.2 and 3.1.3.

Multiple OSGi implementations are already available, and the context middleware architecture prototype implementation was tested on many of them (including Equinox and Knopflerfish). Using a special porting of OSGi on the Dalvik virtual machine, the architecture has also been ported and tested on the Android platform.

3.3.2 Android-based implementation

The OSGi-based codebase was initially ported to Android, but not natively; rather, the full middleware layer and all the plug-ins were bundled in a single package, making little use of the underlying Android architecture. A more recent realization of the middleware aimed at realizing the same architecture and mechanisms described in Sect. 3, while making full use of the Android architecture and services.

Finally, since the middleware aims for mobile and pervasive computing environments, the codebase was kept as small and as portable as possible. As of version 0.5.0 of the middleware (OSGi-based implementation), the context management system bundle had a size of 162 kb (this includes the sub-components of the context model, context repository, and CQP). Furthermore, to keep the code as portable as possible, the middleware was implemented using J2SE version 1.4, which is supported by many JVMs built for mobile devices. The package containing the required libraries and runtime for the Android implementation, as of version 0.2.0, was just 60 kb. The source code

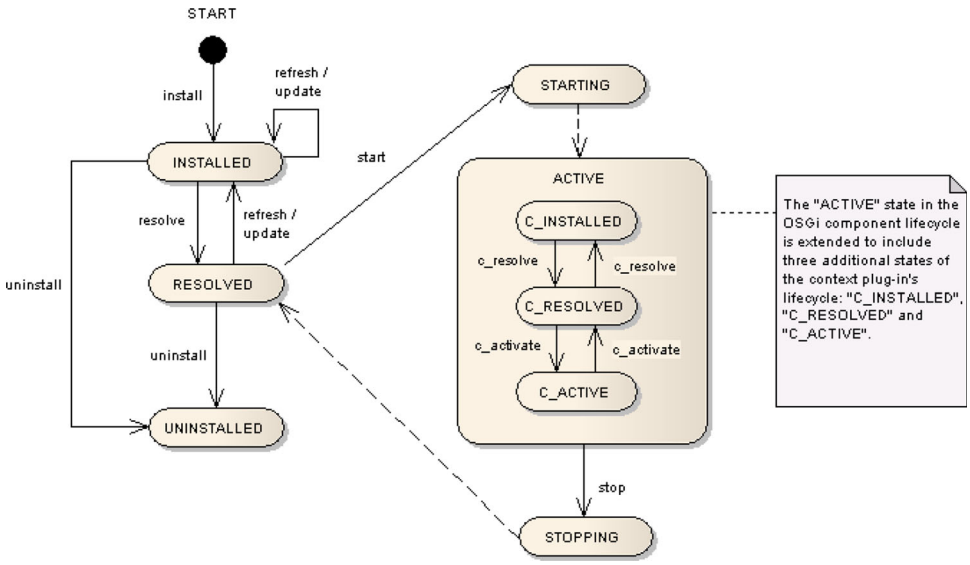

Fig. 3 Extended OSGi component lifecycle state diagram

of both the OSGi- and Android-based implementations is available under the LGPL open-source license (at the BerliOS and Google code repositories respectively. [1,2])

4 Case study: context-aware media player

To facilitate the evaluation of the proposed development methodology and its complementary middleware architecture, we present a case study application to describe the process followed to develop it, as well as how the middleware is used to run it. This case study builds on the OSGi-based implementation. While the CaMP is not a complex application, it was specifically selected to emphasize the development methodology and the use of the middleware.

4.1 Developing the context-aware media player

The *Context-aware Media Player* (CaMP) application extends a typical media player application with the ability to be aware of when the user is entering—or exiting—their office, so that it can automatically resume—or suspend—the media playback accordingly. From a functional perspective, CaMP is a rather trivial context-aware application, which simply needs to be asynchronously notified of when the user enters or exits the room (i.e., where CaMP is deployed). In this respect, separating the concern of its business logic (i.e., media playback) from that of its context-aware behavior (i.e., automatically resuming or suspending the media playback) is straightforward.

CaMP consists of a main component, which makes use of the *Media Stream* and the *Media Player* components. The first one is used to access a media stream (e.g., a URL of an online music streaming service), and the latter is used to convert the media stream to audible music, played on the device's speakers. It should be noted that the latter also includes methods for resuming and suspending the media playback, which are controlled by the CaMP component. Finally, the CaMP component—and, consequently, CaMP application—has a dependency on the context type specified by the entity type `Environment|room` and the specific scope `UserInTheRoom`. These three components are all packaged in a single OSGi bundle. This architecture is illustrated in Fig. 4.

4.1.1 *Developing the user-in-the-room context plug-in*

As plug-ins are common OSGi bundles, the first step is to define their manifest file, which is the one specifying the static dependencies of the plug-in (i.e., which libraries are required). These dependencies are defined as per OSGi's and Declarative Service's standard practice, pointing to the needed libraries both inside and outside the middleware.

Since context plug-ins are also controlled by the *Declarative Services* specification, the next step is the specification of the service descriptor file. This XML-formatted descriptor specifies which services are provided and, also, which ones are required by the plug-in. By default, context plug-ins provide one service: the `IContextPlugin`, discussed earlier in Sect. 3.1. Furthermore, the service descriptor includes optional parameters which, in this case, are used to specify the context types offered by the plug-in. In the case of context sensors, only *provided* context types are specified, while in the case of context reasoners—such as the *User-in-the-room* plug-in—the *required* context types are also specified.

[1] MUSIC, https://developer.berlios.de/projects/ist-music/.

[2] RSCM, http://code.google.com/p/rscm/.

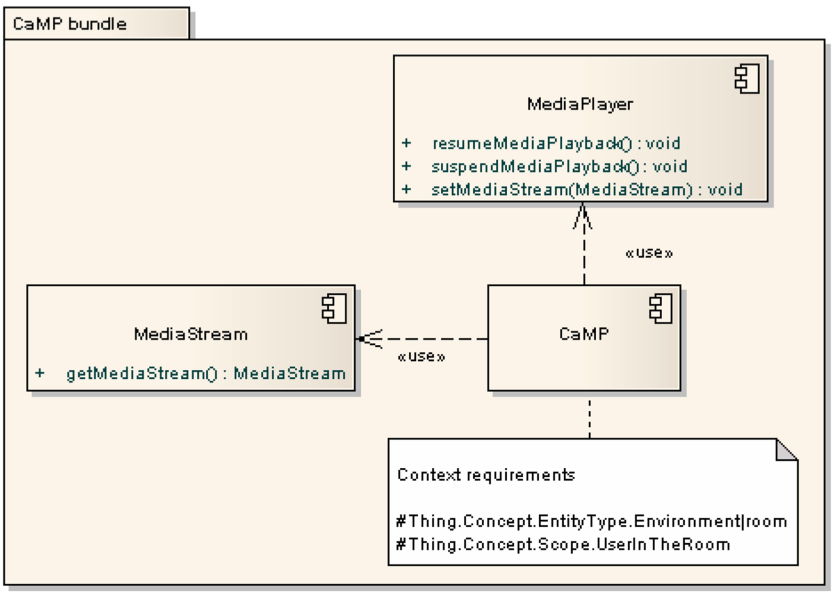

Fig. 4 The context-aware media player business logic

The context middleware provides two abstract classes that can be extended (using standard object-oriented inheritance) to realize context sensor and context reasoner plug-ins. Besides realizing the fundamental functionality specified in the `IContextPlugin` interface, the class defined as `AbstractContextPlugin` also provides predefined methods for *firing* context events, as well as for intercepting incoming context data. When the plug-in's context sensing logic is implemented, it uses this method to communicate context events to the middleware. Plug-in realizations also implement the `activate` and `deactivate` methods, which are automatically controlled by the middleware (discussed in Sect. 3.1.3), so that the sensing is resumed when activated and suspended when deactivated.

4.1.2 Reusing the bluetooth and motion sensor plug-ins

The Bluetooth sensor plug-in monitors and reports if a predefined Bluetooth device (e.g., a smartphone carried by the user) is near her or his office (simple detection by the adapter fixed on the workstation computer is sufficient as typical Bluetooth adapters have a small range of just a few meters). In this way, this plug-in senses whether the specified smartphone, and consequently its owner, is nearby with relatively high accuracy.

The motion sensor plug-in, on the other hand, monitors and reports the motion activity sensed in a room. This is achieved by periodically taking pictures using a Web camera and comparing them, pixel-by-pixel. When two consecutive pictures are found to have significant differences (e.g., over 10 % of their pixels are changed), then a context event is raised. The actual percentage of difference (i.e., the *delta* of the pictures) is used to characterize the accuracy of the event; as a larger delta implies a higher probability, there is actual motion in the room.

To facilitate reusability of software components, public repositories are used. A context plug-in repository is a public directory listing existing, reusable plug-ins that can be used as off-the-shelf components. The attributes of the plug-ins (e.g., supported platforms, limitations, etc) are also listed along with the plug-ins. One such repository was implemented in the scope of the MUSIC project and is available at its Web site.

4.1.3 Binding with the middleware

Having acquired the three plug-ins to be used by the CaMP, the next step is to bind the application to the middleware. Just like all OSGi bundles, CaMP defines a manifest and a service descriptor. The former is a typical manifest file, and the latter is a simple expression of CaMP's dependency on the context access service. Notably, the context access service can be defined as *optional*, which enables the application to be deployed and launched even in the absence of the context middleware (or the context plug-ins). This is possible, because in the case of the CaMP application, the context-aware behavior is treated as *additional* rather than a *required* functionality.

Finally, the code implementing CaMP's context-aware behavior is shown in listing 1 (some details were omitted or simplified to avoid cluttering). Note that the only link to the requested context information is established via the specified *entity* and *scope* parameters. In other words, the component has a logical dependency only on the context type and not on any specific context plug-in. This allows seamless replacement of alternative plug-ins providing the same context type.

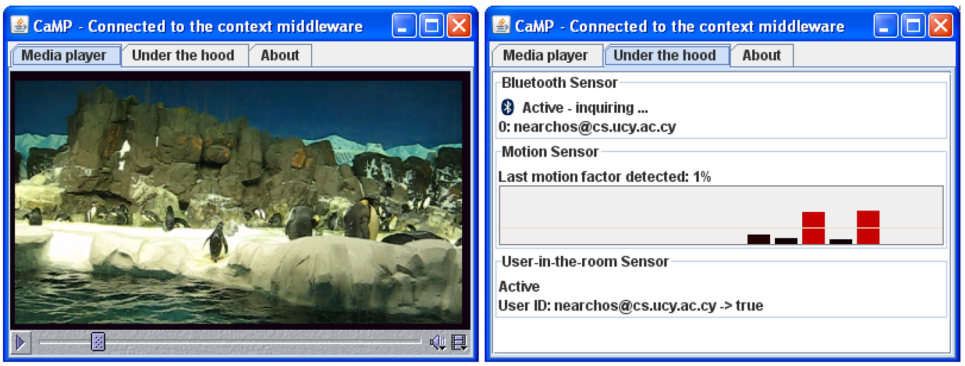

Fig. 5 Screenshots of the context-aware media player

When connected to (or disconnected from) the context middleware, the `setCtxtAccess`/`unsetCtxtAccess` methods are automatically invoked by the *Declarative Services* run-time system. When the first method is invoked, the application uses the context access service reference to register itself for asynchronous notification of the specified entity/scope pair. When the second method is invoked, the application uses the same service reference to cancel the previous registration. This behavior allows the context middleware to be aware of the context needs of the application and thus automatically handles the lifecycle of the corresponding plug-ins.

```
public class CaMP implements IContextListener
{
  public static final IEntity ENTITY = ...;
  public static final IScope SCOPE = ...;

  private final MediaPlayer mp
                        = new MediaPlayer();

  public CaMP() { ... // init the GUI }

  public void setCtxtAccess(ICtxtAccess ca) {
    ca.addCtxtListener(ENTITY, SCOPE, this);
  }

  public void unsetCtxtAccess(ICtxtAccess ca) {
    ca.removeCtxtListener(ENTITY, SCOPE, this);
  }

  public void ctxtChanged(Ctxt..Evnt event) {
    IContextElement contextElement = event...;
    IValue value = contextElement..getValue();
    BooleanValue booleanValue = (..) value;
    if(booleanValue.booleanValue())
      mediaPlayer.start();
    else
      mediaPlayer.stop();
  }
}
```

Listing 1 The implementation code of the Context-aware Media Player

When context events of the requested type are created, those are communicated via the middleware to the CaMP application via the `contextChanged` method. In this case, this method is simply used to extract the boolean value abstracting whether the user is in the room or not. Based on the value encoded in the received event, the media player is started (i.e., resumed) or stopped (i.e., suspended) accordingly.

4.1.4 Deploying the context-aware media player

The three context plug-ins and the CaMP application are all packaged as individual JAR-based OSGi bundles, which are installed along with the context middleware—which is also an OSGi bundle itself.

Once the plug-ins are installed, they are automatically discovered by the middleware which registers their metadata and attempts to resolve them (see Sect. 3.1.2). When the CaMP application is started, the context access service is bound with it, and the former subscribes for notification of relevant context events. This triggers the context middleware to reevaluate the offered and needed context types, and activate the three plug-ins as needed (see Sect. 3.1.3).

The appearance of the application's tabs is shown in Fig. 5. The first tab shows the media player and its manual controls (a single button, which resumes or suspends the playback). The latter provides an *under-the-hood* view of the plug-ins for demonstration purposes. At the top, the Bluetooth sensor viewer shows that the plug-in is active and that it has discovered a Bluetooth device identified as "nearchos@cs.ucy.ac.cy." The second viewer shows that the motion sensor plug-in is also active and further displays a histogram with the recent motion values reported (two of them exceeding the threshold). Finally, the user-in-the-room sensor viewer at the bottom simply displays a boolean value indicating whether the predefined user is detected at present or not. While the application runs, the user can enter or exit their room, with the application adapting accordingly based on the changing context detected by the plug-ins. More details about the CaMP, as well as additional configurations of the application, are described in [21].

5 Evaluation

The evaluation of the proposed architecture and its prototype implementations consists of both quantitative and qualitative analysis. The former was performed by experimentally studying how the middleware was used by developers implementing context-aware applications. The latter was achieved through the analysis of the requirements identified in Sect. 2 with respect to the proposed middleware architecture.

5.1 User-based, quantitative evaluation

This section describes how the middleware architecture was quantitatively analyzed and evaluated. This was achieved by collecting and evaluating feedback received from developers who used both the middleware and a companion methodology [22] to produce context-aware applications. The evaluation comprises two parts: First, the experience of developers working on pilot applications [8] was collected and analyzed. Second, a number of undergraduate students were instructed the development methodology and the use of the middleware. Then, they were asked to use them to build context-aware applications as part of some lab assignments in a course on context-aware systems. Their feedback was collected through questionnaires and analyzed to identify the strengths and weaknesses of the proposed approach [22].

While this form of evaluation was somewhat limited in terms of numbers of participants, it has nevertheless achieved to provide valuable insight concerning the advantages and limitations of the proposed methodology and the underlying middleware architecture. Inevitably, the formality of such quantitative evaluation is limited as a result of the lack of standard evaluation methods in relation to context-aware applications, as it was also argued in [12].

5.1.1 Evaluation by developers

The first form of evaluation was performed with developers implementing a number of pilot applications in the context of the MUSIC project. The developed pilot applications require context data not only for direct use by them, but also for allowing an adaptation middleware to select and apply the optimal application variant. In this regard, the developers of the pilot applications were often able to use the context middleware in a seamless way (i.e., by using basic context types provided by plug-ins that were already bundled with the MUSIC middleware, such as the resource sensors). In some cases, however, the pilot developers had to develop their own custom plug-ins. These developers had long programming experience and were familiar with OSGi-based middleware architectures.

The pilot developers were asked to compare the development of a context-aware application using the proposed model and middleware versus using an ad hoc development approach. Two of them stated that they would use the proposed solution again if they needed to develop more context-aware applications, and the other two said they would use it again unless the targeted applications were too simple, in which case they would use an ad hoc approach. All four of them agreed that the tasks for developing a plug-in were of low-to-medium complexity and that the hardest concept was the design and realization of the context model.

5.1.2 Classroom-based evaluation

As part of an undergraduate course at the University of Cyprus, the students were asked to implement context-aware applications by following the methodology presented in this paper and by using the provided middleware framework. These applications were constructed by realizing the business logic of the application and by producing new, or reusing existing, context plug-ins as per the proposed methodology. With these, the students formed and deployed their applications by integrating the appropriate components (i.e., plug-ins and business logic) with the middleware. Finally, they presented their applications and provided their feedback by answering some questionnaires. The aim of those questionnaires was to evaluate the strengths and weaknesses of the methodology and the middleware.

As the students had limited programming experience in general and no experience with OSGi, they were first provided a quick introduction to OSGi. Next, they were given an introduction to the middleware architecture and a tutorial on how to develop context plug-ins using the proposed methodology. Eventually, they were asked to complete a practical lab assignment, in three steps:

- *Describe a context-aware application* specify its business and context-aware logic, and identify the required context types and corresponding plug-ins.
- *Realize the context plug-ins* develop some plug-ins from scratch and also reuse some existing ones (perhaps developed by other students).
- *Implement the application's business logic* Use the context data provided by the developed (or reused) plug-ins and deploy all of them on the middleware.

This structured approach has helped teach the students the basics of context-aware applications, as well as the benefits of component orientation. By getting hands-on experience with the development of context-aware applications, they were able to better grasp the complexity inherent in their development. Also, they were able to

experience the benefits of COTS-based development via the reuse of existing context plug-ins.

Once the assignments were completed, the students were asked to fill-in an anonymous survey, providing input about the advantages and disadvantages of the development methodology and the middleware architecture they used.

It should be noted that the participants did not consider a point-of-reference (e.g., a competing development platform) when they evaluated the approach proposed in this paper. Rather, their feedback compares the proposed approach to ad hoc development.

In summary, the collected feedback shows that

- Almost all of the students (11 of 12) would prefer to use the proposed development approach and the provided middleware—partly or completely—rather than an ad hoc approach, if they needed to develop more context-aware applications.
- Most tasks related to the development of the plug-ins were of low-to-medium complexity (except the task of creating the context model which was of medium-to-high complexity).
- On average, the students spent approximately 23 h in preparation (i.e., studying the material and the examples) and another 20 h for coding (of both the plug-ins and the business logic of the application), signifying a rather quick and smooth learning curve.

The complete details of the questions in this survey, along with the summary of the answers they provided, are available in [20].

5.2 Requirement-driven, qualitative evaluation

This section evaluates the requirements identified in Sect. 2 by revisiting them and evaluating the proposed development methodology and middleware architecture against them. It should be noted that as many alternative implementations are possible—depending on assumptions about how context information is queried, interpreted, etc.—it is not feasible to present a straightforward and extensive quantitative comparison (beyond, for instance, the one presented in the previous section). Therefore, the analysis in this section is mostly qualitative rather than quantitative, except where feasible. The following paragraphs discuss some functional and extra-functional requirements individually, arguing to which extend they have been addressed in the development methodology and middleware architecture. In some cases, directions for improvement are also proposed.

5.2.1 Evaluation of functional requirements

Functional are those requirements, which deal directly with features explicitly related to context-awareness.

Application-specific context acquisition, analysis and detection This fundamental requirement is addressed by the middleware architecture, which offers a dedicated context access service (see Sect. 3.2.1). The context-aware applications use it to request or register for their desired context types and let the middleware attend to the tasks of acquisition, analysis, and triggering. Furthermore, the context access service also supports a rich context query language [9], which allows context filtering based on predefined conditions. This empowers the developers to specify complex context queries rather than implementing complex context filtering logic inside their applications.

Context histories Providing access to historical context information is important because it enables advanced context reasoning methods, such as location prediction or WiFi signal strength prediction (for examples, refer to [21]). The proposed middleware architecture enables storing historical context data through a dedicated context repository service. Past context data are accessed via the context access service. Context data corresponding to specific time periods can be accessed simply by specifying time conditions.

Support for multiple concurrent applications The support for multiple concurrent, context-aware applications is an important requirement, especially as modern mobile devices are powerful enough and feature multi-task operating systems (such as the Android platform). Both the development methodology and the middleware architecture were designed to accommodate this requirement. The resulting methodology and the middleware architecture split the context-aware applications into context providers (i.e., context plug-ins) and context consumers (i.e., applications). The middleware uses a variant of the Blackboard architecture to connect the context providers to the context consumers in a seamless and dynamically adaptable manner. In theory, any number of context providers can be bound to the middleware, servicing any number of context-aware applications. In practice, these numbers are limited by the actual resource constraints of the device, but in this case, the middleware is not the bottleneck.

Transparent distribution of context This requirement is accommodated in two ways. First, the context model allows for globally valid, unambiguous references to context information [26]. Transformation from-and-to locally valid and globally valid context information (from both a semantic and a representational point-of-view) is also automatically undertaken by the context model [27]. Second, the middleware architecture provides functionality that allows third-party components to inquire the locally provided and required context types. This allows the implementation of context distribution systems, which access this information, and use it to form an extended, federated context space. For instance, a SIP-based

implementation of such context distribution system was implemented [2].

Privacy of context information Privacy concerns are raised in the case of context distribution, and also in the case where a mobile device is stolen or lost. As the context distribution system is treated in this paper as an external component, the main effort is placed on protecting the privacy in the latter scenario. In this case, the main step toward user privacy protection would be the realization of a password-protected, context repository system with data encryption, preventing unauthorized access to the user's data. While the implementation of such a system was beyond the scope of the work described in this paper, it should be noted that once available, such alternative context repository could be easily integrated with the middleware, as discussed in Sect. 3.2.3.

Traceability and control The original scope of this requirement was to allow the end-users understand the behavior of context-aware applications, enabling them to manually intervene when needed. The rationale for this functionality is to allow the end-users to trust the autonomous context-aware logic, and thus adopt the technology. However, the development methodology and the accompanying middleware architecture proposed in this paper primarily aim at facilitating the developers into designing and implementing complex context-aware applications, easier and more efficiently. The actual context-aware behavior, as well as the end-user's ability to control it, still remains largely in the hands of the developers. However, limited support for traceability is provided, either through specialized plug-in viewers (as shown in Fig. 5), or via the use of context viewers. In both cases, the user views the context values at run-time. It should be noted, however, that the context viewer was primarily designed for simulation and testing purposes, and is not really intended for use by the end-users, although custom context monitors and controllers are of course supported, as illustrated in the CaMP case study (see Sect. 4).

5.2.2 Evaluation of extra-functional requirements

Extra-functional requirements deal with general features, not necessarily explicit to context-awareness. While some of these requirements are adequately handled by the underlying OSGi framework, others require much more elaborate handling.

Code reuse Enabling code reuse was one of the key goals of the proposed middleware architecture. Adopting a model, which treats context providers as independent and pluggable components, greatly facilitates this goal. The developed components are treated as black boxes, where their internal functionality is hidden, and only their context offerings and context requirements are explicitly defined as metadata. These metadata are also used for publishing the plug-ins in component repositories, further facilitating code reuse.

Modularity The requirement of modularity was another important goal in the design of the middleware architecture. Allowing an architecture that can be easily configured to match the needs and constraints of different platforms is an important feature, especially in the context of mobile and ubiquitous computing, which are characterized by high heterogeneity and variability. The modularity of the architecture enables different configurations of the middleware, both lighter for resource-constrained devices and more powerful for demanding applications, as it was discussed earlier in Sect. 3.2.

Separation-of-concerns The separation-of-concerns is a popular method for easing the complexity of a problem by breaking it down to smaller, simpler ones. Furthermore, when the individual pieces are independent, then they can be developed in parallel by individual developers. With this rationale, the development methodology described in [22] separates the development of context-aware applications into the tasks of developing the context providers and the context consumers. The former are mostly realized as context plug-in components and are highly reusable. The latter are typically context-aware applications or middleware components and are only loosely coupled with the context providers. This approach has also the advantage that the applications can be more easily tested and debugged, because these tasks can be performed at the level of the individual components.

Dynamic behavior Dynamic updates are enabled by allowing new context plug-ins to be installed and activated at run-time. As the context providers and the context consumers are only loosely coupled, it is possible to have applications replace their context-aware logic at run-time in a seamless manner. This is naturally supported by the underlying OSGi framework, which allows for dynamic installation (or un-installation) and activation (or deactivation) of components. This important feature allows mobile context-aware applications to take advantage of richer context information when it becomes available, and rolling back to basic context data use when it becomes unavailable. For instance, when in a car, a high-fidelity GPS sensor can be activated, while when indoors, a less precise WiFi-based location sensor can be used.

Resource efficiency Accommodating resource efficiency is achieved in two ways: First, by adopting a modular, light-weight architecture, which minimizes the resources consumed by the platform itself. Second, by using an intelligent mechanism to activate and deactivate the context plug-ins as needed (see Sect. 3.1.3 and also the experimental results presented in [24]).

5.3 Related work

Context-aware systems have been the object of extensive research in recent years. Here, we discuss related work and compare it to our approach.

Unlike approaches, which aim at realizing the full extend of their context-aware logic inside the applications (e.g., CML [12, 14] and COSMOS [4, 29]), or approaches that focus on specific domains like ad hoc network topologies (e.g., EgoSpaces [17]), the presented middleware architecture aims at providing a flexible and customizable solution, which the developers can customize or extend as needed (similar to the approaches taken by the Context Toolkit [7] and the Aura's Context Information Service [16]).

The COSMOS approach is a component-based framework for managing context information in ubiquitous, context-aware applications. The basic structuring concept in COSMOS is the *node*, which can be thought of as similar to a context sensor or a context reasoner plug-in. Unlike plug-ins though, the nodes correspond to context types, and they are used to define logic hierarchies which are in turn used to define the context-aware behavior. On the other hand, the context plug-ins defined in our approach are software entities that can be installed/uninstalled and also activated/deactivated.

By providing a uniform abstraction of context information (i.e., the context node), COSMOS supports the composition of context information from low-level sensors to high-level policies. At the lowest level, context nodes reify hardware capabilities, software resources, or embedded sensors. At a higher level, context nodes reuse or develop composition operators to infer advanced context information. Evidently, this is similar to the context sensing and reasoning hierarchy proposed in [22]. However, the use of node-based context-aware logic is limited to runtime access by a single application only and does not facilitate activation or deactivation of the actual hardware sensors. In contrast, the proposed pluggable architecture facilitates sharing of context information among concurrent applications. Furthermore, it allows for automatic activation and deactivation of the plug-in components which is an important advantage for applications deployed on resource-constrained devices. As stated in [29], the core motivation of COSMOS is to isolate context management policies from applications and to enforce their reuse. Because context policies are themselves reflected as context nodes, they can be reused in different contexts. However, as the nodes in COSMOS correspond to finer-grained concepts (such as operators), reusing them can be quite elaborate. In contrast, the context plug-ins are well-defined components, as per the OSGi specification, with added context-specific annotations making them easier to reuse.

The CML is an extensive software engineering framework for enabling context-aware pervasive computing. The authors presented their view of context-awareness as a technique which enables pervasive computing by allowing applications and systems to act autonomously on behalf of users [11]. Henricksen and Indulska argue that their framework achieves this goal while also addressing three basic challenges that they identified: analysis of application's context requirements, acquisition and management of the relevant context data, and, finally, design and implementation of suitable context-aware behavior.

Unlike the current state-of-the-art, our approach offers a methodology and a middleware architecture for developing, deploying, and maintaining context-aware applications. This methodology separates the design and the development of context producers (i.e., sensor and reasoner plug-ins) from that of context consumers (i.e., context-aware applications). Furthermore, the underlying middleware handles a number of common functionalities such as lifecycle support for the plug-ins, context aggregation, context storing, and a rich query language. It also facilitates the development of autonomous, context-aware applications in which the decisions are taken using an elaborate hierarchical process. One of the most important advantages of the presented architecture is its modularity, which facilitates the formation of various instantiations matching the capabilities and the needs of the deployment environment. Also, unlike the related work, the proposed architecture allows for dynamically added/removed context sensors, and additionally, it facilitates their dynamic activation. As it was experimentally shown in [24], this ability provides significant resource optimization.

6 Conclusions and future work

The primary goal of the work described in this paper was to provide software engineering support for the development of context-aware applications. In this respect, a pluggable and modular middleware architecture was presented, enabling the development and deployment of context-aware, mobile applications. It was shown that this architecture features multiple benefits, including platform independence and resource optimization.

The proposed middleware architecture differs from related approaches because it was designed with the aim of being comprehensive, allowing the development of arbitrary context-aware applications. At the same time, a broad set of requirements were identified and used to guide the design and development of the methodology and the middleware, resulting in a highly dynamic and modular architecture. For instance, unlike the current state-of-the-art, the proposed middleware implements a pluggable architecture,

which allows the applications to take advantage of richer or more efficient context providers as they dynamically become available, even while moving about in space.

The pluggable and modular middleware architecture is presented in a case study application, the *Context-aware Media Player*. This case study illustrates the methodology used for developing context-aware applications, and the mechanisms are used for interfacing it with and deploying it on the middleware architecture. Based on this, the proposed architecture is evaluated against the identified requirements. In addition to this qualitative approach, the middleware architecture is also evaluated quantitatively, in the scope of a controlled experiment where selected researchers and the students of a course used them to develop context-aware applications and classified them favorably compared to ad hoc development approaches.

Acknowledgments The authors acknowledge the partial financial support given to this research by UCLan Cyprus and the EU (6th Framework Programme, contract number 35166).

References

1. ITU-international telecommunication union (2010) Measuring the information society: the ict development index. http://www.itu.int/ITU-D/ict/publications/idi/2010
2. Angeles-Pina C (2008) Distribution of context information using the session initiation protocol (SIP). Master of science thesis, KTH Information and Communication Technology
3. Brown PJ, Bovey JD, Chen X (1997) Context-aware applications: from the laboratory to the marketplace. IEEE Pers Commun 4(5):58–64. doi:10.1109/98.626984
4. Conan D, Rouvoy R, Seinturier L (2007) Scalable processing of context information with COSMOS. In: Proceedings of the 7th IFIP international conference on distributed applications and interoperable systems (DAIS'07), vol 4531. Springer, Paphos, pp 210–224
5. Dey AK (2000) Providing architectural support for building context-aware applications. Ph.D. thesis, Georgia Institute of Technology
6. Dey AK (2001) Understanding and using context. Pers Ubiquitous Comput 5(1):4–7
7. Dey AK, Abowd GD, Salber D (2001) A conceptual framework and a toolkit for supporting the rapid prototyping of context-aware applications. Hum Comput Interact 16(2):97–166
8. Floch J, Fra C, Fricke R, Geihs K, Wagner M, Lorenzo J, Soladana E, Mehlhase S, Paspallis N, Rahnama H, Ruiz PA, Scholz U (2012) Playing music building context-aware and self-adaptive mobile applications. Softw Pract Exp J. doi:10.1002/spe.2116
9. Fra C, Valla M, Paspallis N (2011) High level context query processing: an experience report. In: Proceedings of the 8th IEEE workshop on context modeling and reasoning (CoMoRea'11) in conjunction with the 9th IEEE international conference on pervasive computing and communication (PerCom'11). IEEE Computer Society, Seattle, Washington, USA
10. Hallsteinsen S, Geihs K, Paspallis N, Eliassen F, Horn G, Lorenzo J, Mamelli A, Papadopoulos GA (2012) A development framework and methodology for self-adapting applications in ubiquitous computing environments. J Syst Softw 85(12):2840–2859
11. Henricksen K, Indulska J (2004) A software engineering framework for context-aware pervasive computing. In: Proceedings of the 2nd IEEE annual conference on pervasive computing and communications (PerCom'04), pp 77–86. IEEE Computer Society, Orlando, Florida, USA
12. Henricksen K, Indulska J (2006) Developing context-aware pervasive computing applications: models and approach. Pervasive Mob Comput 2(1):37–64
13. Henricksen K, Indulska J, McFadden T, Balasubramaniam S (2005) Middleware for distributed context-aware systems. In: Proceedings of the 7th international conference on distributed objects and applications (DOA'05), LNCS, vol 3760. Springer, Agia Napa, pp 846–863
14. Henricksen K, Indulska J, Rakotonirainy A (2006) Using context and preferences to implement self-adapting pervasive computing applications. Softw Pract Exp 36(11–12):1307–1330
15. Hong JI (2005) An architecture for privacy-sensitive ubiquitous computing. PhD thesis, University of California, Berkeley
16. Judd G, Steenkiste P (2003) Providing contextual information to pervasive computing applications. In: Proceedings of the 1st IEEE international conference on pervasive computing and communications. IEEE Computer Society, Dallas-Fort Worth, Texas, USA, p 133
17. Julien C, Roman G (2006) EgoSpaces: facilitating rapid development of context-aware mobile applications. IEEE Trans Softw Eng 32(5):281–298
18. Kakousis K, Paspallis N, Papadopoulos GA, Ruiz PA (2010) Testing self-adaptive applications with simulation of context events. In: Proceedings of the 3rd DisCoTec workshop on context-aware adaptation mechanisms for pervasive and ubiquitous services (CAMPUS'10) in conjuction with the 10th IFIP international conference on distributed applications and interoperable systems (DAIS), electronic communications, vol 28. EASST, Amsterdam
19. Long S, Kooper R, Abowd GD, Atkeson CG (1996) Rapid prototyping of mobile context-aware applications: the cyberguide case study. In: Proceedings of the 2nd annual international conference on mobile computing and networking. ACM, Rye, pp 97–107
20. Paspallis N (2009) Middleware-based development of context-aware applications with reusable components. PhD thesis, University of Cyprus
21. Paspallis N, Achilleos A, Kakousis K, Papadopoulos GA (2010) Context-aware media player (CaMP): developing context-aware applications with separation of concerns. In: Proceedings of the IEEE Globecom 2010 workshop on ubiquitous computing and networks (UbiCoNet 2010). IEEE Digital Library, Miami, Florida, USA, pp 1–6
22. Paspallis N, Eliassen F, Hallsteinsen S, Papadopoulos GA (2009) Developing self-adaptive mobile applications and services with separation-of-concerns. In: Nitto ED, Sassen A, Zwegers A (eds) At your service: service-oriented computing from an EU perspective. MIT Press, Cambridge, pp 129–158
23. Paspallis N, Papadopoulos GA (2006) An approach for developing adaptive, mobile applications with separation of concerns. In: Proceedings of the 30th annual international computer software and applications conference (COMPSAC '06), vol 1. IEEE Computer Society Press, Chicago, pp 299–306
24. Paspallis N, Rouvoy R, Barone P, Papadopoulos GA, Eliassen F, Mamelli A (2008) A pluggable and reconfigurable architecture for a context-aware enabling middleware system. In: Proceedings of the 10th international symposium on distributed objects, middleware, and applications (DOA'08), LNCS, vol 5331. Springer, Monterrey, pp 553–570
25. Paymans TF, Lindenberg J, Neerincx M (2004) Usability trade-offs for adaptive user interfaces: ease of use and learnability. In:

Proceedings of the 9th international conference on intelligent user interfaces. ACM, Funchal, pp 301–303
26. Reichle R, Wagner M, Khan M, Geihs K, Lorenzo J, Valla M, Fra C, Paspallis N, Papadopoulos GA (2008) A comprehensive context modeling framework for pervasive computing systems. In: Proceedings of the 8th IFIP international conference on distributed applications and interoperable systems (DAIS'08), LNCS, vol 5053. Springer, Oslo, pp 281–295
27. Reichle R, Wagner M, Khan MU, Geihs K, Valla M, Fra C, Paspallis N, Papadopoulos GA (2008) A context query language for pervasive computing environments. In: Proceedings of the 5th IEEE workshop on context modeling and reasoning (CoMoRea'08) in conjunction with the 6th IEEE international conference on pervasive computing and communication (PerCom'08), pp 434–440. IEEE Computer Society, Hong Kong. doi:10.1109/PERCOM.2008.29
28. Romero JJ (2011) Smartphones: the pocketable pc. IEEE Spectrum. Available online at http://spectrum.ieee.org/telecom/wireless/smartphones-the-pocketable-pc
29. Rouvoy R, Conan D, Seinturier L (2008) Software architecture patterns for a context-processing middleware framework. IEEE Distrib Syst Online 9(6):1
30. Salber D, Dey AK, Abowd GD (1999) The context toolkit: aiding the development of context-enabled applications. In: Proceedings of the SIGCHI conference on human factors in computing systems. ACM, Pittsburgh, pp 434–441
31. Schilit BN, Adams NI, Want R (1994) Context-aware computing applications. In: Proceedings of the 1st workshop on mobile computing systems and applications (WMCSA'94). IEEE Computer Society, Santa Cruz, CA, pp 85–90
32. Szyperski C (1997) Component software: beyond object-oriented programming. Addison-Wesley Professional
33. Want R, Hopper A, Falco V, Gibbons J (1992) The active badge location system. ACM Trans Inf Syst 10(1):91–102
34. Want R, Schilit B, Adams N, Gold R, Petersen K, Goldberg D, Ellis J, Weiser M (1996) The parctab ubiquitous computing experiment. In: Mobile computing, the springer international series in engineering and computer science, vol. 353. Springer, Berlin, pp 45–101
35. Weiser M (1993) Hot topics: ubiquitous computing. IEEE Comput 26(10):71–72
36. Yau SS, Karim F, Wang Y, Wang B, Gupta SKS (2002) Reconfigurable context-sensitive middleware for pervasive computing. IEEE Pervasive Comput 1(3):33–40