

2019-01-06

Context Oriented Software Middleware

Basel Magableh

Technological University Dublin, 453543@tudublin.ie

Follow this and additional works at: <https://arrow.tudublin.ie/scschcomart>



Part of the [Computational Engineering Commons](#)

Recommended Citation

Magableh, B. (2019). Context Oriented Software Middleware. *International Journal of Computer Science and Information Technology*, 1(1). doi.org/10.21427/yndh-x108

This Article is brought to you for free and open access by the School of Computer Sciences at ARROW@TU Dublin. It has been accepted for inclusion in Articles by an authorized administrator of ARROW@TU Dublin. For more information, please contact arrow.admin@tudublin.ie, aisling.coyne@tudublin.ie.



This work is licensed under a [Creative Commons Attribution-NonCommercial-Share Alike 4.0 License](#)

Context Oriented Software Middleware

Basel Magableh

*School of Computer Science,
Dublin Institute of Technology,
Technological University,
Dublin, Ireland*

Abstract

This article proposes a new paradigm for building an adaptive middleware that supports software systems with self-adaptability and dependability. In this article, we wish to explore how far we can support the engineering of self-adaptive applications using a generic and platform-independent middleware architecture provided by non-specialised programming languages such as Context-Oriented Programming (COP), and Aspect-Oriented Programming (AOP), and not limited to a specific platform or framework. This gives the software developers the flexibility to construct a self-adaptive application using a generic and reusable middleware components that employ popular design patterns, instead of forcing the software developers to use a specific programming language or framework to implement self-adaptive software in mobile computing environment, which are heterogeneous and resource-constrained.

Our middleware approach, Context-Oriented Software Middleware (COSM), supports context-dependent software with self-adaptability and dependability in a mobile computing environment. The COSM-middleware is a generic and platform-independent adaptation engine, which performs a runtime composition of the software's context-dependent behaviours based on the execution contexts. Our middleware distinguishes between the context-dependent and context-independent functionality of software systems. This enables the COSM-middleware to adapt the application behaviour by composing a set of context-oriented components, that implement the context-dependent functionality of the software. Accordingly, the software dependability is achieved by considering the functionality of the COSM-middleware and the adaptation impact/costs. The COSM-middleware uses a dynamic policy-based engine to evaluate the adaptation outputs, and verify the fitness of the adaptation output with the application's objectives, goals and the architecture quality attributes. These capabilities are demonstrated through an empirical evaluation of a case study implementation.

Keywords: mobile middleware, context oriented programming, component composition, self-adaptive application, context oriented software development

1. Introduction

Mobile computing infrastructures make it possible for mobile users to run software systems in heterogeneous and resource-constrained platforms. Mobility, heterogeneity, and device limitations create a challenge for the development and deployment of mobile software. Mobility induces context changes to the computational environment and therefore changes to the availability of resources and services. This requires software systems to be able to adapt their functionality/behaviour to the context changes (Inverardi and Tivoli, 2009). This class of software systems are called Context-dependent/self-adaptive applications, which have the ability to modify their own structure and behaviour in response to context changes in the environment where they operate (Oreizy et al., 1999). Such level of self-adaptability presents the challenge of tailoring behavioural variations dynamically to both a specific user needs and adapt to the context changes. Moreover, because of the software pervasiveness, and in order to make adaptation effective and successful, adaptation processes must be considered in conjunction with software dependability and reliability by

Email address: `basel.magableh@dit.ie` (Basel Magableh)

providing dynamic verification and validation mechanism, which validates the adaptation output with the adaptation goals, objectives, and architecture quality attributes (Cheng et al., 2008; de Lemos et al., 2011). Software dependability refers to the degree to which a software system or component is operational and accessible when required for use (Barbacci, 2004).

We believe that software self-adaptability and dependability can be achieved by dynamically composing software from context-oriented modules based on the context changes rather than composing the software from functional-oriented modules. Such composition requires the design of software modules to be more oriented towards the context information rather than being oriented towards the functionality. The principle of context orientation of software modules was proposed in the Context Oriented Software Development (COSD) (Magableh and Barrett, 2012a). COSD proposes a decomposition mechanism of software based on the separation between context-dependent and context-independent functionality. Context-independent functionality refers to software functionality whose implementation is unaffected by the context changes. For example, the map view and user login forms in mobile map application are context-free functionality (i.e. context changes would not change their functionality). The context-dependent functionality refers to software functionality, which exhibits volatile behaviour when the context changes. Separating the context-dependent functionality from the context-independent functionality enables software systems with adaptability and dependability with the aid of middleware technology. The middleware can adapt the software behaviour dynamically by composing that interchangeable context-dependent modules based on context changes. We argue that software self-adaptability is achieved by having both an adaptive middleware architecture and a suitable decomposition strategy, which separates the context-dependent functionality from the context-independent functionality of the software systems.

The objective of this research is to explore how far it is possible to support the development of self-adaptive applications using generic and platform independent middleware architecture provided by a non-specialised programming language such as Context-Oriented Programming (COP), and Aspect-Oriented Programming (AOP), and not limited to a specific platform or adaptation mechanism. This gives the software developers the flexibility to construct a self-adaptive application using generic reusable middleware components that employ popular design patterns and to implement their applications in several mobile computing platforms.

Our COSM-middleware is a generic and platform-independent adaptation engine, which performs a runtime behavioural composition of the context-dependent functionality of the software based on the operational context. The software dependability is achieved through evaluating the middleware functionality and the adaptation impact/costs. The COSM-middleware uses a dynamic policy-based engine to evaluate and verify the fitness of the adaptation output among the application's objectives, goals and the architecture quality attributes. These capabilities are demonstrated through an empirical evaluation of a case study implementation. The implementation targets the construction of a self-adaptive map personalisation application, called eCampus. The objective of this evaluation is to measure the impact of the COSM-middleware implementation over the allocated resources of a mobile device. The evaluation comes in two folds. The first experiment compares the performance and self-adaptability of the case study application implemented using COSM-middleware and Dynamic Aspect Oriented Programming (DAOP) engine. The second experiment compares the performance of the COSM-middleware implementation with several frameworks and middleware architectures, including Java Context-Oriented Programming (JCOP) (Schuster et al., 2011), Java Context Oriented Language (JCOOL)(Sindico and Grassi, 2009), Mobile USers In Ubiquitous Computing (MUSIC)(Geihs et al., 2011), and Mobility and ADaptation enAbling Middleware (MADAM)(Mikalsen et al., 2006).

This article is organised as follows: Section 2 illustrates a case study implemented by COSM-middleware. Section 3 provides an overall description of the COSM-middleware internal components. A detailed description of the COSM implementation is provided in Section 4. The Context-Oriented Software Middleware is evaluated in terms of energy utilisation and adaptation time as discussed in Section 5. Section 6 provides an overview of the related work.

2. Case study: Self-adaptive Map Application

eCampus is a self-adaptive map application, which helps students, lecturers, staff members and visitors to explore the campus of the National University of Ireland, Maynooth (NUIM). In order for context-awareness capabilities to be made available, different aspects of spatial data need to be exploited including location, semantics, and time. The context-aware functionality offered in this application can be very useful to staff, students, visitors and the general public alike when navigating and otherwise interacting within our eCampus environment, specifically but also within

any local environment generally while at home or on-site through web-based or smartphone connections. There are two types of users envisioned for this application: registered and non-registered users. A registered user would include anyone who wishes to log on (e.g. using their student/staff ID) and a non-registered user would include visitors to the campus (e.g. general public) who are not required to log on. As such, both user types are presented with different levels of functionality. The registered users profiles are recorded so that their personal timetables and interests can be displayed. This can be displayed with a grid/table of all activities for the user on the current day (course, lectures during the day/week) associated with the venue. The other option is that the schedule is directly displayed on the personalised map where an overlaid vector layer identifies the geometry coordinates of the venue and detailed information about that activity. The recommendation of events is suited to each user profile (ie. based on their interests). The application selects the most related events to the user based on his profile. The features on the map refer to point-of-interest classified according to their score attribute (calculated based on user's interests).

In our case study, eCampus is required to adapt its behaviour and offer different levels of personalisation to the users depending on the available resources. For example, eCampus is required to adapt to the condition of low battery. This can be achieved by using a location service that consumes less power. The "Battery level" requirement needs a Decision Policy (DPL) to manage its context changes and display less geospatial data to suit the battery level. In other words, the map application monitors the battery level and the bandwidth connectivity. If the battery level is high and healthy, eCampus uses the GPS service with more accurate location's update, displays more features to the user, and animates the features' appearance on the map when they are recommended. If the battery level is less than a specific limit, a WIFI-based location is used to save the battery energy and obtain less geospatial data. Finally, if the battery level is low, the eCampus application switches off the GPS or WIFI location services and uses the cell-tower location services. Using a cell-tower location reduces the accuracy of the location but saves battery energy. In addition, the application may reduce the number of features it displays on the map based on different interest score levels (values).

3. Context-Oriented Middleware

Before demonstrating the design of the context-oriented middleware, this section provides a general overview of the COSD methodology. According to the COSD methodology, context-oriented software is built from a set of components that follows the context-oriented component model (Magableh and Barrett, 2012a). Figure 1 shows a conceptual diagram of context-oriented component. A context-oriented component consists of three major parts: static and dynamic parts, and a delegate object. The static part implements a context-independent code fragment responsible for implementing the application core functionality. The dynamic part implements the context-dependent functionality and participates in the context-driven adaptation. The dynamic part consists of multiple layers. Each component layer implements a specific context-dependent functionality, which implements a decision policy to be used for adapting specific application behaviour. A layer is executed only if the associated context condition is found in the environment at runtime. For example, small display, low memory, and low battery are contextual conditions that need to be considered in the adaptation.

The COSM-middleware invokes a specific layer in the execution that suits only the current condition of execution context. For example, the COSM-middleware needs to reduce the amount of geospatial data to consider low battery condition in a mobile map application. A decision-policy is implemented inside a specific layer method. The method is associated with the low battery condition. This method will be executed whenever the COSM-middleware receives the notification *BatteryLevelWillChange*. In this case, the COSM-middleware executes the layer method, which implements a decision-policy that displays less amount of geospatial data for the user (for example, displaying the features that have interest score between "0.7" and "1"). This approach can personalise the map content based on the availability of computational resources.

The context-oriented component is given an opportunity to do dynamic context-driven adaptation by executing different layers, which implement different decision policies and methods implementation. Each layer must implement two or more methods that encapsulate decision-policies and associate them with context conditions. For example, the two methods inside the layer class of the context-oriented component, could have the names *ContextConditionDidChange* and *ContextConditionWillChange*. This allows the context-oriented component to perform adaptation action about a specific context condition in active or proactive mode. Active adaptation refers to the ability of the COSM-middleware to execute a layer when the condition is currently found in the execution context. Proactive adaptation refers to the ability of the COSM-middleware to execute a decision-policy that will handle a condition that can happen

after a certain amount of time. An example of active adaptation is when the user's location changes. The associated layer will execute a piece of code, that displays the user's location and nearby point-of-interests. On the other hand, an example of proactive adaptation is when the user's mobile is capturing high speed and acceleration. Our middleware can decrease the frequency of location-updates from the web mapping service until the speed of the mobile device is decreased to save allocated resources. More frequent updates of geospatial data is a power consuming process that needs more bandwidth and CPU throughput, consumes the allocated resources and decreases the battery life.

The fourth part of the context-oriented component is the delegate object see (Figure 1). The idea of using a delegate object is that two components coordinate to solve a problem. A context-oriented component is general and intended for reuse in a wide variety of contextual situations. The base-component is the digital map object, which stores a reference to that context-oriented component (i.e. its delegate) and sends messages to inform the delegate (context-oriented component) that some context condition was changed. This gives the delegate an opportunity to de/activate a layer implementation (i.e. rule implementation) dynamically (Buck and Yacktman, 2010).

Our map application case study is designed from a set of base-components (the context-independent components) and context-oriented components. Whenever the map application notifies the COSM-middleware about changes on context, the middleware activates a layer implementation that adapts to the changes by specifying an adaptation action that suits the context condition and considers the allocated resources. The following describes the structure of context-oriented software and the internal components of the COSM-middleware.

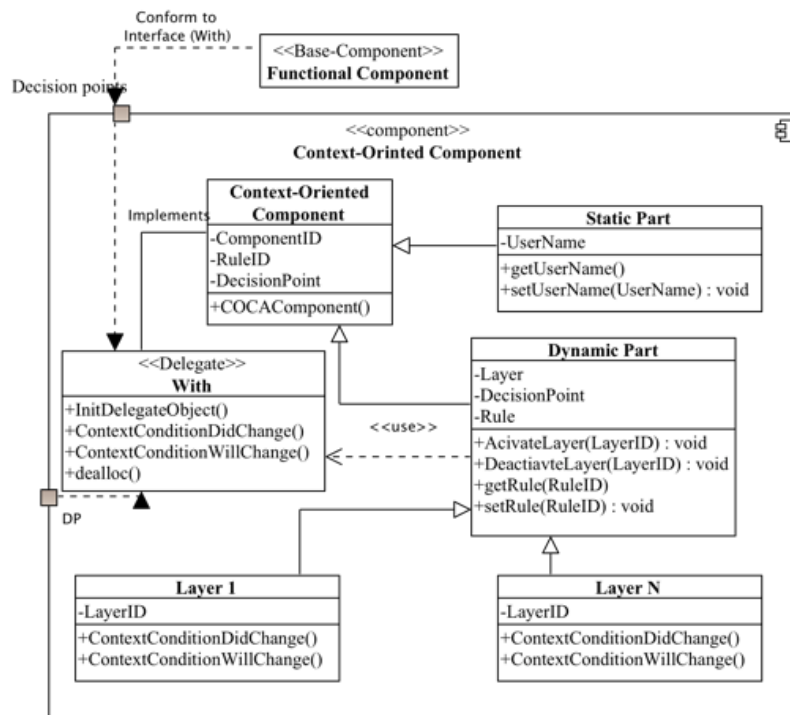


Figure 1: Context-Oriented Component Model

In this article, we explore the design practices that can be used to implement the middleware architecture without relying on a specific framework for performing dynamic adaptation in context-dependent software systems. To address this issues, this section focuses on describing the COSM-middleware architecture. The result of COSD methodology is a component-based architecture described by a Context-Oriented Component-based Applications Architecture Description Language (COCA-ADL). The final step of the COSD is to generate the application code and the COCA-ADL XML file. The COCA-ADL provides a description of the components, connectors, and the architecture's configuration, COCA-ADL is a platform-independent model that can be transformed by a model-to-model transformation tool into the desired platform-specific model. This provides code mobility for the same application into various deployment platforms and provides a runtime model of the self-adaptive application. However, the use of



Figure 2: Context-Oriented Software Architecture.

COSD for building self-adaptive applications for indoor wayfinding for individuals with cognitive impairments was proposed in (Magableh and Barrett, 2011). Evaluating the COSD productivity among the development cost and effort using Constructive Cost Model II (COCOMO II) (Boehm et al., 2000) was demonstrated in (Magableh and Barrett, 2012b). This article focuses on demonstrating the design principles and implementation of the COSM-middleware, supported by an empirical evaluation of its performance and adaptability.

The COSM-middleware offers a self-adaptive application a runtime support for adjusting the application's behaviour dynamically. Figure 2 shows the context-oriented software architecture. The platform is layered into four major layers. Each layer provides an abstraction of the underlying technology. Each layer is platform independent of any given technology. The first layer represents the context-aware application. It provides the user with GUI, functional properties, and non-functional properties (i.e. the map view). The second layer in the platform represents the COSM-middleware. The COSM-middleware subcomponents are shown in Figure 2. The OS sensor retrieves information about the operating systems. Function calls are used to retrieve information about CPU, memory, and disk space (Magableh and Barrett, 2009). The third layer represents the resources and services available in the execution platform, and the core services layer found in the O.S kernel.

3.1. Context Manager

The first component of the COSM-middleware is the context manager, as shown in Figure 2. The context manager gathers and detects context information from the sensors. If the context is changed, the context manager notifies the adaptation manager and the observers (i.e. the context-oriented components) about the changes. Each context-oriented component is designed to be an observer for one or more context entities. This type of interaction is called context binding.

The observer pattern reduces the tight coupling between the context provider (e.g. battery and memory) and the context consumer, (e.g. the context-oriented component). In addition, it enables the middleware to identify which context-oriented component has to be manipulated in response to context changes. Figure 3 demonstrates the

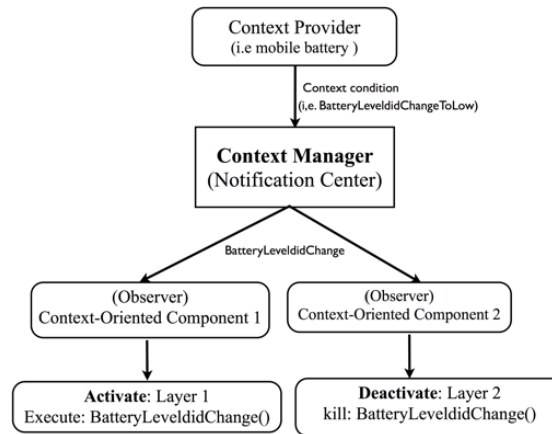


Figure 3: Observer Design Pattern

observer design pattern with one context entity and two observers. At runtime, both context-oriented components register themselves as observers for the context entity (e.g. *BatteryLevelDidChange*) by sending a registration request to the notification centre. The context-change event is sent to the notification centre queue instead of the context-oriented component, then the notification centre broadcasts the context changes, and only the registered component receives the notification. The context-oriented components 1 and 2 have registered as an observer for the context entity. Whenever the context changes, both components 1 and 2 are notified by the notification centre and execute their embedded layer that may implements the methods (e.g. *BatteryLevelDidChange*). In this way, the adaptation manager can identify both components 1 and 2 to be included in the adaptation action, which embeds a subdivision of their implementation by de/activating the associated sub-layer. For example, the adaptation manager can activate layer 1 and executes the method (*BatteryLevelDidChange()*) implemented in the first component, and deactivate layer 2 in the second component, which kills the execution of the method (*BatteryLevelDidChange()*) as shown in the Figure 3.

Supporting context-binding mechanisms with observer pattern provides a clear separation between the context provider and consumer. In addition, it classifies the components implementation based on the execution context conditions. This makes identifying which component must respond to a specific context condition an easy task in the design phase and runtime. However, to achieve this integration, the developers have to consider the following two aspects in the application design: how to notify the adaptation manager about context changes by identifying which components need to observe a specific context condition, and how the adaptation manager can identify the internal parts of the component that need to respond to these changes by clearly stating which method implements the needed behaviour for a specific condition.

3.2. Component Manager

The component manager performs three major functions in the middleware: it searches for a context-oriented component in the component repository, adds components from the repository, and provides context-oriented component instantiation. The component manager achieves the intercession operation by adding a component, or a component sub-layer, to the application structure. To add a component, the adaptation manager asks the component manager to instantiate a specific component. The component manager performs several inspections of the application components through the operation time of the software. Further details about the implementation of the component manager are provided in Section 3.2.

3.3. Policy Manager

The COSD provides the developers with the ability to specify the adaptation goals, actions, and causes associated with several context conditions using a policy-based framework. For each context-oriented component, the developers can embed one or more Decision PoLicys (DPLs) that specify the architecture properties including (quality attributes,

constraints and resources limits). At the design phase, the software developers use a state-machine model to describe the DPL by specifying a set of internal and external variables and conditional rules. The rules determine the true action or else an action based on the variable values. The action part of the state diagrams usually involves invoking one or more of the component's layers. A single layer is activated if a specific context condition is found, or deactivated if the condition is not found (Anthony et al., 2009). The policy manager uses the DPL objects to store policies in the policy repository. The DPL is stored in the policy repository, which conforms to the Associative Storage design pattern (Buck and Yacktman, 2010). This pattern organises the policies into data and keys; this reduces the computation overhead from processing them at runtime.

3.4. Decomposition Manager

The decomposition manager operates as an XML parser for the COCA-ADL, and a constructor that constructs the application's components graph. At development time, the application's models are transformed into a COCA-ADL XML file. At runtime, the decomposition manager reads the COCA-ADL XML file, and adds the architecture instances, including the components, connectors, and configuration, to the application graph. In other words, the decomposition managers constructs a runtime model of the software architecture as it is described by the COCA-ADL. This runtime model is manipulated by the adaptation manager during the adaptation process and It will describe the new adapted architecture of the software.

3.5. Adaptation Manager

The adaptation manager starts the adaptation process after receiving the notifications that identified the context changes and the context-oriented components that observed the notification. The first function of the adaptation manager is to produce the composition plan. The composition plan recursively describes the composite components and the connections between them by describing several connectors and interfaces. To construct a composition plan for the new adapted software architecture, the following information is needed by the adaptation manager. 1) A component graph: The decomposition manager generates the component graph after parsing the COCA-ADL XML file. 2) Decision PoLicy (DPL): The DPL rules determine the true action or the else action based on the values of its variables. 3) Runtime structure style: When several context conditions are found at the same time, the DPL proposes a runtime instance of a design pattern, which may imply combination of multiple components or their internal parts to fulfil the execution context. The structure style describes a structure modification combining a set of the component's layers.

The adaptation process starts the adaptation action, including two types of composition mechanism: internal composition and external composition. In internal composition, the adaptation manager switches a component's layers on or off, based on the composition plan, using the delegation and decorator patterns. The decorator pattern can be used to extend, decorate, the functionality of a certain object at run-time, independently of other instances of the same component. The decorator pattern is an alternative to subclassing. Subclassing adds behaviour at compile time, and the change affects all instances of the original class; decorating can provide new behaviour at run-time for individual objects. In internal composition, the adaptation manager introspects the application's graph and achieves the self-tuning attribute of the self-adaptive software, self-tuning is achieved by redirecting the execution into the delegate object, which activates the desired sub-layer implementation.

In external composition, the adaptation manger adds or replaces components from the application structure, based on the composition plan. The decomposition manager builds the application graph by reading the COCA-ADL. In external composition, a context-oriented component is loaded into the application. This requires the adaptation manager to conform to the bundle pattern (Buck and Yacktman, 2010). The bundle pattern achieves the following goals: 1) Keep executable code and related resources together even when there are multiple versions and multiple files involved in the underlying storage. 2) Implement a flexible plug-in mechanism that enables dynamic loading of executable code and resources. In addition, the invocation design pattern is used to provide a means of capturing runtime messages so that they can be stored, rerouted, or treated and manipulated according to the context state, and allows new messages to be constructed and sent at runtime without requiring code recompilation process (Buck and Yacktman, 2010). For example, when a component receives a message, a method implementation is usually invoked to handle the message. However, this is not always the case. As an illustration, if a component does not implement a particular method, then there is no method, which can be invoked and a runtime exception is raised instead. Because

of the invocation design pattern, it is possible for a message to be delayed, rerouted to other components, or even ignored at runtime without re-compiling the application's code.

In addition, the adaptor design pattern lets components work together, even if they have incompatible interfaces (Buck and Yackman, 2010). Assume a base-component needs to communicate with a context-oriented component, but its interfaces make that unachievable. To solve this problem, the context-oriented component applies to the delegate pattern by defining a protocol, which is essentially a series of method declarations unassociated with the component. The base-component then adapts the protocol and confirms this by implementing one or more of the protocol's methods. The protocol may have mandatory or optional methods. The base-component can then send a message to the protocol interface, which redirects the message to the context-oriented component. At this stage, the adaptation manager can verify whether the context-oriented component can respond to the message call, before invoking the message call by adapting the chain of responsibility pattern. This pattern verifies whether the component can respond to the method call using the responder pattern, which avoids coupling between the sender of a request and its receiver by giving more than one context-oriented component sub-layers a chance to handle the request.

3.6. Verification Manager

As long as the COSM-middleware is aware of the architecture configuration, which is supported by the COCA-ADL configuration element. The COSM-middleware can anticipate the associated configuration with specific context changes. In each Decision Point (DP), the COSM-middleware transforms the software from $state_i$ into $state_{i+1}$, considering the properties of the self-adaptive software. These properties include the following: 1) The set of DPLs attached to the context-oriented components that participate in the adaptation; 2) the architecture configuration elements in the COCA-ADL, which includes the description of the DPLs and the architecture properties, the DPLs specify the external and internal variables that are evolving through the adaptation process; and 3) the adaptation goals, actions, and rules specified by the DPLs.

4. The Context-Oriented Software Middleware Runtime Platform

The final step of the COSD is to generate the code and the COCA-ADL XML file. The runtime functions start once these two inputs are in place. The major component in the COSM-middleware is the adaptation manager. On a broader scale, the adaptation manager defers as many decisions as it can from compile time and link time to runtime. Whenever possible, it performs actions dynamically and executes the compiled code. Handling the context-oriented component framework, the composition plan, the application singleton, and the way in which several components interact with the runtime system is the focus of the following sections. Figure 4 shows a class diagram for the COSM-middleware. Both the map application and COSM-middleware were implemented in IOS mobile operating system for iPhone devices (Apple iPhone Operating System IOS, 2011). For this reason, Objective-C is used to describe the method implementation of the COSM-middleware components.

The runtime platform is a dynamic shared library with a public interface, consisting of a set of functions and a data structure in the header file, located within the context-oriented framework. Many of these functions allow the application to perform the adaptation actions through the adaptation manager. The following sections describe the implementation of the COSM-middleware components, each components is described in terms of the main design principles we used to implement it.

The context-oriented runtime platform starts once it has the compiled code for the application base-components and the context-oriented components plus the COCA-ADL XML file. When the application is launched, the COSM-middleware components are executed first. The adaptation manager then calls the decomposition manager to build the application composition graph and the inheritance tree. The decomposition manager parses the COCA-ADL XML file for the component elements. The decomposition manager adds the components to the graph and the component repository. Each graph node has a component dispatch table. This table has entries that associate method selectors with the component-specific addresses of the methods they identify. In the same way, the decision policies are attached to the associated context-oriented component and added to the policy repository. Figure 5 shows a decomposition mechanism. Each component and its subdivisions are added to a graph in the buffer.

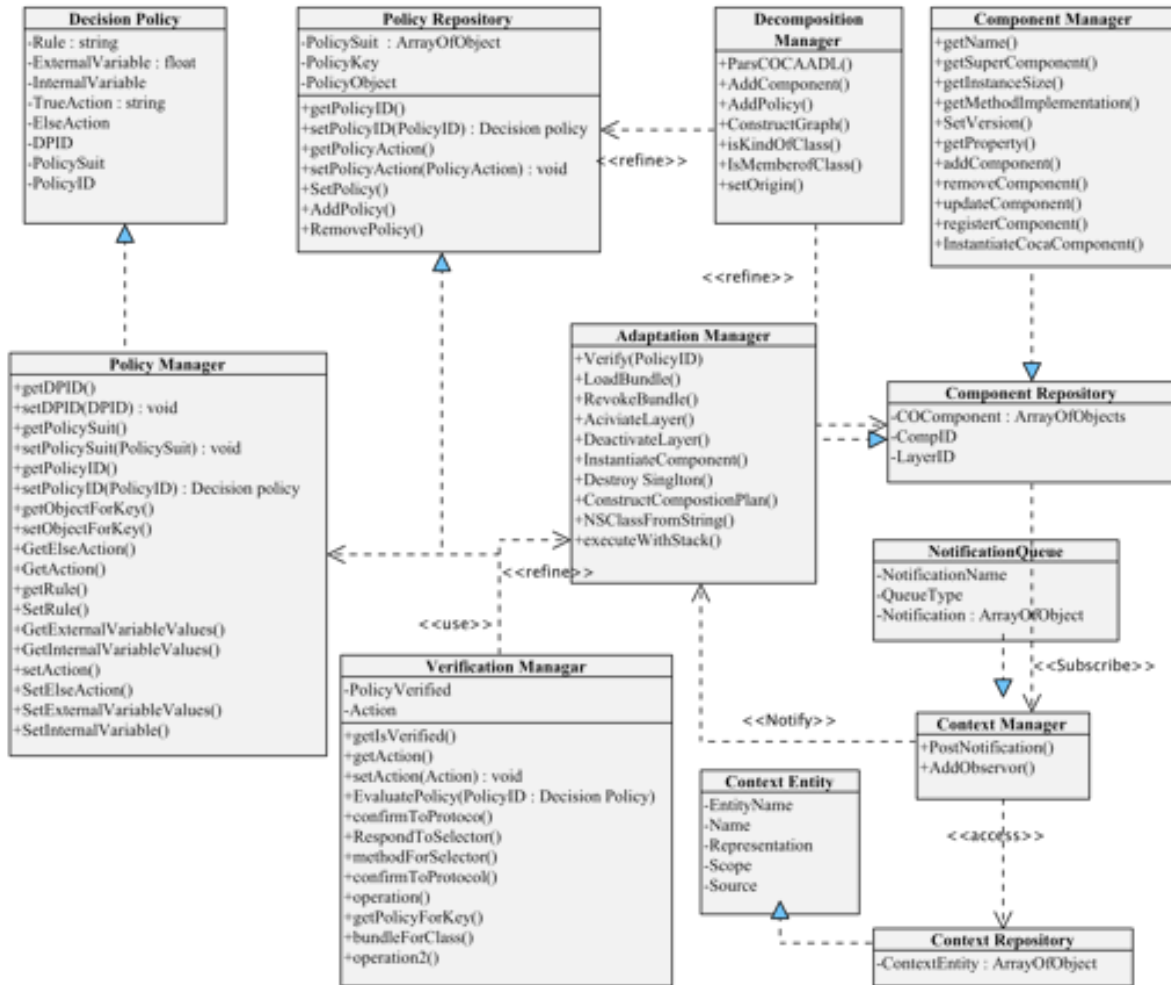


Figure 4: The Context-Oriented Middleware Design

4.1. Adaptation Manager Runtime Functions

Once the decomposition is finished, the adaptation manager asks the context manager for the context state. At the same time, the adaptation manager runs the component instance for the base-component type. The adaptation manager checks each class by parsing the graph. In each node, the adaptation manager performs the following operations: 1) create the application singleton, 2) add the base-components to the singleton, and 3) construct the primary composition plan.

Afterwards, the context manager notifies the adaptation manager about the context state. Based on the context state, the adaptation manager reads the description of the component from the dispatch table. It confirms whether the component has the right objects, and methods which suit the context state. This is accomplished by asking the object to identify its class using *isKindOfClass* and *isMemberOfClass*. This verifies an object's position in the inheritance tree. The COSM-middleware design conforms to the delegation design pattern. This requires each context-oriented component to define a protocol or a formal interface. During the composition, the adaptation manager identifies whether a specific component does conform to a protocol by calling the method (*conformsToProtocol:*), which indicates whether an object claims to implement the methods in a specific protocol, then the operation (*RespondToSelector:*) is performed, which indicates whether an object can accept a particular message. After that, the adaptation performs (*methodToSelector:*), which provides the address of a method's implementation. These methods enable the

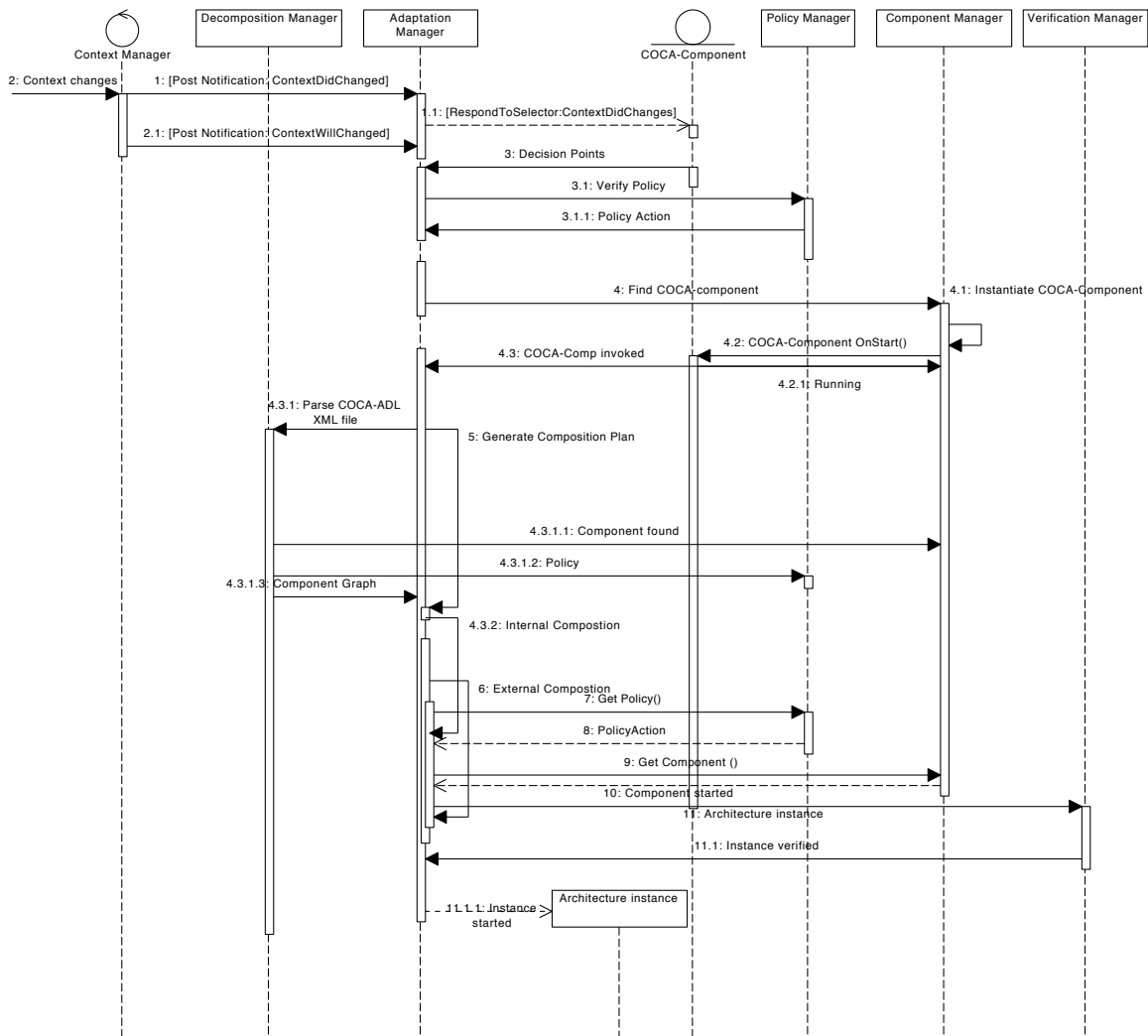


Figure 5: The Context-Oriented Middleware Adaptation Runtime Model

adaptation manager to introspect the application structure. The class diagram in Figure 4 shows the relation between the adaptation manager class and the other COSM-middleware components.

Parsing the COCA-ADL XML file by the adaptation manager to construct the graph has some drawbacks with respect to device performance. A reasonable approach to parsing the XML file with less impact on the quality attributes is the use of the Flyweight design pattern (Buck and Yacktman, 2010). The Flyweight pattern minimises the amount of memory and/or processor overheads required to use objects (Buck and Yacktman, 2010). The Flyweight pattern enables instance sharing, to reduce the number of instances needed, while preserving the advantages of using objects. Classes that implement the Flyweight pattern are called ‘flyweights’. Flyweights encapsulate non-object data so that the data can be used in contexts where objects are required. Flyweights reduce storage requirements when a large number of instances are needed. Flyweights act as stand-ins for other objects (Buck and Yacktman, 2010).

In addition to Flyweight, another pattern that can be used during implementation is the Associative Storage pattern. The most important feature of this pattern is the efficient storage of arbitrary data associated with objects; this promotes flexibility by delaying the selection of which data to access until runtime.

The *MutableDictionary* data structure is used to implement the composition plan and the decision policies (Apple iPhone Operating System IOS, 2011). The use of *XMLParser* implements an event-driven approach with a delegate

object implementing methods for handling each of the ‘events’ the parser encounters during its single pass over the XML data (Apple iPhone Operating System IOS, 2011). Events most commonly of interest are the beginning and ending of ADL elements and attribute data within elements. The *XMLParser* reads the XML elements, then uses *MutableDictionary* to store them in the dictionary. The *setObject:ForKey:* method is used to create new associations in the dictionary. When keys and values are added and removed from a mutable dictionary, the memory allocated for storing objects grows and shrinks automatically. If *setObject:ForKey:* is called with a key which is already in the dictionary, the object associated with that key is replaced by the new object. Each key is stored at most once (Apple iPhone Operating System IOS, 2011).

After completing the composition plan, the adaptation manager implements the dynamic creation pattern to load and execute the application’s components. Once the composition plan is completed, the adaptation manager introspects the application’s structure. Redirecting the context-oriented component delegate to the desired layer activates the component sub-layers. In some cases, a context-oriented component is loaded into the application. This requires the adaptation manager to employ the Bundle and Invocation design patterns (Buck and Yacktman, 2010). The *forwardInvocation:* method is used to give a default response to the message, or to avoid the error in some other way. For example, suppose that the adaptation manager receives a message call for a method *memoryLevelDidChange*. First the verification manager verifies whether the receiver object can respond to this message using *respondToSelector* (Apple iPhone Operating System IOS, 2011). When the object cannot respond to the message because it does not have a method matching the selector message, the runtime system informs the object by sending it a (*forward Invocation:*) message. Every object inherits the method (*forward Invocation:*) from the super-class context-oriented component.

However, the object version of the method simply invokes *doesNotRecogniseSelector*. In this case, the adaptation manager forwards to other objects. First, the adaptation manager determines where the message should go and sends the message with its original arguments. The message can be sent with the *invokeWithTarget:* method. If the invocation has failed in the desired sub-layer, the method forwards the invocation to the context-oriented component, which forwards it into its sub-layers until one of the sub-layers responds to it. If the sub-layers do not respond to it, the adaptation manager introspects the component graph and reconstructs the composition plan. Once an object found in the distributed environment in a remote component.

The adaptation manager performs “invocation” by obtaining the method signature and the selector (Apple iPhone Operating System IOS, 2011). When a message is sent to an object that does not implement it, the actual implementation assumes that the stack frame for the arguments of the method already exists. All further changes to the method’s arguments using Invocation’s methods are performed on that stack frame. After the method invocation returns, the adaptation manager can access the returned value and possibly change it, using the (*getReturnValue:*) and (*setReturnValue:*) methods.

4.2. Policy Manager Implementation

As illustrated in Figure 4, the policy manager uses the decision policy objects to store policies in the policy repository. The policy dictionary stores each policy in loosely coupled objects, which are accessed through the method *getObjectForKey*. Once the object is retrieved, the policy manager obtains the policy actions, attributes, rules, external variables, and internal variables. Then it passes them back into the verification manager. The verification manager evaluates the current value for the variables among the predefined values in the policy syntax.

Whenever the application execution reaches decision points and/or the context manager has notified the adaptation manager of a context change, the decision points must be executed to advise the application of pre- or post-actions among specific notifications. The manager implements the necessary methods to manipulate the decision policy syntax. The policies are stored in an array of objects. Each object is accessed through the method *getPolicyForKey*. The key refers to the policy ID that is attached to every context-oriented component. In the same way, the policy manager is used to upgrade the policy by calling the methods *setObjectForKey*, *SetPolicySuit*, *SetRule*, *setAction*, and *setElseAction*.

The decision policy is stored in the policy repository, which conforms to the Associative Storage design pattern. A binary representation for each policy is stored in the *MutableDictionary* data structure, where the key value is used to access the desired policy. The method *addPolicy* is used to add a new policy to the repository. In the same way, policies can be removed using the method *RemovePolicy*. Once the policy manager updates the policy, the method *setPolicy* is used to update the policy syntax in the repository. The policy syntax is retrieved through the method

getPolicyForKey. This implementation of the Associative Storage design pattern reduces the computation overhead for retrieving the policies and evaluating them.

4.3. Verification Manager Implementation

A context-aware application is self-configurable if it is able to adapt autonomously to changing environmental conditions or internal status by altering its structures, behaviours, and data to meet its functionality and quality requirements. From a middleware perspective, such a feature relies on the following key characteristics: A) The middleware’s ability to monitor and define its internal status and external conditions (e.g. application modes, CPU and memory use, and attachment of external devices); B) its built-in knowledge of configuration variability and related policies/rules for deciding and planning changes; and C) its ability to perform dynamic configuration changes without violating the constraints relating to overall system functionality, performance, and dependability.

The Flyweight pattern is used to guarantee that the verification process does not affect the quality attributes. This is accomplished by adapting the feature of instantiation in the Flyweight. Moreover, if there are many external and internal variables in the decision policy, all these variables will be instantiated once and share this instance to multiple values. In addition, the Flyweight acts as a temporary place holder for other more heavyweight objects.

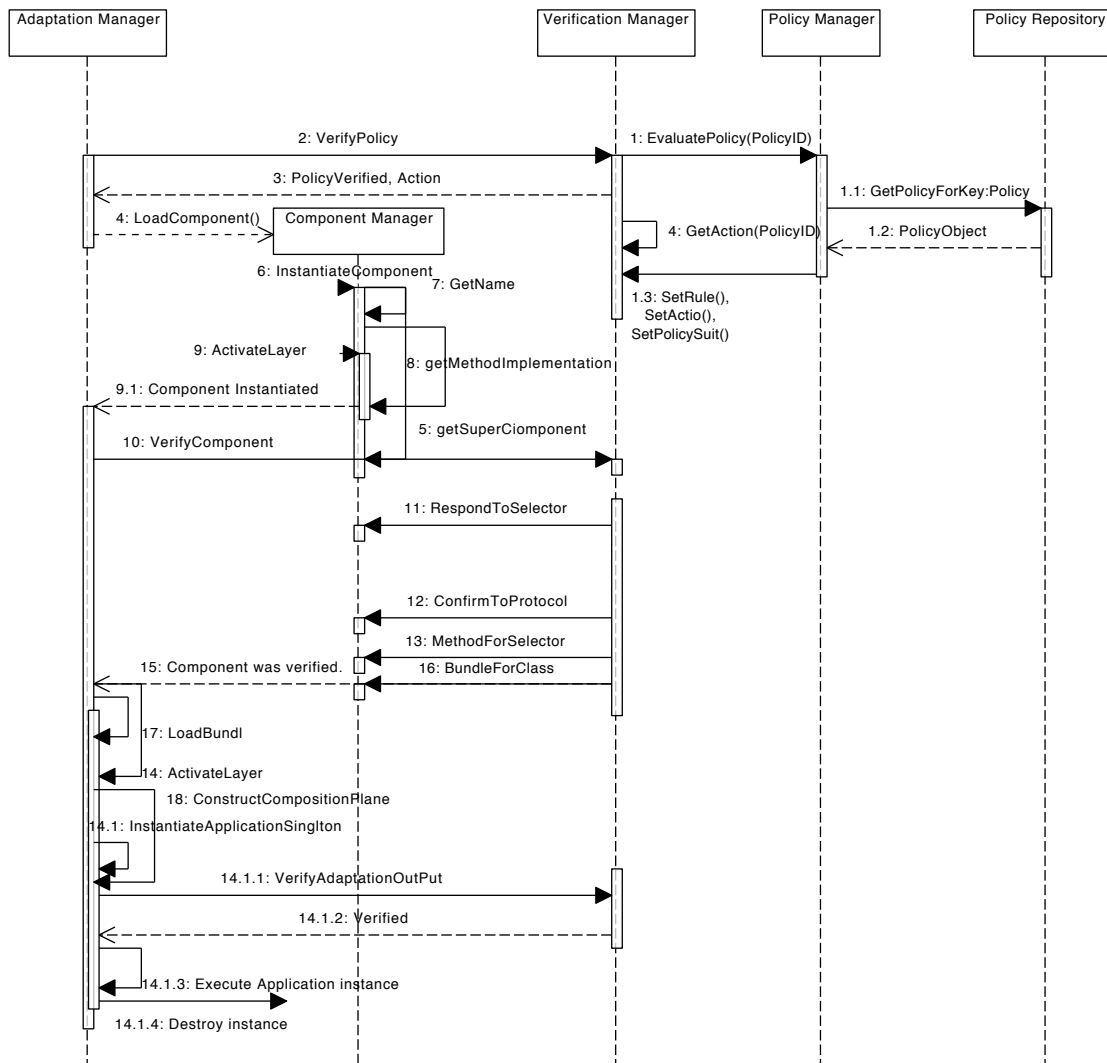


Figure 6: Adaptation self-assurance verification

Figure 6 shows a sequence diagram of the self-assurance verification. The verification manager evaluates the policies by calling the *verifyPolicy:(NSInteger) PolicyID* method. This method asks the policy manager to retrieve the stored policy by its key. The policy manager searches the array of objects for the specific policy ID. The evaluation result, which contains the proposed action to be performed, is passed back to the adaptation manager. Afterwards, the adaptation manager locates the desired component and/or sub-layers which need to be executed. Then it asks the verification manager to verify them using the methods *methodForSelector*, *confirmToProtocol*, *RespondToSelector*. Once the verification has been accomplished, the verification manager sets the boolean variable *PolicyVerified* to be true.

The motivation behind the use of selectors is to postpone specifying the message that will be sent to an object until runtime. This reduces coupling between objects by limiting the information that message senders need about the message sent. In the same way, whenever the adaptation manager needs to verify the implementation of a context-oriented component or a subdivision, the selector will be used to determine whether the object has the proposed method. The verification manager can load bundles/components by executing the method *bundle = [NSBundle bundleWithPath:theBundlePath]*. However, after loading the bundle that contains a context-oriented component, the verification manager testifies its ability to respond to the desired method call using *bundleForClass* and *respondToSelector* methods.

5. Context-Oriented Software Evaluation

This section focuses on evaluating the performance and modifiability quality attributes of context-oriented software, including the COSM-middleware and the case study implementation. Before evaluating the performance of the COSM-middleware it is desirable to evaluate the modularisation strategy of context-oriented component to find out if there is any impact of such modular structure over the software self-adaptability and dependability. We do not know the impact of modularisation technique over software performance. The general belief is that using different decomposition mechanisms will not affect runtime properties of software. However, this may not apply to self-adaptive software systems. For this reason, the case study application was implemented using Aspect Oriented Software Development (AOSD) (Filman et al., 2004) and COSD, as they adapt different decomposition techniques for supporting software with adaptability. Their ultimate goal is to support the adaptability and variability of software systems, and to be able to reduce development cost and effort, while improving the software modularity and complexity. This motivates us to evaluate these technologies with respect to their ability to support software adaptability (modifiability) and the performance gain from using these technologies to implement the case study application in a mobile computing environment. This article claims that COSD and the COSM-middleware are better suited to build self-adaptive software in the mobile computing domain. To this end, an evaluation of the two major paradigms (AOSD and COSD) is required to find out which one is better suited for building self-adaptive applications.

5.1. Metrics

In the first experiment, the case study application eCampus was implemented as a real iPhone application using the COSD and AOSD paradigms. The second experiments compared the performance of the COSM-middleware implementation with several frameworks and middleware architectures including JCOP (Schuster et al., 2011), JCOOL (Sindico and Grassi, 2009), MUSIC (Geihs et al., 2011), and MADAM (Mikalsen et al., 2006). Then the following questions were analysed. First, how expensive is it to perform context monitoring? Secondly, what is the effect on the allocated resources after the software has performed context detection, particularly when multiple heterogeneous events are detected? Thirdly, what is the performance gain of activating and executing multiple and collaborated aspects in comparison with context-oriented components composition, in response to multiple context events arriving at the same time. The following experiments focus on evaluating each paradigm implementation of the eCampus application to support adaptability and dependability, based on the following criteria.

1. **Battery Usage.** This criterion evaluates the device's battery durability while running the eCampus application and performing the adaptation processes, including context monitoring, detecting, decision-making, and adaptation. The iPhone Energy Diagnostic tool provides a relative energy usage on a scale of 0 to 20. These values explain how expensive it is with respect to the battery life to run a specific process over the execution time.

2. **CPU Activity.** This criterion analyses the CPU activities and CPU time required for performing the adaptation processes, including context monitoring, detecting, decision-making, and adaptation. This includes the time required for components/aspects composition in response to multiple and heterogeneous context change events.
3. **Real Memory Allocation.** This criterion measures the amount of memory allocated by the application during the execution of particular functionalities, including context monitoring, context detection, and adaptation.
4. **Sleep/Wake.** This parameter captures the eCampus application's ability to adjust its activity while the device is running in sleep mode. Normally, if the application keeps running in the background and performs some kind of operation, for example, updating the current location while the device is in sleep mode, the allocated resources are intensively degraded. This feature evaluates the architecture's ability to adjust the application behaviour, while considering the interoperability between the middleware functionality and the allocated resources.
5. **Adaptation/reconfiguration time.** This criterion captures the required time for the application to adapt its structure and behaviour by adding, removing, or updating components/services. The adaptation time was measured from the start to the end of the adaptation action. The reconfiguration time measures the time required by the middleware to load and execute the plug-in (bundle) implementation.

5.2. Hardware and Software Configuration

The CPU activity, CPU time, real memory allocation, and energy usage are measured for performing each adaptation process separately. These values were measured using the energy diagnostics and activity-monitoring tools, which analyses the running application on the iPhone device (Apple iPhone Operating System IOS, 2011). The Energy Diagnostic tool was used to measure the battery while the device was not connected to an external power supply; after the experiment was finished, the data were imported from the iPhone and then analysed. For measuring the CPU activity, CPU time and the real allocated memory, the eCampus application was executed on the same iPhone devices for each paradigm/framework implementation separately. The instrument tool was executed on Macbook pro, which traces the data from the iPhone device using the activity-monitoring tool. The iPhone was connected wirelessly with the activity-monitoring tool. This allows the tool to capture more accurate data for the energy usage and the CPU activity with respect to each process under evaluation. To testify variation in the application behaviour, a simulator was included with each eCampus implementation. The simulator is used to allow the user to simulate specific context changes, which are used to test the application's ability to adapt the desired behaviour. For each adaptation process, the experiment was established as follows:

Context monitoring. The CPU activity, CPU time, real memory allocation, and energy usage are measured for performing context monitoring at two different time intervals. First, when the application "did Finish Launching With Options" and the UI views did loaded. Second, the simulator interface was used to trigger the context monitoring process, this operation excluded any events related to the application load time. In addition, it allowed us to estimate the time required to process 10 contextual events enqueued at the same time. The simulator generates these events generally in First In First Out (FIFO) order. This experiment was executed 200 times, then the variance and standard deviation were calculated for the above criteria.

Context detection. The CPU activity, CPU time, real memory allocation, and energy usage are measured for performing context detection with the aid of the simulator. When the context detection button is pressed, the simulator generates and en-queue multiple events, which are executed in First In First Out (FIFO) order. This allows the experiments to evaluate the required time to process these context events, and the time required to perform the reasoning action (i.e. decision-making) by the application. This experiment was executed 200 times, then the variance and standard deviation were calculated for the above criteria.

Adaptation time/re-configuration time. The CPU activity, CPU time, real memory allocation, and energy usage are measured for performing the adaptation/re-configuration in two modes. In the first mode, the application was executed for the same period of time (five hours). Then, the adaptation time was measured once the adaptation action was started until finished, the CPU time was taken from the activity-monitoring tool. In the second mood, the simulator was used to generate multiple context events, that measures the application response with regrade to low battery context. This experiment was executed 200 times, then the variance and standard deviation for each value were calculated. This allows us to identify the positive error as described in the following sections.

5.3. COSD Vs. AOSD Experiments

The assumption made by the AOSD community is that dynamic aspect weaving can be used to adjust the software behaviour dynamically, regardless of the complexity involved in implementing Aspect-Oriented Programming (AOP) applications. Existing Dynamic AOP techniques tend to add a substantial overhead in both execution time and code size (Hundt et al., 2010). The eCampus implementation was re-engineered to be integrated with the Objective-C AOP framework, called AspectCOCA proposed in (AspectCOCA, 2011). The AspectCOCA framework offers many benefits to the AOP implementation, as it provides dynamic code weaving without any need to implement special AOP engine like PROSE 2 (Popovici et al., 2002). In addition, AspectCOCA implementation excluded the impact of Java virtual machine from the overall performance of the self-adaptive application, as AspectCOCA does not need a virtual machine to perform dynamic runtime operations including behavioural Intercession and Introspection. This offer great benefits for the application and reduces the development cost. We believe it is appropriate to implement the DAOP-eCampus using this technology, which excludes the impact of aspect engine over the performance and device resources. As a result, we implemented several aspects for handling the context monitoring and detecting processes. In addition, the context-dependent behaviours for the location service, battery level, and the camera flashes were implemented. However, for the location service, there are three nested aspects implemented to provide behavioural variation of the battery level. These aspects are the GPS-based, WiFi-based, and IP-based location services. In COSD, these aspects are implemented using three context-oriented components.

5.3.1. Experiment 1: Context Monitoring and Sensing

This experiment evaluates the processes of context monitoring and environment sensing, based on the above criteria. Specifically, it evaluates how the software uses the allocated resources such as battery, CPU, and memory. In the DAOP approach, context monitoring is handled using separate aspects which span the application's main execution. In COSD, this is handled using the context manager.

Designing aspects that become active when particular contexts are verified requires the possibility of referring to a context definition in a pointcut construction. This means that joinpoints such as `BeInContext(Context Battery-LowCTX)` should be provided by the framework. In addition, the aspects composition needs to keep track of past context conditions and their associated states, which require more CPU activity and memory allocation to perform this functionality. This adds much overhead to the advice execution, because the AOP framework must perform context snapshots through the monitoring and sensing process. The problem behind this is that the context snapshot is made every time the context is changed. This makes the platform storing and processing the context history for multiple events at multiple times. In addition, the AOP framework must transform the context changes into basic entities like joinpoint request. The joinpoints are activated by registering them to the execution monitor. When the execution reaches one of the activated joinpoints, the execution monitor notifies the DAOP engine, which executes the advice method. This implies, that the AOP will evaluate each joinpoint with regard to the passive and active context through the context detection and decision-making processes as shown in the next experiment.

As a result, battery energy is consumed faster in comparison to the implementation based on COSM-middleware, which includes a dedicated context manager supported by a context repository (used to store and process the past context information). In addition, each context-oriented component registers its interest on a specific context change. This makes the context manager sense the environment for a particular set of context information.

The results of the experiment on battery usage is shown in Figure 7. This shows that the context-oriented component uses less battery energy than the DOAP implementation. The COSM-middleware optimises the context-monitoring process by storing and processing the context information that was considered by the context-oriented component registration. Such enhancement of the context monitoring preserves the battery energy by 11.5% of the total energy usage. This value is supported by the evaluation results of the CPU activity shown in Figure 8. For context monitoring, the DOAP-eCampus requires more activity to be executed in comparison to COSD-eCampus; this consumes more battery energy. With regard to memory allocation, the DOAP-eCampus allocated more real memory to execute and perform the context snapshot (i.e. storing and processing the past context) than is needed by COSD-eCampus, as shown in Figure 9. This figure shows how expensive it is to allocate and process the context snapshot in the DOAP-eCampus application.

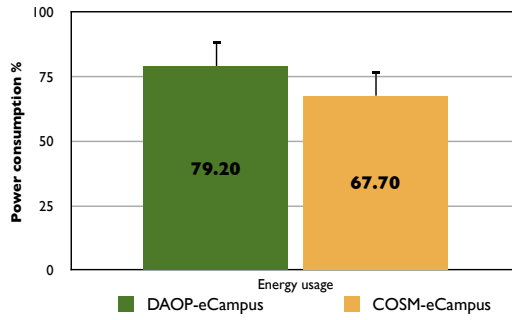


Figure 7: Context Monitoring battery usage

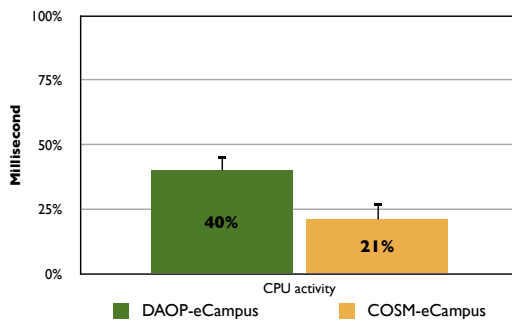


Figure 8: Context Monitoring CPU Activity

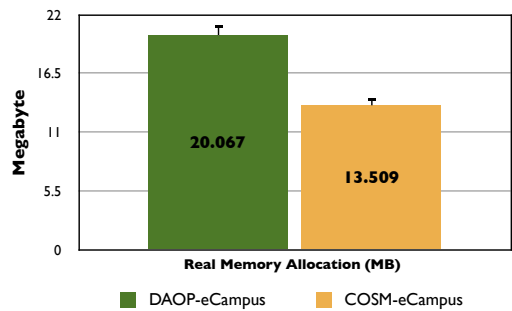


Figure 9: Real Memory Allocation (MB)

5.3.2. Experiment 2: Context Detection

For the context detection process, both implementations were evaluated based on the above criteria. The evaluation results for energy usage are shown in Figure 10. The evaluation results show that DOAP-eCampus consumes more energy to notify the application components about multiple context changes which were detected in short frequency. This requires more CPU activity to process the context changes and evaluate them with the passive context values stored in the joinpoints. The CPU activity for both applications is demonstrated in Figure 11. In addition, the DOAP application requires more memory for allocating the aspect contexts and notifying them because each aspect must be allocated and executed. The AOP framework then notifies the aspects about the context changes. Later, the decision is left to the aspect methods implementation to decide whether to adapt or not. Such implementation of the context detection process using DOAP intensively consumes the allocated resources to notify multiple aspects about multiple events. In some cases, the aspect implementation was independent of the execution context, but it was executed and notified. The real memory allocation for the context detection process is shown in Figure 12.

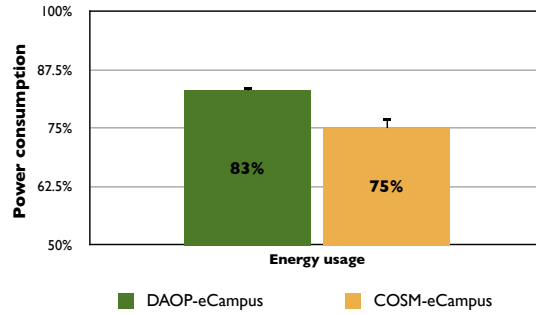


Figure 10: Context Detection Battery Usage

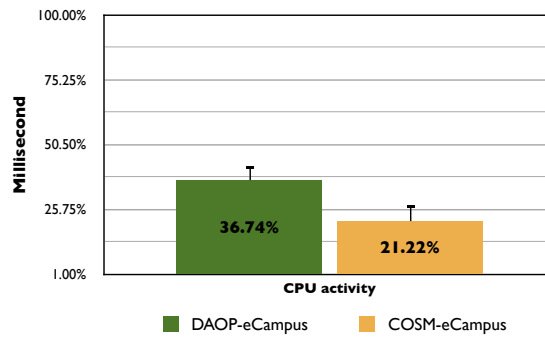


Figure 11: Context Detection CPU Activity

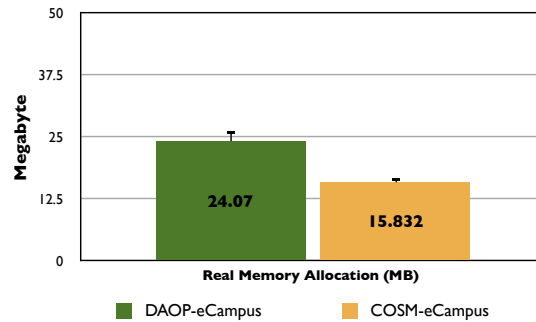


Figure 12: Context Detection Real Memory Allocation (MB)

5.3.3. Experiment 3: Activating Collaborative Aspects

It is claimed that in AOSD, dynamic aspect weaving can inject tangle-free code in the program execution; as explained before, context-dependent behaviours are collaborated aspects entangled with each other. It is claimed that in COSD, context-oriented components can be activated dynamically to adjust the application behaviour, with affordable costs, during the adaptation. Designing context-dependent behaviour using an aspect-oriented programming paradigm requires platform support for activating aspects driven by the context state; such an implementation requires the AOP platform to evaluate each joinpoint in conjunction with the associated context state and the passive context values. In addition, once the decision has been made, the AOP platform must search for the associated method implementation that implements the required context-dependent behaviour. Moreover, from our own experience, it is very complex to decide which aspect should be woven first, because of the implicit dependence among the aspect implementations. For example, the platform should decide when the battery level is low, and which aspects must be activated. On the

other hand, when activating the location aspect, the platform must consider the battery level before deciding which location service to use; such processes provide cyclic dependence among the aspects implementations and lead to unguaranteed adaptation outputs.

Figure 13 shows the battery usage when multiple contextual aspects are activated and executed compared to the composition of multiple context-oriented components. The figure shows that the DAOP-eCampus consumes more energy to perform the adaptation, as it requires more energy to process the context state in each joinpoint. In addition, it requires the AOP framework to resolve the dependence between several aspects before and after the advice methods execution. The CPU activity is shown in Figure 14 and the real memory allocation for performing the activation and execution is shown in Figure 15.

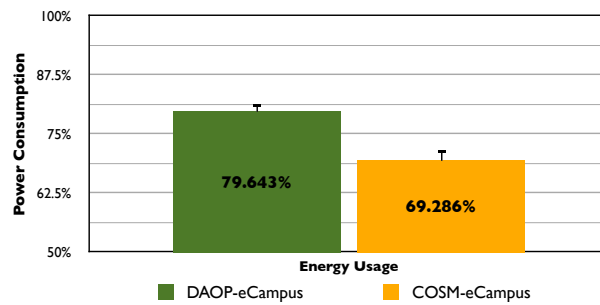


Figure 13: Activating Collaborated Aspects/context-oriented components Battery Usage

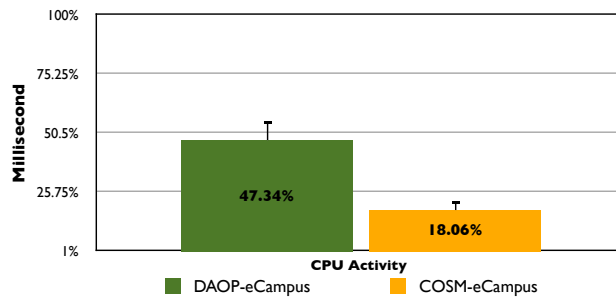


Figure 14: Activating Collaborated Aspects/context-oriented components CPU Activity

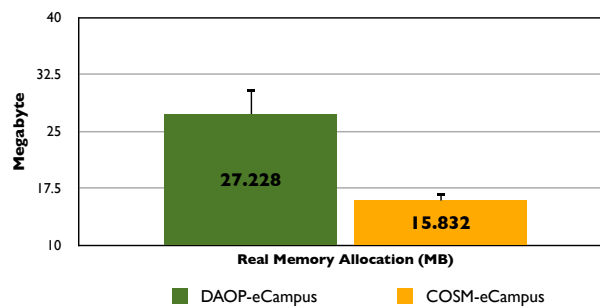


Figure 15: Activating Collaborated Aspects/context-oriented components Real Memory Allocation (MB)

The aspects composition needs to keep track of past context conditions and their associated states; more CPU activity and memory allocation are needed to perform this functionality. This experiment describes how each platform responds to multiple events (i.e. context conditions) detected at the same time. The adaptation/reconfiguration time

for composing aspects/components is shown in Figure 16. The values were taken every 2 min from the Apple Instruments tool while executing the application for 30 min continuously. As shown in Figure 16, the COCA-eCampus requires less CPU time for composing the components, but DOAP requires more time for activating and executing the contextual aspects. The evaluation of aspects activation and execution shows an increased adaptation time because each aspect requires more memory allocation and CPU time to resolve the execution context with the context snapshot (i.e. context history). On the other hand, the COSM-middleware requires more adaptation time for loading and executing the bundle implementation, but it can switch between weak/strong adaptation actions based on the execution context and the allocated resources. As shown in the figure, context-oriented components composition requires less adaptation/reconfiguration, based on the adaptation mechanism. Such variations in the adaptation time provided by COSM-middleware can make use of the adaptation process and increase the device durability. The adaptation time in DAOP, as shown in the figure, may increase over the execution time, which leads to poor performance and lower efficiency.

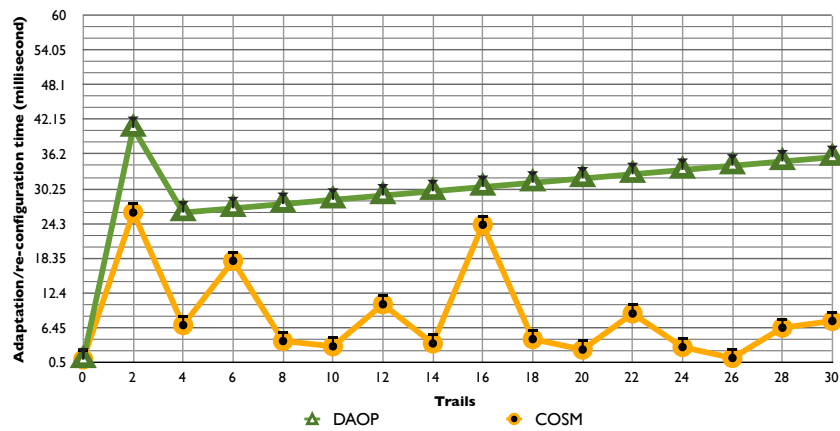


Figure 16: Aspects/context-oriented components Composition

5.4. Context-Oriented Middleware Evaluation

The case study was implemented with COSM-middleware and other approaches proposed in the literature. These approaches include the context-oriented programming paradigm targeting mobile devices, called JCOP (Schuster et al., 2011), JCOOL, supported by CAMEL methodology, which used aspect-oriented programming and middleware for context-dependent behaviours de/activation (Sindico and Grassi, 2009), MUSIC-middleware (Geihs et al., 2011), and MADAM-middleware (Mikalsen et al., 2006), which was fully implemented by Paspallis (Paspallis, 2009). The implementation of the eCampus in JCOP followed the COP approach (Schuster et al., 2011). The implementation in JCOOL was accomplished with the aid of the aspect-oriented programming framework for Objective-C (Aspect-COCA, 2011) and CAMEL framework. Both MUSIC-middleware and MADAM-middleware functionalities were implemented using the MUSIC development paradigm proposed by Rouvoy et al. (Rouvoy et al., 2008).

	JCOP	JCOOL	MUSIC-Middleware	MADAM-middleware	COSM-middleware
Self-tuning	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>
Self-reconfiguring	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 17: Autonomic Properties Support

The objective of this experiment is to evaluate the ability of our COSM-middleware in supporting the autonomic properties, self-tuning and self-configuring, of the self-adaptive eCampus application. The support of these properties in the above-mentioned solutions can be summarised as shown in Table 17. JCOP and JCOOL support only fine-grained adaptations using ad-hoc programming-level techniques; this is not able to change the application structure.

MUSIC-middleware supports both autonomic properties using parametric tuning and plug-in architecture. The implementation of plug-in architecture (i.e. building a software architecture from multiple plug-ins) used for adding or removing services/components at runtime. MADAM-middleware supports only self-configuring property of self-adaptive system, as it models a separate plug-in architecture for each context provider.

5.5. Experiment 4: Self-tuning Evaluation

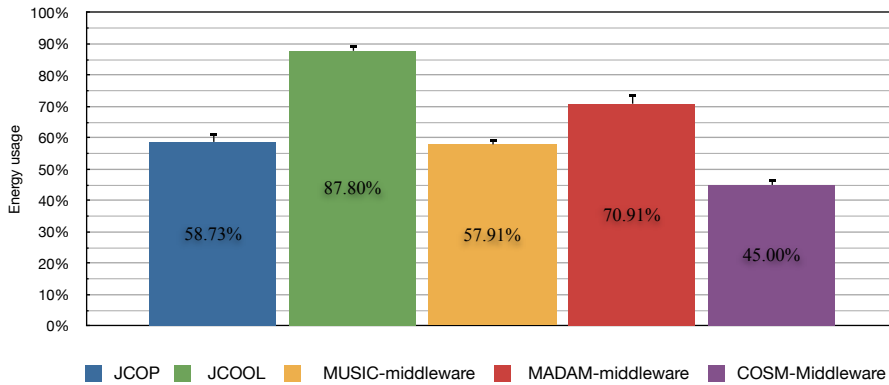


Figure 18: Energy Usage for eCampus Application

Figure 18 shows the experimental results for energy usage analysis for the eCampus running on the five platforms: JCOP, JCOOL, MUSIC-middleware, MADAM-middleware, and COSM-middleware. The experiment shows that the COSD-eCampus implementation of the eCampus application used 15% less battery energy than the MUSIC implementation used. The eCampus implementation with JCOOL consumes more energy during the adaptation processes because it does not consider the battery level or status during the adaptation action. In JCOOL, the context values are only considered for evaluating the context-dependence in each joinpoint implementation. In the same way, the MADAM-middleware drained the battery faster because each location service was implemented in a distinct plug-in (bundle) architecture, which requires more processing time for loading the bundle implementation. In contrast, when the same application was adapted by the COSM-middleware, the application was able to adapt its behaviour and use less energy because the COSM-middleware adapts to the location service by redirecting the delegate object to activate the required service implementation. In JCOP, the application was able to adjust its behaviour and adapt the required location service; unfortunately, the context monitoring was relying on the mobile device operating system to deliver and detect the context information; this leads to faster consumption of the battery resource as a result of trying to process too many context events at the same time. The MUSIC-middleware performs better with respect to battery consumption because MUSIC-middleware uses a fine-tuning mechanism for manipulating components implementation. However, the application implemented using MUSIC-middleware consumes more energy than that using COSM-middleware because MUSIC-middleware calculates the fitness of the application variant using a utility function every time the context state changes. Such verification at runtime requires more CPU time and memory allocation, which, as a result, consumes more energy.

Figure 19 shows the experimental results for CPU activities analysed for the eCampus application in the five platforms previously mentioned. The evaluation considered adaptation processes including context monitoring, detecting, decision-making, and adaptation. As shown in Figure 19, context monitoring requires much more CPU activity in the JCOP and JCOOL platforms as they have no dedicated context-monitoring process, and they rely on the infrastructure to deliver the context information. The MUSIC and MADAM architectures come second with regard to context monitoring as they both implement a dedicated context manager which is able to process and filter the context information. Unfortunately, JCOP, JCOOL, MUSIC, and MADAM do not consider the effect of a contentious and unbalanced monitoring process for the context environment. This implies notifying the application several times about multiple context events, which requires more of the CPU time to process and handle these events. On the other hand, adapting the observer pattern allows the context manager in the COSM-middleware to notify the interested components about the context changes when needed, so that the context-monitoring time drops from 57 ms in MUSIC-middleware

to 33 ms in COSM-middleware. For the same reason, the context detection process drops from 72 ms to 46 ms in COSM-middleware.

The decision-making in JCOP and JCOOL require less CPU activity as both platforms use static decision-making. The JCOP and JCOOL approaches assume that the developers can predict when and where the context-dependent behaviour is needed in the application source code. For the same reason, the MADAM-middleware requires less time for decision-making as it uses predefined rules supported by a rule engine to control the adaptation action. In the MUSIC-middleware, the decision-making process is performed at runtime with the aid of a utility function; this requires more computation activities for analysing the architecture’s constraints, adaptation goals, predefined rules, and the quality of services, plus the user’s preferences. Afterwards, the application’s variations model (adaptation plan) is selected based on the utility function results. In COSM-middleware, the decision-making process is performed at runtime with the aid of the policy and verification manager, which both consider the decision policies in conjunction with the available resources and the quality-of-services (QOS). In general, the dynamic decision-making process requires more effort from the CPU than the static approach does, as shown in Figure 19.

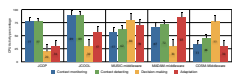


Figure 19: Energy Usage for eCampus Application

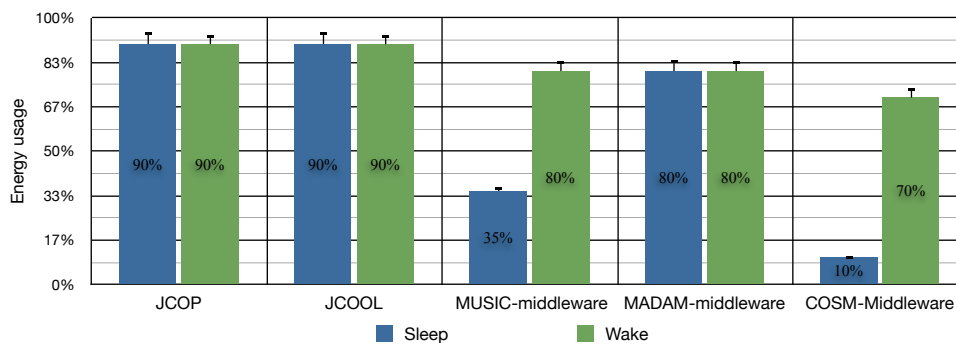


Figure 20: Using the Allocated Resources in Sleep/Wake Mode

Figure 20 shows how the five platforms use the allocated resources while the device is in Sleep/Wake mode. Such an evaluation reflects the middleware’s ability to adjust its own functionality as long as the application is running in the background. At the same time, it shows how the COSM-middleware wakens the application to notify the user about events in the monitored region. The other approaches have not considered the trade-off between the adaptation process and the allocated resources. The result of this is that the eCampus implementations on JCOP, JCOOL, and MADAM were consuming battery life even when the application was in sleep mode or running in the background. The applications keep updating the current location of the device. Such an action was unnecessary, as the device was in sleep mode, so they consumed 90% of the battery life after executing the application for 5 h. With the context-oriented middleware and MUSIC-middleware, the application was executed for the same period of time, and consumed less battery energy.

5.6. Experiment 5: Self-reconfiguring Evaluation

As mentioned before, self-reconfiguring is the capability of the software to adapt and behave autonomously in response to context changes. To evaluate this attribute, we considered the three versions: the MUSIC-middleware, MADAM-middleware, and COSM-middleware. The JCOP and JCOOL platforms were excluded from this experiment as they did not support architecture reconfiguration.

The adaptation/configuration time for adapting the eCampus application as mentioned before. The scenarios include adding a suitable location service according to the battery level, adding a location service component, and

adapting to a web mapping service. Figure 21 shows the evaluation results for the three architectures. The MADAM-middleware requires more time to perform the adaptation because three bundles are loaded and executed. The total adaptation time was 210 ms. The MUSIC-middleware required less time for reconfiguring the software as it adapted the location service using parametric tuning rather than loading a complete bundle for it. The COSM-middleware comes first in the analysis, as it can switch autonomously between several location services by de/activating the associated layers. In addition, it verifies whether the plug-in can provide the necessary services before physically executing its implementation. In addition to this, according to the context state, the COSM-middleware performs a runtime composition of the software components so that only the needed components are executed. In MUSIC, the adaptation takes longer as it evaluates multiple application variations, then one variant is selected and executed. Loading a precompiled code from the component repository after instantiation (as done in the COSM-middleware implementation) is accomplished in a shorter time than it takes to load the whole bundle implementation at once in the MUSIC-middleware. This is illustrated in Figure 22, which shows the memory allocations for the three platforms. It is worth mentioning here that when the battery level is low, the COSM-middleware allocates less memory because of the size of the context-oriented component, which is small compared to the bundle implementation in the MUSIC- and MADAM-middlewares.

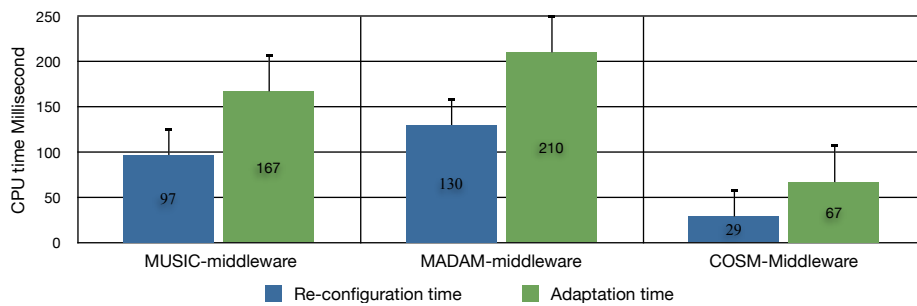


Figure 21: Adaptation/Reconfiguration Time (ms)

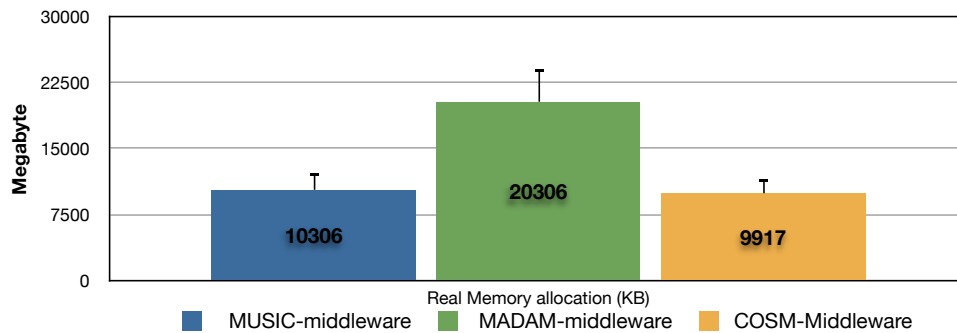


Figure 22: Memory Allocation for eCampus

6. Related Work

Supporting the development and execution of self-adaptive software systems raises numerous challenges. These challenges include the development processes for building them, the design space, which describes the design patterns and the best practices of designing their building blocks, (i.e. component model or code fragments), and the adaptation mechanism, which describes the best adaptation action that can be used under the limited resources of the execution environment. The proposed approaches in the literature can be classified into model-centric, middleware-centric, and programming-level techniques. The ultimate goal of these approaches is to support adaptability, variability and

increase the software quality by managing the context-dependent functionality at the programming level, middleware layer, or architecture model. In addition to that, they try to provide an adaptation mechanism, that have less impact on the allocated resources under the mobility constrains of the execution environments.

In self-adaptive applications, the selection of a particular component or code block at runtime is presumably made based on the active or passive context information plus the context state and its dependency (Lincke et al., 2011), and the possible composition of the context-dependent parts (Hirschfeld et al., 2008), which will exhibit volatile behaviour in the face of context changes. One could ask whether the current software domain techniques have sufficient support or suitable mechanism for performing such dynamic selection and composition of a software component based on its context-dependent functionality. Behavioural composition and re-configuration concerns require the software modules to be loosely coupled, and their behaviour variations can be combined and activated autonomously, according to the context changes. Such a challenge has been tackled by a composition strategy performed through the development process and depending totally on a static view of the self-adaptive software design. Such a view implies that the developers have to explicitly predict the final composition of the software and possible variations of the application using a programming-level technique such as Context-Oriented Programming (COP) (Schuster et al., 2011) and Aspect-Oriented Programming (AOP) (Filman et al., 2004).

COP enables context-dependent adaptation and dynamic behaviour variations (Gassanenko, 1998; Hirschfeld et al., 2008). In COP, context can be handled directly at the code level by enriching the business logic of an application with code fragments responsible for performing context manipulation, thus providing the application code with the required adaptive behaviour (Salehie and Tahvildari, 2009). Costanza et al. (Costanza et al., 2006) proposed the design of context-aware systems following a layered approach. The term "layer" refers to a specific context-dependent functionality, which might include a partial implementation of a class or a set of methods (Costanza et al., 2006). Alternatively, the whole class is encapsulated inside a layer (Hirschfeld et al., 2008). Hirschfeld et al. argued that the class-in-layer approach is more effective than the layer-in-class approach for encapsulating the context-dependent functionality, starting from the claim that context-dependent behavioural variations occur separately or in any combination, and in most cases they are collaborating and entangled with each other. A layer can be dynamically activated and composed with other layers, allowing fine-grained control of an application's runtime behaviour (Hirschfeld et al., 2008). An example for using COP for implementing context-aware applications was proposed by Schuster et al. (Schuster et al., 2011). Schuster et al. proposed Java Context-Oriented Programming (JCOP), which uses a layered approach for achieving behavioural de-/activation for a prototype mobile application. The application was implemented based on a simple context model, which was implicitly encoded with the application code. Such approach shows the feasibility to use COP for implementing self-tuning context-aware application for mobile computing environments. However, mobility induces context changes to the computational environment and therefore, changes to the availability of resources, and continuously evolving requirements would need software systems to be able to adapt to context changes. In most cases the developers on the provisional software design did not anticipate such context changes. Moreover, because of the software pervasiveness, and in order to make adaptation effective and successful, adaptation processes must be considered in conjunction with dependability and reliability by providing dynamic verification and validation mechanism, which validates the adaptation output with the adaptation goals, objectives, and architecture quality attributes (Inverardi and Tivoli, 2009). This requires the adaptation logic to be totally separated from the business code of the application. In addition, it needs a dynamic decision-making mechanism that maintains the architecture quality attributes during the adaptation.

For a more complex context-aware system, the same context information would be triggered in different parts of an application and would trigger the invocation of additional behaviours. In this way, context handling becomes a concern that spans several application units, essentially crosscutting into the main application execution. A programming paradigm aiming at handling such crosscutting concerns (referred to as aspects) is AOP (Kiczales et al., 1997). Dynamic Aspect Oriented Programming (DAOP) has emerged to enforce separation of concerns and support runtime adaptations through weaving code blocks in the application execution (Popovici et al., 2002). The assumptions made by the COP and AOP approaches, i.e. that the developer knows all the possible software adaptations in advance and designs the application accordingly, is not sufficient to fulfil this need. In addition, in COP and AOP (Tanter et al., 2006), the context model and the adaptation logic are explicitly hard-coded in the application's business code (Lincke et al., 2011); this often leads to poor scalability and maintainability (Kapitsaki et al., 2009). In contrast, Context Oriented Software Development (COSD) separates the context model and the adaptation logic from the application code, which provides the software with the ability to adapt different context models at runtime without maintaining or

modifying the application's business code (Magableh and Barrett, 2012a).

Dynamic weaving of aspects can be used for adjusting the software behaviour at runtime. However, existing DAOP techniques tend to add a substantial overhead in both execution time and code size, which restricts their practicality for small devices with limited resources (Hundt et al., 2010). The major reason for this poor performance is that the DAOP architectures like PROSE 2 (Popovici et al., 2002) provide an AOP engine running at the Virtual Machine Layer. This engine accepts aspects at runtime, then transforms them into basic entities like joinpoint requests. Joinpoint refers to a point in the control flow of a program. In aspect-oriented programming a set of join points is described as a pointcut. A join point is a specification of when, in the corresponding main program, the aspect code should be executed. The joinpoints are activated by registering them to the execution monitor. When the execution reaches one of the activated joinpoint, the execution monitor notifies the DAOP engine, which executes the advice method after evaluating the actual and past activated contexts in each joinpoint.

The actual current solution in AOP frameworks is to take snapshots of context conditions only if necessary as stated in the Reflex framework (Tanter et al., 2006), or having a context repository to store the historical context information. However, context repository would not solve this problem because at each joinpoint the framework has to evaluate the actual (active) context with the passive (historical) context stored in the repository. This requires the framework to store and process the context history for multiple events at multiple times, which adds a substantial overhead to the allocated resources. In contrast, Context-Oriented Software Middleware (COSM)-middleware maintains and evaluates the architecture evolution by tuning the adaptation process. More specifically it adapts its own functionality and verifies the adaptation results dynamically by means of dynamic decision making. Moreover, the COSM-middleware reduce the tight coupling between the context providers and context consumers, it notifies the interested components about a specific context condition without any need to compare the current context condition with previous context values.

Context Awareness ModELing Language (CAMEL) is an MDD-based approach proposed by Sindico and Grassi (Sindico and Grassi, 2009). The approach uses a domain-specific language called Java COntext Oriented Language (JCOOL), which provides a metamodel for context sensing with the supports of the context model designed using the JCOOL meta model. However, Sindico and Grassi implemented the context binding as the associate relationship between context value and context entity. On the other hand, context-driven adaptation refers to structure or behaviour elements, which are able to modify the behaviour based on context values. The structural or behavioural insertion is accomplished whenever a context value changes; it uses AOP inter-type deceleration, where the behavioural insertion is accomplished by means of an AOP advice method to inject a specific code into a specific joinpoint. The authors used their former domain-specific language to support the COP approach proposed by Hirschfeld et al. (Hirschfeld et al., 2008). Irrespective of this, JCOOL is specific to an AOP framework called the Simple Middleware Independent LayEr (SMILE) (Bartolomeo et al., 2008). SMILE platform was used for distributed mobile applications (Bartolomeo et al., 2008). The model approach in JCOOL supports only ContextJ, which is an extension of the Java language proposed by Appeltauer et al. (Appeltauer et al., 2009). The CAMEL methodology requires the software to be re-engineered whenever a new context provider is introduced into the context model. The developers must build a complete context model for the new values and maintain the underlying JCOOL Domain Specific Language (DSL) and the Unified Modelling Language (UML) model. The CAMEL methodology has adapted AOP and the Eclipse Modelling Framework (EMF) to produce a context-oriented software similar to the layered approach proposed by Hirschfeld et al. (Hirschfeld et al., 2008). This makes CAMEL limited to the EMF tool support and the ContextJ language (Appeltauer et al., 2011). From our point of view CAMEL tightly couples the software with modelling language, tool and the target deployment platform. In contrast, COSM-middleware is a generic adaptation engine provided by non-specialized language frameworks, and not being limited to a specific platform or mechanism. This gives the software developers the flexibility to construct a self-adaptive application using any object-oriented programming language and deploy it on several platforms (Magableh and Barrett, 2012a).

A context-driven adaptation requires the self-adaptive software to anticipate its context-dependent variations among its operational environment. The use of middleware in adapting the suitable adaptation approach provides a lead to achieving the adaptation results with less cost and several levels of granularity (Salehie and Tahvildari, 2009). Mobility and ADaptation enAbiling Middleware (MADAM) aims to build adaptive applications for mobile devices using architecture models (Floch et al., 2006). The middleware is responsible for constructing and analysing several variability models at runtime, which adds an intensive overhead over the mobile device. Mobile USers In Ubiquitous Computing (MUSIC) middleware (Rouvoy et al., 2008) is an extension of the MADAM component-based planning

framework that optimises the overall utility of applications when context-related conditions occur. The planning-based adaptation of MADAM employs dynamic configuration of component frameworks. In MUSIC, the planning extends further, to support seamless configuration of component frameworks based on both local and remote components and services. Thus, both components and services are plugged in interchangeably to provide functionalities defined by the component framework. In MADAM and MUSIC the dynamic decision-making is supported by a utility function. A utility function is defined as the weighted sum of the different objectives based on user preferences and QoS. However, this approach suffers from a number of drawbacks. First, it is well known that correct identification of the weight of each goal is a major difficulty. Second, the approach hides conflicts among multiple goals in a single, aggregate objective function, rather than exposing the conflicts and reasoning about them. At runtime, a utility function is used to select the best application variant; this is the so-called 'adaptation plan'.

7. Conclusions

In practice, performance and modifiability trade-off with each other as showed in the middleware evaluation. The COSM-middleware achieves self-tuning and self-configuring without degrading the allocated resources. With regards to the adaptation processes including context monitoring, detecting, decision-making and adaptation, the COSM-middleware shows better performance compared to other approaches proposed in the literature.

The evaluation of the COSD paradigm in comparison to Aspect Oriented Software Development (AOSD) shows that using different decomposition mechanisms can affect the performance of aspects/components composition at runtime. The performance and energy usage in context-oriented applications are better than in DAOP-applications. There is no doubt that Aspect-oriented frameworks can be used for developing and implementing self-adaptive applications, but their adaptability performance is poor in comparison to that of COSM-middleware. The evaluation results show that implementing self-adaptive applications with the aid of COSM-middleware can support software adaptability and variability with affordable adaptation costs and less impact on the allocated resources. Programming-level approaches like JCOP and JCOOL tend to support self-tuning of software systems with an acceptable level of performance, but the overall support for adaptability and variability is very limited in comparison with architecture evolution approaches such as MUSIC, MADAM, and COSM-middleware. However, the programming techniques are better suited to small-scale context-dependent applications, and they require intensive modification for supporting context monitoring, context detection, and dynamic decision-making.

References

- Anthony, R., Chen, D., Pelc, M., Persson, M., Torngrén, M., 2009. Context-aware adaptation in dycas. *Electronic Communications of the EASST* 19, 15.
- Appeltauer, M., Hirschfeld, R., Haupt, M., Masuhara, H., 2011. Contextj: Context-oriented programming with java. *Information and Media Technologies* 6 (2), 399–419.
- Appeltauer, M., Hirschfeld, R., Masuhara, H., 2009. Improving the development of context-dependent java applications with contextj. In: *Proceedings of the International Workshop on Context-Oriented Programming. (COP '09)*. Genova, Italy, pp. 5:1–5:5.
- Apple iPhone Operating System IOS, 2011. Ios 4.0 apple developer library. <http://developer.apple.com/library/ios/navigation/>, "[Online; accessed 1-April-2011]".
- AspectCOCA, May 2011. Aspect oriented programming framework for cocoa and objective-c. <http://www.cocoadev.com/index.pl?AspectCocoa>, "[Online; accessed 1-June-2011]".
- Barbacci, M., 2004. Software quality attributes: Modifiability and usability. Software Engineering Institute, Carnegie Mellon University, Pittsburgh PA 15213.
- Bartolomeo, G., Salsano, S., Melazzi, N., Trubiani, C., 31 2008-april 3 2008. Smile- simple middleware independent layer for distributed mobile applications. In: *Proceedings of the Wireless Communications and Networking Conference. (WCNC 2008)*. Las Vegas, USA, pp. 3039–3044.
- Boehm, B. W., Clark, Horowitz, Brown, Reifer, Chulani, Madachy, R., Steece, B., 2000. *Software Cost Estimation with Cocomo II*, 1st Edition. Prentice Hall PTR.
- Buck, E., Yacktman, D., 2010. *Cocoa design patterns*, 2nd Edition. Developer's Library.
- Cheng, B. H., Giese, H., Inverardi, P., Magee, J., de Lemos, R., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G. D. M., Dustdar, S., Finkelstein, A., Gacek, C., Geijs, K., Grassi, V., Karsai, G., Kienle, H., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J., 2008. Software engineering for self-adaptive systems: A research road map. In: *Proceedings of Dagstuhl Seminar, Software Engineering for Self-Adaptive Systems. (Dagstuhl Seminar '08)*. Dagstuhl, Germany, pp. 1–26.
- Costanza, P., Hirschfeld, R., Meuter, W. D., 2006. Efficient layer activation for switching context-dependent behavior. In: *Proceedings of the 7th Joint Modular Languages Conference. (JMLC '06)*. Oxford, UK, pp. 84–103.

- de Lemos, R., Giese, H., Müller, H., Shaw, M., Andersson, J., Baresi, L., Becker, B., Bencomo, N., Brun, Y., Cikiric, B., Desmarais, R., Dustdar, S., Engels, G., Geihs, K., Goeschka, K. M., Gorla, A., Grassi, V., Inverardi, P., Karsai, G., Kramer, J., Litoiu, M., Lopes, A., Magee, J., Malek, S., Mankovskii, S., Mirandola, R., Mylopoulos, J., Nierstrasz, O., Pezzè, M., Prehofer, C., Schäfer, W., Schlichting, W., Schmerl, B., Smith, D. B., Sousa, J. P., Tamura, G., Tahvildari, L., Villegas, N. M., Vogel, T., Weyns, D., Wong, K., Wuttke, J., 2011. Software engineering for self-adaptive systems: A second research roadmap. In: Proceedings of Dagstuhl Seminar, Software Engineering for Self-Adaptive Systems. (Dagstuhl Seminar '11). Dagstuhl, Germany, pp. 1–26.
- Filman, R. E., Elrad, T., Clarke, S., Akşit, M. (Eds.), 2004. Aspect-Oriented Software Development. Addison-Wesley.
- Floch, J., Hallsteinsen, S., Stav, E., Eliassen, F., Lund, K., Gjørven, E., 2006. Using architecture models for runtime adaptability. *IEEE software* 23 (2), 62–70.
- Gassanenko, M., 15-18 October 1998. Context-oriented programming. In: Proceedings of the European Forth Conference. (EuroFORTH '93). Mariánske Lázně (Marienbad), Czech Republic, pp. 1–14.
- Geihs, K., Evers, C., Reichle, R., Wagner, M., Khan, M. U., 2011. Development support for qos-aware service-adaptation in ubiquitous computing applications. In: Proceedings of ACM Symposium on Applied Computing. (SAC '11). TaiChung, Taiwan, pp. 197–202.
- Hirschfeld, R., Costanza, P., Nierstrasz, O., March 2008. Context-oriented programming. *Journal of Object Technology* 7 (3), 125–151.
- Hundt, C., Stöhr, D., Glesner, S., 2010. Optimizing aspect-oriented mechanisms for embedded applications. In: Proceedings of the 48th international conference on Objects, models, components, patterns. (TOOLS'10). Malaga, Spain, pp. 137–153.
- Inverardi, P., Tivoli, M., 2009. The future of software: Adaptation and dependability. In: Lucia, A., Ferrucci, F. (Eds.), *Software Engineering*. pp. 1–31.
- Kapitsaki, G., Prezerakos, G., Tselikas, N., Venieris, I., 2009. Context-aware service engineering: A survey. *Journal of Systems and Software* 82 (8), 1285–1297.
- Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwin, J., 1997. Aspect-oriented programming. In: Proceedings of the European Conference of Object-Oriented Programming, (ECOOP '01). Vol. 1241 of LNCS. Budapest, Hungary, pp. 220–242.
- Lincke, J., Appeltauer, M., Steinert, B., Hirschfeld, R., December 2011. An open implementation for context-oriented layer composition in contexts. *Science of Computer Programming* 76, 1194–1209.
- Magableh, B., Barrett, S., 2009. Pcoms: A component model for building context-dependent applications. In: Proceedings of the First International Conference on Adaptive and Self-adaptive Systems and Applications. (Adaptive '09). Athens, Greece, pp. 44–48.
- Magableh, B., Barrett, S., June 2011. Self-adaptive application for indoor wayfinding for individuals with cognitive impairments. In: Proceedings of the 24th International Symposium on Computer-Based Medical Systems. No. 1 in (CBMS '11). Bristol, United Kingdom, pp. 1–6.
- Magableh, B., Barrett, S., June 2012a. Context oriented software development. *Journal of Emerging Technologies in Web Intelligence (JETWI)* 3 (4), 206–216.
- Magableh, B., Barrett, S., 2012b. Productivity evaluation of self-adaptive software model driven architecture. *International Journal of Information Technology and Web Engineering (IJITWE)* 6 (4), 1–19.
- Mikalsen, M., Paspallis, N., Floch, J., Stav, E., Papadopoulos, G., Chimaris, A., 2006. Distributed context management in a mobility and adaptation enabling middleware (madam). In: Proceedings of the 2006 ACM symposium on Applied computing. (SAC '06). Dijon, France, pp. 733–734.
- Oreizy, P., Gorlick, M., Taylor, R., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D., Wolf, A., 1999. An architecture-based approach to self-adaptive software. *Intelligent Systems and Their Applications* 14 (3), 54–62.
- Paspallis, N., Nov 2009. Middleware-based development of context-aware applications with reusable components. Ph.D. thesis, University of Cyprus, Department of Computer Science.
- Popovici, A., Gross, T., Alonso, G., 2002. Dynamic weaving for aspect-oriented programming. In: Proceedings of the 1st international conference on Aspect-oriented software development. (AOSD '02). Enschede, The Netherlands, pp. 141–147.
- Rouvoy, R., Beauvois, M., Lozano, L., Lorenzo, J., Eliassen, F., 2008. Music: an autonomous platform supporting self-adaptive mobile applications. In: Proceedings of the 1st workshop on Mobile middleware: embracing the personal communication device. (MobMid '08). Leuven, Belgium, pp. 6:1–6:6.
- Salehie, M., Tahvildari, L., May 2009. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* 4, 14:1–14:42.
- Schuster, K., Appeltauer, M., Hirschfeld, R., July 2011. Context-oriented programming for mobile devices: Jcop on android. In: Proceedings of the Workshop on Context-oriented Programming (COP) 2011, co-located with ECOOP 2011. Lancaster, UK, pp. 180–195.
- Sindico, A., Grassi, V., 2009. Model driven development of context aware software systems. In: Proceedings of International Workshop on Context-Oriented Programming. (COP '09). Genova, Italy, pp. 7:1–7:5.
- Tanter, É., Gybels, K., Denker, M., Bergel, A., 2006. Context-aware aspects. In: Proceedings of the 5th International Symposium on Software Composition. (SC 2006). Vienna, Autriche, pp. 227–242.