CHAPTER 3

PROPOSED DYNAMIC ASPECT WEAVING
ADAPTATION APPROACH

There are a number of challenges arising from the complexity of ubiquitous computing environments that feature large mobile and distributed systems containing a variety of inter-dependent services and devices. These systems are highly dynamic and fault-prone, and must be able to adapt to the changing contexts and resource availabilities and recover from the failures of devices, services and applications. In order to make these systems portable to different environment, those systems are to be re-developed for new environments by considering environment specific policies regarding the usage of resources for performing various kinds of tasks. These issues place a bottleneck on the rapid development and prototyping of the systems in these ubiquitous environments (Anand 2005).

The dynamic adaptable mechanisms for addressing the issues of accommodating changes are to be established with less performance overhead. Many such solutions have been developed using reflections and aspect-oriented programming. The works described in the papers (Grace et al 2007, Bencomo 2006, Bencomo 2005) specified the adaptable solutions derived using reflection and aspect-oriented approaches. Using reflection it is possible to get the structural and behavioral information about the component and change its behavior. This process leads to define meta-level components
to maintain the base-level component’s structural and behavioral information and make the changes at the meta-level, which is reflected on the base-level component through reconfiguration.

Another approach is the Aspect-Oriented Programming (AOP), which supports to separate the concerns distributed across the system, and enables to make the changes on these concerns in an efficient way. AOP allows encapsulating cross-cutting concerns as ‘aspects’ and uses weaving process to bind these concerns with the corresponding components. It allows only handling the changes on the cross-cutting concerns in an efficient way. The work described here focused on handling dynamic changes in any function of a software system using aspect-oriented approach. Also, applying observer pattern to handle the issues of binding the aspects across the system and using base languages reflective facility to enable the run-time reflection stated in (Hanenberg et al 2003, Kai Bollert 1999) are used to derive the following principles to achieve the dynamic adaptability.

- Redefining ‘aspects’ to encapsulate the changes in the functionalities of the system instead of cross-cutting concerns using modularization principle.

- Generation of dynamic meta-components to provide structural information using introspection concept.

- Dynamic wrapper for the invocation of component functions using adapter pattern and dynamic meta-component.

The detailed discussion about these principles is given in the following sections.
3.1 DYNAMIC CHANGE ASPECT SOLUTION FOR ADAPTABILITY

Nowadays, most of the software systems are distributed systems deployed in ubiquitous environment. These systems should be capable of dynamically adapt the changes in the environment or in the user needs and should have the feature of incremental adaptability. Incremental adaptability means coping with changing requirements without modifying the previously defined software components. Better modularization results in increasing the size of design model of a software system and providing stability over the changes (Rashid et al 2010). Hence, modularization is considered as one of the approaches used to create dynamically adaptable system. The conventional object-oriented model provides modularization for localizing the related abstractions.

An object-oriented model is defined with composition, encapsulation, message passing and inheritance features to implement adaptability (Pryzyblek 2011). By making use of these features, adaptability is achieved through the composition of objects that implement the changes in the requirements with the existing application objects. This makes the system adapt to these changes through recompiling or reconfiguring the whole system, which will affect the structure and behavior of the existing applications. It is concluded that this process might lead to re-engineering the whole software system (Constantinides et al 2000).

The main feature of AOP is to separate the concerns spread over the system (cross-cutting concerns) and representing them as aspects. This separation of concerns principle gives a new dimension for modularization through encapsulating the cross-cutting concerns in separate modules (Steimann 2006) and makes the process of changing the cross-cutting concerns localized to the aspects without affecting the existing application
components. The aspect-oriented paradigm addresses the two major issues in the software development such as software evolution and maintenance using the separation of concerns principle, which are considered as the limitations of object-oriented paradigm (Haussge 2006, Moreira and Araujo 2004). This separation of concern principle is used in this work with a modification in the definition of an aspect.

This work proposes the solution for dynamic adaptability by introducing the novelty in the AOP modularization approach. Instead of modularizing the cross-cutting concerns, the dynamic changes are modularized using aspect-oriented approach. Here all dynamic changes in the requirements or in the system functions are captured and modularized as aspects. This modularization approach separates the implementation of adaptation strategy form the existing application components, which allows the system, to adapt to the dynamic changes without changing the structure or behavior of the existing application components.

3.1.1 Modularization using Object-Oriented and Aspect-Oriented Paradigms

Modularization is defined as a narrow interface, which hides the implementation details, keeps low coupling and high cohesion and it allows making the changes in the system without major effect. It also allows for handling complexity of the system and reusing the various parts of the system in a different context (Brito-E-Abreu 2001). At an abstract level, the architecture of a system is specified with the set of modules and their interconnections. In general, a module is defined as an aggregation of algorithm’s implementations and data structures to deliver a specific task. Each module should have its own state, an interface and should collaborate with other modules to deliver its functionality. Modularity is an internal quality characteristic of a software.
Using object-oriented approach, a system is decomposed into the set of abstractions, and each abstraction is specified using an abstract data type or class, which encapsulates the data structure, its internal linking, accessing procedure and modifying procedure. In an object-oriented system, the module is defined as the set of classes, module cohesion is defined as the cohesion among the classes belonging to the module, and module coupling is defined as the coupling among the classes across the modules. Module coupling implies scattering of the functions across the module, which increases the dependency among the modules and affects the primary objective of modularization. Even an object-oriented system is carefully designed using object-oriented principles that may implement some concerns in a non-modular way, which leads to have the scattered and tangled code throughout the system. Some of those concerns are logging, auditing, synchronization and transactional management functions. Aspect-Oriented Programming improves this situation by supporting the modularization of cross-cutting concerns or by moving the scattered functions into aspects.

3.1.2 Modularization and Adaptability

Software evolution for incorporating environmental changes and/or user’s requirements changes are considered as an inevitable process. This can be performed by defining the modules using the principle of semantic coherence, which localizes the changes. An adaptability feature of software supports the system to evolve easily. Hence a better way of designing a system with adaptability is achieved using modularization principle. This implies creating modules as independent and semantically coherent units through decomposition and can be integrated and deployed as it is required. While creating such adaptable software, facility has to be provided to incorporate the changes without affecting the existing structure and behavior
of the system. Also, the adaptable system should be able to implement and reflect the changes which are scattered across the modules.

Modularization principles are realized in object-oriented paradigm by exploring the powerful abstraction and encapsulation capabilities. Thus, object-oriented adaptable software can be created by properly applying abstraction and encapsulation principles. In an object-oriented paradigm establishing the communication between the modules requires explicit knowledge about the modules, which violates encapsulation. Also, representing the concepts or entities or logics in a system using abstraction does not imply that they are independent of each other and localize the changes. This implies that the abstraction and encapsulation are inadequate (Khatchadourian 2006) for creating adaptable software systems. Hence, reducing the coupling between the modules through abstraction and encapsulation is not sufficient to effective implementation of the adaptability.

The adaptability of the software can be improved by increasing the degree of software element independency through removing scattered information across the software elements. In general, concerns represent the functional and non-functional elements of the system. The information scattered across the functional and non-functional elements is represented as cross-cutting concerns. In order to manage evolution in these cross-cutting concerns, modularization principles should be applied. For proper modularization, the identification and quantification of the cross-cutting concerns are required (Eaddy et al 2007) The separation of concerns principle stated in aspect-oriented programming supports to encapsulate the cross-cutting concerns in a separate unit that imposes change localization in a better way. Hence, aspect-oriented approach is considered as the better solution for modularization. As it is stated by the authors Rizvi and Khanam (2010) in
their work, the aspect-oriented programming does not replace object-oriented programming but it improves the quality of object design.

3.1.3 Redefined Aspects and Dynamic Adaptability

Pervasive and mobile systems are required to operate in environments in which the availability of resources and services may change significantly during system operation. As a result, these systems should be capable of adapting to these changes at run-time to offer the best possible level of service to their users. The systems that are capable of accommodating dynamic changes in the resources or services are referred to as dynamic adaptable software systems. The changes may be anticipated or unanticipated ones. The anticipated changes are handled by determining them during the development period. The solutions to represent the changes are incorporated into the system by altering the structure of the functions associated with them. But, unanticipated changes cannot be handled as like anticipated changes. For improving the dynamic adaptability, the adaptation of unanticipated changes should be taken into account. Hence, the adaptability solutions offered using object-oriented and aspect-oriented approaches are not sufficient to achieve dynamic adaptability of the unanticipated changes.

Since, aspect-oriented approach provides support for creating adaptable software, it is derived that refining this approach can be considered as a solution for dynamic adaptation of the unanticipated changes. The process of refining the aspect-oriented approach involves refactoring the concerns which modify the purpose of separation of concerns principle. Here „concerns” are named as „change-concerns” that represent the changes that occurred during software evolution. Since these changes may be associated with one or many component functions, the change-concerns are considered as cross-cutting change-concerns over those functions. These cross-cutting change-concerns are to be modularized to improve the dynamic adaptability.
Dynamic adaptability through modularization is achieved by applying the separation of concerns principle stated in the aspect-oriented approach. The purpose of this principle is redefined as to capture the change-concerns in the separate modules, which allows the system to perform change adaptation process without considering the structure and behavior specification of the system. The unit that encapsulates the change-concern is named as „Dynamic-Change Aspect”. Hence aspect is renamed as „Dynamic-Change Aspect”, which has been illustrated in Figure 3.1. While modularizing the changes using aspects, fragile point-cut problem may arise due to addition or deletion of code fragments in the advice as stated in (Stoerzer et al 2005). This point-cut semantic change is addressed in this work by specifying dynamic wrappers as the point-cuts.

Figure 3.1 Dynamic change aspect process structure
The proposed solution modularizes the dynamic changes as aspects, which allows the system for adapting to the changes without modifying the components of the existing system. This reduces the effort executed towards the implementation of the dynamic adaptation strategy, which improves the adaptability efficiency of the system. The steps involved in making the system adaptable to the dynamic changes are stated below.

a. Specify the dynamic changes in the requirement as Dynamic-Change aspects.

b. Define point-cuts, which specify the function where the Dynamic-Change aspect is to be weaved.

c. Define advices, which specify the implementations of the dynamic changes in the requirement.

d. Define joint-points, which specify the locations in the functions into which the advices are to be weaved.

3.2 DYNAMIC STRUCTURAL INTROSPECTION

There is a growing demand for the software that will cope with diversity in devices, platforms and user needs. In order to achieve this, the software should have the feature of adapting to these diversified environments. The software has to take up the behavior based on the environment type and user need. Hence, the software should have the ability to identify the variation element and to alter its functionality. For altering the functionality, the system has to reason about itself and act accordingly. This property is referred to as reflective (Kasten et al 2002).

Reflection is used to maintain the structural and behavioral information as self-representation through meta-level components, so as to
modify the structure and behavior of the base-level components and update
the meta-level components with the changes incorporated on the base-level
components. The reflection process is used for performing (i) introspection to
extract the structure and behavioral information of the base-level components
(ii) intercession to perform the modification on the base-level components and
(iii) reification for maintaining the consistency between meta-level and base-
level components. Hence, reflection is suitable for incorporating the adaptability feature into a software system. While performing the adaptation
process using reflection, performance overhead is increased due to
maintaining the consistent self-representation of the base-level components
throughout the software life time without knowing the actual needs. The
primary requirement of the current software is to adapt the dynamic changes
with less performance overhead. Hence, in the present work the reflection
model is refined by separating introspection or inspection of the components
from reflection or intercession of the changes on the components. Also a
separate technique has been introduced to incorporate the changes on the
components.

3.2.1 Reflection in Adaptability

Reflection is used in software systems to reason about itself
through maintaining the meta-information about the interfaces, methods and
objects defined in them. For instance, IMetaEncapsulation interface in
OpenCOM maintains the internal structure of interfaces (Clarke et al 2001):
OpenORB meta-level interfaces contain information about the structure of the
component, binding interfaces of the component, execution environment of
the component and resources used by the component (Saikoski and Coulsons
2001). The meta-frameworks of the OpenORB reconfigurable systems consist
of details about various communication services, discovery services, security
services and resource services. Thus the various systems designed with the
capability of reflection are maintaining the meta-details, which are used for
dynamic arbitrary invocation of interfaces and dynamic composition of
services.

The communication between base-level and meta-level component
during reification and intercession operations is established using the Meta
Object Protocol. The issues involved in reflection process are defining data
structures for representing and storing meta-information, implementing
protocols to associate base-level and meta-level components and consuming
time to perform reification, intercession and introspection operations. Also,
the tasks involved in the reflection process are implemented as static tasks.
The performance of the reflection process can be improved by addressing the
above stated issues in an efficient way.

The solution to define the reflection in an efficient way is to reduce
the storage of meta-information, which will reduce the effort for performing
the introspection, intercession and reification operations. This solution can be
achieved by allowing the operations of the reflection process to be executed at
run-time based on the demand rather than in a static way. This proposed
approach is realized in the present work through the dynamic way of
performing introspection to extract the structural information of the
component, which is associated with the changes stated in the requirements at
run-time.

3.2.2 Dynamic Meta Model

As the existing reflection models suffer from performance
overhead, better optimal solution for adaptation has to be provided. The
present work mainly focused on providing the support for adapting to
dynamic changes in the environment or in the user needs. Hence, the
adaptation strategy has to be invoked only at the time of the arrival of change
events. Also, maintaining meta-information of all the components at all times will consume more effort and will lead to inconsistency. In order to overcome this issue, a new strategy has been derived, which allows separating the components that are used for meta-information maintenance from modification of the components that are associated with the changes. This separation supports applying different strategies for each process.

The strategy for creating self-representation or meta-information of the component is defined as inspecting the structural information of the components, which includes the details about interfaces, operations, parameters, implementation classes and attributes based on the change request. The structure to describe the change request is formatted with the data that includes component service name, operation name and change specification. Introspection for extracting the complete information about the component using service name at run-time has to be applied. This mechanism is referred to as „Dynamic Structural Introspection” (DSI). Since the objective of the present work is to provide the solution for adapting dynamic unanticipated changes in the functionalities of a software system, it is identified that DSI mechanism can play a vital role in providing the information about the functional components in an on-demand way. Hence, the model for applying introspection to provide structural information about components at run-time is appropriate for dynamic adaptation. This model is referred to as „Meta-Component Model”. This model should be described with the set of services for getting the complete structural details about the component in order to reuse this solution, which has been shown in Figure 3.2.
Figure 3.2  Dynamic meta model using structural introspection

Applying introspection at run-time will lead obtaining current consistent information of the component. It also reduces the unnecessary utilization of resources for static maintenance of the meta-information of the components.

3.3  Dynamic Function Invocation

Software has to modify itself while facing the changes in the requirements of the user or in the context information. These changes can be categorized as pre-determined that includes changes in the business service mechanism based on the customer type, variation in the communication mechanism based on the network type, variation in access policy according to the user type and non-pre-determined types, which specify unanticipated dynamic changes like business policy changes based on the current need of
the customers, changes in plantation crop lifecycle monitoring policy according to weather prediction and display mechanism variation due to the arrival of new device.

The adaptability solutions described in Section 2.4 are mainly concentrated on addressing the pre-determined changes that are analyzed and defined at the design and implementation phase. These approaches could not be applicable for adapting the non-pre-determined changes. This issue can be easily addressed by separating the implementation of the changes from directly modifying the system components that are incorporating the changes. In order to accomplish this, the functions of the software components should change their behavior due to the arrival of dynamic changes by retaining their structure without any modification. While analyzing this, it is identified that binding the change implementation code with the wrapper of the component function generated at run-time is more appropriate solution and it is used to achieve dynamic binding. This dynamic binding is the simplest mechanism when compared with the approach stated in (Gilani and Spinczyk 2004) to achieve dynamism in applying aspects through feature set model and family based aspect weaver. The adapter pattern is the one of the design patterns stated by Erich Gamma et al (1995) is appropriate to generate the wrappers at run-time using dynamic structural introspection.

3.3.1 Dynamic Invocation Technique

The functions defined in a software system are invoked for processing the requests posted by the users either directly using procedure calls or indirectly using messages or method signatures. In a distributed system, the direct and indirect invocation mechanisms are used only for invoking the local procedures or objects or methods, not for invoking remote procedures or methods. Hence, the separate set of protocols namely Remote Method Invocation (RMI), Remote Procedure Call (RPC) and Simple Object
Access Protocol (SOAP) are used to invoke remote methods or procedures or objects. Most of the middleware frameworks such as CORBA, COM and web service are using these remote method invocation techniques for accessing remote objects or services, while executing the static request.

For processing the dynamic request, the methods or services associated with the request are to be invoked at run-time. Hence, the middleware frameworks extended their remote invocation capability using reflection to access the functions at run-time. The dynamic invocation implies that the binding between the client and server object should be established in a dynamic way. In order to achieve this, the stub or proxy objects defined in remote invocation frameworks are to be generated at run-time through reflection process. Hence, the limitations of the reflection process restrict the performance of the dynamic proxy or stub mechanism. To overcome these limitations, this work proposed the generation of dynamic wrapper to invoke the methods at run-time through dynamic structural introspection.

3.3.2 Dynamic Wrapper Generation

Adapter pattern comprises the adapter, adaptee and the existing interface classes. An adaptee class is defined with new a set of services; existing interface is defined with the facility to access the set of services which are not compatible to access the new services. The adapter class allows the existing interface to access the new set of services by invoking those methods.

According to the dynamic change adaptability scenario, the issue is to establish the compatibility between the dynamic change implementation and the component functions that are incorporating these changes. Hence, Adapter object is designed to connect the ‘change implementer’ with the component functions associated with the change. The software element that is
responsible for implementing the dynamic change is termed as change implementer. The connection between the change implementer and component functions is specified as invocation of the change related functions at run-time and binding the change implementer with that function, which is shown in Figure 3.3. It shows that the role of adapter object is to dynamically invoke the functions that are affected by the dynamic changes.

The responsibilities of the adapter object are the generation of the method signature, which includes method name, parameter names and their values and the invocation of the function using the method signature. The content of method signature is defined using meta-component model through dynamic structural introspection. The process of invoking the function to handle the dynamic changes is referred to as „Dynamic Wrapper’ process and the component that encapsulates this process is termed as „Dynamic Wrapper Component”. Hence, this dynamic wrapper supports binding the changes with the corresponding components functions at run-time irrespective of the incompatibility existing between them.

Figure 3.3 Dynamic wrapper structure
3.4 DEMONSTRATION OF THE PROPOSED DYNAMIC ASPECT WEAVING ADAPTATION APPROACH

Consider a banking transaction system, which provides support to the bank customers for performing various transactions including opening or closing their accounts in the bank, depositing and withdrawing the money, fund transfer between the accounts and bill payment through the bank anywhere in the world using various platforms including J2EE, .NET, Android and so on and also through different communication protocols according to the underlying networks at any time and using different types of devices including desk top, lap top, tablets and mobile phones. This system should be made available at all times for the customers to satisfy their requirements. It leads to updating the system with variations in the transaction strategy, platform, communication strategy and device. The above specified variations may occur while the system is in use and are referred as dynamic changes. Hence, the system has to be defined with the feature of adapting to these dynamic changes without shutting it down. The following discussion is focused on describing the scenarios in the system, which involves the issues that trigger the event of modifying the functionalities of the system.

Authentication Scenario: The customers of a banking transaction system are allowed to perform the transaction in a controlled and secured way. In order to accomplish this feature, only authenticated users are allowed to access the transaction services. For ensuring the authentication, the customers are provided with the user name and password that is auto-generated using the bank account and customer specific information. Using the auto-generated secret code at the time of transaction, the user will get validated before performing the banking operations. This approach may fail in certain cases
that will lead to modifying the strategy to overcome the limitation faced. Lack of uniqueness in the generation of password and secret code is considered as the major cause of this failure. Hence, the need for following changes in the authentication strategy occurs at run-time to handle this issue.

**Change AC1:** Modify the auto-generation procedure by increasing the password and secret code length.

**Change AC2:** Use the unique identity number provided by the government for their citizens to generate the password.

**Change AC3:** Include the second level of verification using bio-metric information.

**Money Transaction Scenario:** All the transactions in a bank involve money. Nowadays, the various online services are supporting the people to do their daily activities in a better way. In this regards, all the money associated transactions are to be integrated with the banking service. Consider that all the banking transactions that involve the amount that are chargeable under income tax are to be recorded in the corresponding customer’s income tax file. To accomplish this, changes are to be performed in the banking transaction operations. Similarly for credit payment transaction, support has to be provided by the bank through modification in the appropriate services. These service inclusions may be considered at run-time through the following changes.

**Change MTC1:** Changes in the money deposit, money withdrawal, bill payment and fund transfer to provide the support for income tax department.
Change MTC2: Modifying the money withdrawal and bill payment functions by incorporating an account authorization service for supporting credit payment service.

User Interface Management Scenario: To facilitate the customer to have an easy accessibility of the banking transaction service, the well designed user interface forms are provided because the users are accessing the transaction services through various devices includes desktop, laptop and mobile phones. To leverage the usability of the user interface forms in these devices, the changes in display format is required. These change issues occur while the system is in execution.

Change UIC1: The change in the screen layout to accommodate the various display devices.

Change UIC2: Modifying the content block of the screen based on the device type.

Realization of these changes in the system using redefined dynamic aspect weaving approach is achieved as follows: (i) Dynamic Aspects are used to capture the changes described in the various scenarios (ii) The information of the online banking transaction components associated with these changes are gathered using Dynamic Structural Introspection mechanism and (iii) The corresponding functions in those components are invoked through Dynamic Function Invocation process. By applying these techniques, the changes on the running services are made without modifying the structure of those services.
3.5 SUMMARY

The proposed techniques for achieving dynamic adaptability are dynamic change aspect to capture changes in the requirements at run-time, dynamic structural introspection to capture component details associated with dynamic changes and dynamic function invocation to invoke the methods associated with the changes at run-time. Dynamic change aspect provides the features of encapsulating the changes in a separate unit, which provides flexibility in adding or removing the changes and abstracts the implementation process and implementing the changes using different strategy according to the context. Dynamic structural introspection and dynamic function invocation enabled the generation of wrapper for the functions associated with the changes stated at run-time. These techniques allowed dynamic weaving of aspect with the function of the component through weaving the dynamic change aspect into the wrapper generated using dynamic function invocation and dynamic structural introspection. The separate framework for the generation of dynamic change aspect, the implementation of the dynamic structural introspection and the generation of dynamic wrapper are to be considered in the future work.