CHAPTER 2  
BACKGROUND CONCEPTS AND RELATED WORKS

2.1 SOFTWARE ADAPTABLE

Software maintenance is defined according to (IEEE STD 1219 1998) as the modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment. Maintenance is an evolutionary process aided by understanding the happenings of a software system over time. Maintenance is needed to ensure that the system continues to satisfy user requirements. Maintenance process associated with adapting the changes in user requirements or in execution environment is referred as to adaptive maintenance.

Changes to the software system are defined in the form of adding, removing, or replacing software components or services. Software in ubiquitous environment will need to cope with variability, as software systems get deployed on an increasingly large diversity of computing platforms and operate in different execution environments. Software adaptability is defined as the ability of software systems to withstand changes in their environment (Fox and Clarke 2009). A software system will be adaptable provided its software architecture is itself adaptable. Software adaptation is one of the fundamental design requirements for the modern software systems. The ability to adapt so as to meet future change requests is
a necessary condition to prolong the lifespan of a software product. Software adaptation can be categorized as either static (adaptable) or dynamic (adaptive) (Bader et al 2005). The components of adaptable software system are selected, customized, and then only executed after compilation. In contrast, adaptive software systems can be altered while the system is running without the need to recompile and reboot the running system.

Software maintenance cost is estimated to be about 60% of the total development cost of a system, which can be reduced by designing software architecture with evolvable and reusable components (Tavainen 2008). During software evolution, poorly designed software architecture requires components to be reengineered before they continue to be used. Hence, the principles of software architecture designing should be changed towards the creation of evolvable coarse-grained components and their overall connection infrastructure through the middleware software layer. Such evolution software architecture should allow for change, extensibility and the reduction of component interdependencies.

Evolution can occur at a number of levels in a given software architecture, with changes persisting both over the short and long term (Betty et al 2009). Short-term change requirements are:

- local adaptation involving small and localized change
- data evolution, which implies behavioral adaptation or adapting data streams depending upon resource availability
- service evolution, which is a non-localized adaptation (Pemberton 2008).
Long-term change requirements are new service integration that is integrating client application running on a low-power device with Wireless Access Protocol (WAP) enabled device and paradigm/process shift that is a move from distributed banking model to internet banking model (Pemberton 2008). Also, Fayad and Cline (1996) in their work stated that the system can adapt to high level changes through extensibility and flexibility and it can adapt to low level changes through tunability and fixability.

2.2 ASPECT ORIENTED PROGRAMMING

Nowadays, Object-oriented programming (OOP) has become the dominant programming paradigm where a problem is decomposed into abstractions that are represented as objects. Object is defined to encapsulate one concern at a time with a well-defined interface to access it. Concerns can be functional such as business logics, entities and/or non-functional ranging from high-level quality of service to low-level services like buffering, caching and logging (Lee 2002). This encapsulation separates the user from the implementation of the object. Also, OOP promotes code reusability through inheritance property. Enhancing the concern encapsulated in an object using inheritance will lead to repeat some common parts of the concern in all subclasses. This code repetition will lead to the code tangling problem. This will also induce the maintainability problem.

Aspect-Oriented Programming (AOP) is a technology provides solution to the code tangling problem through separating the common operations existing in various classes, which are referred as crosscutting concerns and encapsulating them into single unit called aspect. An aspect is a modular unit of crosscutting implementation. It encapsulates behaviors that affect multiple classes into reusable aspects. Aspectual requirements are
concerns that introduce crosscutting in the implementation. With AOP, each aspect can be automatically combined with the software elements that are affected by it, using an aspect weaver, which is shown in Figure 2.1. As a result, a single aspect can contribute to the implementation of various procedures, modules or classes and it can introduce reusability. An AOP language has three critical elements for separating crosscutting concerns. They are:

- join-point that specifies the location in a system where the cross-cutting concern can be plugged in
- point-cut that specifies the context describing the join-points
- advice that describes the behavior associated with the concern (Filman et al 2004).

Aspect-oriented software development has to be focused on all the phases of software life-cycle for creating aspect-oriented system (Rashid 2003). Hence, capturing aspects in requirement phase is essential to increase the consistency of requirements and architectural design with implementation. It also provides traceability for aspects across life-cycle activities. AOP systems can be analyzed with respect to three critical dimensions: the kinds of quantifications allowed the nature of the actions that can be asserted and the mechanism for combining base-level actions with asserted actions (Filman and Friedman 2000). The role of aspects also includes the representation of the cross-cutting concerns like handling command injection vulnerability in web services as described in (Shamughaneethi et al 2012).
Figure 2.1 Aspect weaving process structure
AspectJ is a general-purpose AO extension to Java, developed by Xerox Palo Alto Research Center, which enables plug-and-play implementations of crosscutting in Java (Lee 2002). AspectJ development tool consists of a set of command line tools, a set of GUI tools and online documentations. The command line tools include a compiler for AspectJ language (ajc), a debugger for Java class files produced by ajc (ajdb), and a documentation generator (ajdoc) that produces HTML API document with crosscutting structure. The compilation process in AspectJ is described with two tasks are source file compilation in the front end and byte code transformation in the back end (Hilsdale and Huginin 2004).

The basic language constructs are join-point, advice and aspect. The following discussion about the constructs is taken from (Przybylek 2011).

- An aspect is a module that implements the behavior and structure of a cross-cutting concern. It can be a class or interface or an extended class or declaration of attributes and operations. In addition, it can extend other aspects; it can also declare advices, point-cuts, introductions and parent declarations.

- A join-point is an identifiable location in the program flow where the implementation of a cross-cutting concern can be plugged in. Typical examples of join-points include a throw of an exception, a call to a method and an object instantiation.

- A point-cut is a language construct designed to specify a set of join-points and obtain the context (the target object and the arguments of an operation) surrounding the join-points as well.

- An advice is a method-like construct used to define an additional behavior that has to be inserted at all join-points
picked out by the associated point-cut. The body of an advice is the implementation of a cross-cutting functionality. The advice is able to access values in the execution context of the point-cut. Depending on the type of advice, whether “before”, “after” or “around,” the body of an advice is executed before, after or in place of the selected join-points. An around advice may cancel the captured call, may wrap it or may execute it with the changed context.

2.3 REFLECTION

Software must be flexible to support changes in the continually growing requirements. A process of design for evolution, which is defined with evolvable and reusable component based software architectures using software engineering principles, supports to implement adaptive maintenance changes. Such evolvable and reusable components are created by applying self-discovery (reflection) to support evolution through self-reasoning and adaptation. Reflection is the capability of the computational system to maintain the internal structural and behavioral information about itself and to adapt to the changing conditions using this information (Cazzola et al 2004). Soler and Lovelle (2001) categorized reflection as follows:

- introspection, which observes the structure of the components or extracts information about classes, methods and properties at dynamic; it is also referred to as refraction and does not allow modifying the components or classes
- structural reflection is the ability of reasoning and modifying the structure of the components
- computation reflection is the determination of component’s internal logic and ability to change its behavior. The reflection
supports the dynamic adaptation of object-oriented software and it has the ability to separate the functional and adaptation code (Dowling et al. 1999).

The design of a reflective system mainly consists of providing reflective mechanisms to establish meta-models. The reflective mechanisms provide observation and control features that can be divided into four classes as stated by (Killijian and Fabre 2003).

- Reification mechanism, which allows the base-level components to exhibit information about its own computation
- Introspection mechanism, which is used by the meta-level to obtain structural information about the base-level
- Behavioral intercession that enables the meta-level to control base-level computation
- Structural intercession, which enables the meta-level to update base-level entities.

A reflective system architecture is constituted with two layers, namely base-level and meta-level as shown in Figure 2.2. The base-level realizes the functional aspect of the system, whereas the meta-level realizes the nonfunctional aspect of the system (Sadjadi 2004). Functional requirements are associated with core functionalities, whereas non-functional requirements are associated with quality aspects of a system such as fault tolerance, security and evolution. The meta-level is causally connected to the base-level using the data structures maintained for representing every characteristic such as structure, behavior, interaction and so on of the base-level, generally called reification. The base-level is continuously kept consistent with its reification that is each action performed in the base-level is reified by the reification and each change performed in the meta-level is also reflected on the base-level.
2.4 ADAPTIVE MIDDLEWARE

Software architecture provides an abstraction for modeling, implementing and evolving complex software systems in distributed, heterogeneous, mobile and pervasive environments. Such software architectures are described in terms of software components, connectors and their configurations. Since these architectural abstractions are not providing direct support at the level of system implementation, the evolution of middleware architectures to provide implementation level software solution for providing communication, security, persistence, transaction and other quality of services is indispensable. Middleware is a software infrastructure...
offering abstractions and services for distributed applications in heterogeneous environments (Mellon 1997).

Middleware also enables a separation of concerns between processing, application level logic and its communication infrastructure (David and Ledoux 2002). Such middleware solutions have been placed between user clients and application servers and they are used to address the software engineering challenges inherent in distributed, pervasive and mobile computing across platforms. These middleware models can be deployed in a separate server to perform the functionalities of the clients or servers of an application, which are located across the world and connected through wired or wireless structure.

Traditional middleware models such as Component Object Model (COM), Common Object Request Broker Architecture (CORBA), Enterprise Java Bean (EJB) and .NET are used to provide distributed object handling, communication and configuration services in distributed environment. These middleware platforms have the limitation of providing these services in a pre-determined environment. But there is a need for middleware platforms that should satisfy the broad and varying set of requirements arising from the users of the applications and in dynamically changing environments. Hence, it is concluded that the existing traditional middleware models are not sufficient to handle the dynamic changes in user needs and/or in execution environments, there is a need for adaptive middleware models.

Autonomic computing provides a platform and technologies for creating self-managing, self-tuning application servers (Sadjadi 2004). Applications server platforms are key elements of autonomic computing, providing a standardized service layer to support the development and deployment of distributed, adaptive applications. Application server technologies such as J2EE and .NET can be used to build server applications
that support various levels of quality-of-service, making them suitable platforms for high performance and reliable software systems. Adaptive applications remain challenging to construct due to the complexity of application server technologies and the fact that server applications operate in dynamic environments with variable request loads, fluctuating resource usage and unpredictable system faults.

The complexity of handling the requirements arising from dynamic operational environment is achieved by augmenting existing middleware platforms with adaptive capabilities. It allows to transparently building adaptive capabilities into the existing applications. The augmented middleware platforms are defined with components having adaptation and dynamic (re)configuration capabilities in response to changes in context, which allow avoiding the construction of large monolithic systems. The evolution of such middleware framework named as Adaptive Server Framework (ASF), support the development of adaptive behavior for server side components running on application servers (Gorton 2006).

The key design principle of ASF is to separate the implementation of adaptive behavior from the server application business logic. ASF implements adaptation behavior components and runs them in an adaptive engine, which interacts with the application server, monitors the runtime environment, analyzes collected data and changes the application behavior or the server configuration to fulfill the business goals specified in policies. This ASF is placed on top of J2EE middleware server. Current software architectures have been built for infrastructure based environments. These environments consist of powerful machines where increasing usage of memory, processor power and network bandwidth are solved by extending the required resources. These architectures with their resource requirements are not applicable to pervasive environments, where resource limitations, random
network connectivity behavior and heterogeneous issues have to be considered. Hence, architecture for the systems in these environments should be designed with the feature of pervasiveness and adaptiveness to make the applications adapt to available services and the devices’ capabilities.

The middleware solution for separating concerns in adaptive applications has been provided at the architectural level, which makes the requirements engineering to address the problem of dealing with adaptive requirements (Malek et al 2006). Requirements for adaptation are to be analyzed in the perspective of a system that makes a transition between satisfying different user requirements depending on the context or continue to satisfy the same user requirements in the face of changing context. This requirement engineering approach imposes clear separation of concerns between application requirements and adaptability requirements (Bencomo 2006). This leads to maintain clear traceability links between user requirements at the application level and the adaptability requirements identified by analysis and refinement of the user requirements. Hence middleware architecture should support this requirement engineering approach. The middleware models using reflection mechanism and aspect-oriented approach are evolved to meet the challenges in mobile and pervasive environments.

2.4.1 Reflective Middleware Models

Reflective architecture provides objects with the ability of dynamically changing their behavior by using their design information. The meta-level of the reflective architecture supervises the evolution of the base-level software system that is to be adapted. The meta-level system is composed of cooperating components; these components carry out the evolution against sudden and unexpected environmental changes on a reification of the design information specified in static and behavior models.
of the system to be adapted. The evolution takes place in two steps: first a meta-object, called evolutionary meta-object, plans a possible evolution against the detected event; and another meta-object, called consistency checker meta-object, validates the feasibility of the proposed plan before really evolving the system (Costa 2002). Meta-objects use the system design information to govern the evolution of the base-level system. Introducing reflective middleware approaches that extend the container and application programming interface based architectures enable the application and internal middleware structure to be exposed through meta-models (Blair 2001).

Prism-MW is an architectural middleware geared to distributed, mobile and pervasive environments (Malek et al 2006). The challenges incurred in architecting software systems for pervasive and mobile environment are (i) maintaining the efficiency of a system due to resource constraints such as limited CPU, memory and network bandwidth; (ii) predicting the run-time behavior for maintaining the properties like reliability and latency; (iii) managing the heterogeneity of hardware and software platforms; (iv) deploying a software on a centralized server with the facility of providing communication and coordination with a large number of heterogeneous hosts; (v) locating the services required for running the application software to complete its task; (vi) providing support for multiple architectural styles to create pervasive and mobile systems and (vii) making a system to function correctly due to limitations in computing platform.

Prism-MW addresses the above challenges and it is composed of three distinct layers, namely the virtual machine layer at the bottom, which allows the middleware to be deployed on heterogeneous platforms efficiently, architectural constructs layer, which leverages the abstraction facilities provided by the virtual machine that is placed on top of it and pervasive computing facilities to support the implementation of the architectural
constructs. Prism-MW supports architectural abstractions by providing classes for implementing each architectural element with methods for creating, manipulating and destroying the element and it enables direct mapping between architecture and its implementation. This middleware laid the foundation for incorporating variations in a software system.

Component technology has recently emerged as a promising approach for building highly adaptable software systems. Component based systems achieve adaptability as a result of characteristics of components include dynamic or independent configuration of the constituent components and dynamic deployment feature (Szyperski 1999). The other benefits of component technology include reusability, dynamic extensibility, understandability, and reduced development costs. It is applied at the application level on top of middleware infrastructures, which hide distribution and other non-functional concerns from component developers (Parlavantzas et al 2000). In order to address the adaptability, middleware should be built using component based architecture and should be reflective to manage the run-time changes in component configurations. Such component-based reflective middleware architecture is designed using the principles of (i) component frameworks, which is a set of components and collection of rules and contracts that govern the interaction of a set of components associated with a particular problem domain; (ii) component framework specific meta-interfaces to expose the implementation of component framework based subsystems in a controlled and principled way by maintaining information about current configuration and by applying it to perform inspection and adaptation and (iii) uniform component model to represent the architecture of the application and middleware framework.

OpenCOM component model represents Microsoft’s COM with richer reflective facilities and it is used for the construction of adaptive
software system (Clarke et al 2001). The fundamental concepts of OpenCOM are interfaces, receptacles and connections (bindings between interface and receptacles) (Surajbali et al 2007). An interface expresses a unit of service provision and a receptacle describes a unit of service requirement. OpenCOM deploys a standard runtime substrate that manages the creation and deletion of components and acts upon requests to connect and disconnect components. OpenCOM improves the basic reflective facilities of COM by providing meta-information about the interface types through IMetaEncapsulation interface, and this meta-information is used to support dynamic invocation of arbitrary interfaces. OpenCOM also supports interception at specified interfaces that insert wrapping behavior around method invocations and used to construct families of middleware. Each middleware is constructed as a set of configurable component frameworks, and it uses reflection to discover the current structure and behavior. Also, it enables selected changes at run-time. A component framework in OpenCOM is itself an OpenCOM component that maintains internal structure to implement its service functionality.

Middleware platforms must offer a high degree of configurability at deployment and at run time to satisfy the variations in user needs, deployable, and execution environments. The reflection, components and component frameworks are used to develop dynamically configurable middleware platforms. Components and component frameworks provide extendable structure and functionality. Reflection provides the essential support for run-time evolution and adaptation. Such dynamic reconfigurable middleware platforms should provide necessary support for systematic and automated checking of both functional and non-functional properties.

GRIDKit is an experimental reflective middleware for grid computing, which supports adaptive grid and pervasive applications (Grace et al 2006) (Bencomo 2006). It is one of the dynamically configurable
middleware models developed using OpenCOM component model. A key feature of GRIDKit is its ability to handle diversity or to take dynamic behavior to meet different application requirements in different environmental contexts. It is composed of component frameworks, where each framework is defined with set of components related to a particular domain of middleware behavior. It is composed of overlay framework, which provides multiple virtual network services required by various virtual frameworks placed on top of it. The various virtual frameworks are (i) interaction framework, which accepts multiple interaction type plug-ins such as Remote Procedure Call (RPC), publish-subscribe and group communication; (ii) resource framework, which accepts plug-ins to discover application services like Service Location Protocol (SLP), Universal Plug and Play (UPnP) and Salutation; (iii) resource management and resource monitoring frameworks, which are responsible for managing and monitoring resources and (iv) security framework that provides security for the frameworks.

Frameworks are also unit of adaptation and policies for configuration and reconfiguration associated with individual framework task.

Genie is a prototype for a development environment that offers a Domain Specification Language (DSL) for the specification, validation and generation of artifacts for OpenCOM based middleware platforms (Bencomo et al 2008). It allows the creation and validation of models that drive the lifecycle of the reflective middleware families. It allows generating source code, configuration files, results associated with model checking and validations, testing code and documentation. It offers a common modeling notation for all the models called OpenCOM DSL. The component and component framework configurations are specified using graphs. It models the component with offered, required and bind interfaces. Component
framework is described with export and required interfaces for internal components and to satisfy the requirements of the components respectively.

REFLEX middleware model is designed using reflective, self-adaptive, and component based architecture that can reduce the cost and effort required for evolving a system to cope with changing requirements (Pemberton 2008). The Object Request Broker (ORB) facilitates the deployment of adaptable, dynamically evolvable and distributed component-based software systems. The ORB’s design is outlined using a dispatch-delegate reflective proxy-based implementation. The REFLEX meta-interface provides an abstraction of the ORB’s dispatcher that allows adjustment of component implementations allowing for adapting changes in the functionality of the component. Components deployed through the ORB are referred to as RComponents, which internally consist of four core entities: the meta-level, dispatcher, external application interfaces and delegated processing behavior objects which are encapsulated inside a management container. It is the RComponent’s internal decoupling that allows flexibility in terms of interface, and behavioral component evolution. Reconfiguration is performed at the meta-level allowing adaptive change of the reflective dispatcher. RComponent bridges the integration with JavaBeans, EJB and ActiveX.

OpenORB is a reflective component middleware model, which is organized as a set of components, and dynamic adaptation is supported by means of computational reflection. The main elements of OpenORB base-level are interfaces, local bindings, components and capsule (Cacho et al 2006). A component is a unit of independent deployment defined with the composition of provider and required interfaces. Interfaces represent the access point of a component that is called ports. Each interface can export and import methods. Exported methods are those provided by a component
while imported methods are those required by a component. Bindings between interfaces are represented as components. There are two types of bindings: local bindings and distributed bindings. A local binding associates exported interfaces with imported interfaces. Distributed bindings are distributed components, which are composed of components bound by local bindings. A Capsule is a logical container of components that provides an API for loading and binding components.

The OpenORB meta-level defines the reflective mechanism that implements a causal connection with the base level (Saikoski et al 1999). In order to organize the meta-level, OpenORB proposes four meta-models: (i) encapsulation that exposes the implementation of an object. This meta-model can be used to monitor, to control the access to an object, its attributes and methods and also to make the changes in an object (ii) composition, which is responsible for providing the bindings graph of a component to insert, to remove and to replace components (iii) environment that exposes the execution environment of each interface including method invocations for servers and clients and (iv) resource, which is responsible for reifying and managing the resources used by each object.

Mobile applications are required to operate in environments in which the availability for resources and services may change significantly during system operation (Costa et al 2005). As a result, mobile applications need to be capable of adapting to these changes to offer the best possible level of service to their users. However, conventional middleware is limited in its capability of adapting to the environment changes and different user requirements. Computational reflection applied to middleware design is named as reflective middleware. Reflective Middleware for Mobile Computing (ReMMoC) combines reflective middleware and the Web Service Description Language (WSDL) programming model to provide a solution to
the problem of interoperation from mobile clients (Grace et al. 2003). ReMMoC is a configurable and reconfigurable reflective middleware that supports mobile application development and overcomes the heterogeneous properties of the mobile environment. ReMMoC uses OpenCOM as its underlying component technology, and it is built as a set of component frameworks. Using many component frameworks (e.g. as found in OpenORB) increases the size of the middleware implementation; extra management functionality for managing reconfiguration exhausts the constrained resources of a mobile device. Therefore, ReMMoC consists of only two component frameworks: (i) a binding framework for interoperation with mobile services implemented upon different middleware types and is configured by plugging in different binding type implementations such as Internet Inter-ORB Protocol (IIOP) Client, Publisher, Simple Object Access Protocol (SOAP) client and (ii) a service discovery framework for discovering services advertised by a range of service discovery protocols.

A generic engineering framework for adaptive mobile services was proposed in the FAMOUS – the flow modeling and rapid design for dynamically reconfigurable system project (Hveding 2005). This framework includes support for mechanisms required by mobile applications to adapt their services to the changes in underlying context at run-time and context models that allow applications to access update and reason about contextual information. To achieve this, the framework is constructed based on component frameworks, variability engineering, property modeling and generic reflective architectural middleware to handle adaptation. When changes occur in the context requiring adaptation, the middleware analyzes the meta-model to find the configuration of the components that best fits the current context. The application is then reconfigured to meet the new context requirements.
The dynamic adaptability model has been proposed in (Amano and Watanabe 1999) based on the states of runtime environments, adaptable procedures and adaptation strategies. The procedure invocation is changed dynamically depending on the states of runtime environments. This mechanism forms a meta-level architecture and realizes the dynamic adaptability in reflective way. LEAD++, an object-oriented reflective language has also been proposed in that work to realize the proposed model.

2.4.2 Aspect-Oriented Middleware Models

Software engineering moves towards the development of systems in terms of a set of interactions among detached components. The complex enterprise applications are defined with a large number of business functions. These business functions are entangled with non-functional concerns such as logging, data persistence, transaction management and security, and are scattered across the unrelated modules of a system. The various middleware architectural models such as Common Object Request Broker Architecture (CORBA), Distributed Component Object Model (DCOM) and J2EE with predefined services were considered as a solution to support design and analysis of non-functional concerns, which are used to reduce the complexity involved in the development of an enterprise system.

Achieving these non-functional requirements requires an appropriate separation of the system concerns in independent modules. These middleware frameworks did not provide the solution for separating these entangled concerns from the functional concerns (business logic). Using the principle of separation of concerns, a set of aspect-oriented middleware models, each built for a specific purpose or non-functional requirement of a system have been designed. The component models with the features of reusability, extensibility and adaptability can be achieved through the separation of concerns in independent modules. Extending object-oriented
programming with the dimension of concerns named as aspects is required to construct the system with the adaptability feature. Aspects allow the components in a system to change or evolve independently. Hence, aspect-oriented middleware has been considered as a promising and relatively successful paradigm to improve usability, extensibility, modularity and customization capabilities of middleware platforms (Colyer and Clement 2004, Bodkin et al 2003).

Aspect-Oriented Programming (AOP) technologies offer different aspect definition languages, weaving processes and aspect models to tackle the separation of concerns issue. Most of the aspect-oriented languages support model aspects to represent specific types of concerns such as concurrency, synchronization, distribution and security. Hence, Aspect-oriented (AO) frameworks were evolved to model components and aspects as separate entities, and to generate the composition of components and aspects at run-time, which offers better performance. These AO frameworks also support model domain specific aspects. Dynamic Aspect-Oriented Middleware Framework was designed with the feature of dynamically composing components with aspects and for defining different types of aspects namely environment-oriented, user-oriented, type-oriented and component-oriented (Pinto et al 2002). Environment-oriented is a single instance for a system such as persistence aspect; user-oriented aspect is shared among the components for a specific user to have multiple visual views (2D/3D); type-oriented is an aspect that is shared by the set of components that play the same role like aspect for providing different access control mechanisms depending on the component type; and component-oriented is an aspect per component instance.

The continuous evolution in the requirements of systems is realized by constructing open software systems. The Aspect Moderator is a framework
proposed in (Constantinides et al 2000) as an extension to the object model and designed to support the encapsulation of main functionality of an object along with its aspectual behavior. Proxy is used to register the aspect objects and it maintains the order for executing them. During method invocation proxy is responsible to intercept and manipulate the incoming message in order to invoke the aspectual behavior along with the called method.

A dynamic aspect oriented framework along with Extensible Markup Language (XML) based domain specific languages is used to define dynamic adaptable system named as ambient intelligent system. Adaptable applications can be viewed as the composition of components and aspects where the adaptation is localized inside aspects, and general purpose functionality is encapsulated inside components. The problem for producing dynamically adaptable software systems is making the application to be aware of all the changes that can happen in the environment and define strategies for reacting to all of them. The use of aspects allows extracting the behavior of the components that receive these change events and reuse this behavior in other components. It is highly desirable to model these behaviors as independent aspects in order to easily modify their implementation according to the changes in the application requirements.

An Ambient Intelligence (Aml) application is defined with a list of adaptation strategies to solve a concrete local problem such as the appearance and disappearance of devices in the environment, errors in communications, the reception of data not compatible with the application and posterior adaptation of this data, problems related to the management of the energy consumed by the device, the need to retrieve data located from other devices and demands for the adaptation of the application to the preferences of the user that is using the device (Fuentes and Jimenez 2005).
AOP technologies are integrated with the middleware models to enhance the features of extensibility and adaptability. Cohen (2007) proposed Shakeins, a programming construct to create AOP based middleware models. Shakeins are constructed by combining features of aspects with selected properties of generics and mixins. Shakein receives a class parameter and optional configuration parameters, and generates a new class which has the same type as the original, but with different data and code implementation. Like aspects, they can be used to modularize non-functional concerns without tangled and scattered code. Unlike traditional aspects, Shakeins preserve the object model, present better management of aspect scope, exhibit a more understandable and maintainable semantic model, and use configuration parameters to make changes on a class. Hence, Shakeins address two current issues of J2EE that are configuration of existing services and greater flexibility in their application.

ASPECTJ2EE framework was introduced in (Cohen 2007) by combining AOP and J2EE. It adopts semantics from Shakeins and from AspectJ. It is geared towards the generalized implementation of J2EE application servers and of applications within this framework. It can be smoothly integrated into J2EE implementations without breaking their architecture, which is achieved by applying deploy-time aspect weaving mechanism for the process of binding services to user applications in the J2EE application server. In the ASPECTJ2EE framework, Factories, which is a new language construct used for providing each class with complete control over its own instance-creation process. Different clients, based on their needs, can obtain instances of the same class decorated in a different manner using client-side factories. The class itself can choose to provide only instances augmented in some specific manner using supplier-side factories.
An Adaptive CORBA Template (ACT) framework is the enhancement of adaptive CORBA framework, which enables CORBA applications to support unanticipated adaptation at run time without the need to modify and recompile the application source code (Sadjadi 2004). The solution for unanticipated adaptation is confined to the CORBA framework. ACT implements interception and redirection inside the supporting middleware instead of the program code. In addition, ACT enables interoperation among otherwise incompatible adaptive CORBA frameworks. Dynamic AOP frameworks with the characteristics of applying adaptation dynamically, encapsulating adaptations and removing adaptation easily made them to be used for implementing the autonomic systems (Greenwood and Blair 2003). Aspectwerkz and Prose are some of the frameworks with the above mentioned characteristics used for dynamic adaptation.

An alternative way of handling complex business applications is using web services. Web services are emerging technologies for integrating heterogeneous applications. The internal services are interconnected with other external resources to form a virtual enterprise. This needs to standardize the external specification that is a combination of service interfaces and business protocols. This external specification will undergo changes due to variations in the external resources, which lead to a mismatch between internal service implementation and a standardized external specification. A framework and a tool for managing service adaptation are required to separate the adaptation logic from the business logic. Such separation helps to avoid the need of developing and maintaining several versions of a service implementation and isolates the adaptation logic in a single place.

The adaptation of service can be considered as a cross-cutting concern from the developer and project architecture. If the invoked service changes the interface or protocol, then all the composite services invoking it
will have to undergo analogous changes to interact with the new version of the invoked service. In order to accomplish this, an Aspect-Oriented Programming (AOP) approach is used to weave adaptation solutions into the different composite services that need to be modified. The AOP framework proposed by Knogdenfha et al (2006) consists of (i) a taxonomy of the different possible types of mismatch between external specification and service implementation; (ii) a repository of aspect-based templates to automate the task of handling mismatches and (iii) a tool to support template instantiation and its execution together with the service implementation.

A middleware platform for mobile environments has to cope with a wide diversity of challenges such as mobility of the users, devices carried by the user and applications based on the location and the environment, resource-awareness to make the applications to aware of the devices and their location on which they are running, resource-constraints devices in the mobile environment which are restricted to processor power, energy and memory, heterogeneous environments, distributed infrastructures and various protocols. Middleware layers placed between user clients and application servers were used to address the challenges stated in mobile environment (Kanoc 1999). A context management middleware was proposed in (Autunes da Rocha and Endler 2006) to efficiently handle the heterogeneous execution environment and the evolution in that environment. It uses context meta-information to improve a context-aware system's overall performance. Context management must also support model evolution without restarting, reconfiguring or redeploying applications and services. Since aspect-oriented middleware allows easy of use through declarative composition and extensibility of the service container by combining container-based middleware and reflective middleware, it is more appropriate to use aspect-oriented frameworks to address the issues in context management (Greenwood et al 2008). The
following are the aspect-oriented frameworks proposed in various research works.

The main functionalities of Mobility and Adaptation enabling Middleware (MADAM) framework are (i) detect context changes (ii) reason about the changes and make decisions about the adaptation to perform and (iii) implement the adaptation choices (Lombardo 2007). MADAM follows an architecture-centric approach to generate architecture variability models at runtime based on the reconfiguration of the component services associated with the implementation of the adaptation choices. MADAM framework supports adaptation through configuring the services according to the context, but not supporting the adaptation of dynamic functional changes in the mobile services.

MUSIC – Middleware for Self Adaptation in Ubiquitous and Service-Oriented Environment. It is an extension of the MADAM planning framework. It deals directly with service level agreement protocols supported by the services to negotiate the best quality of service for the user (Rouvoy et al 2008). This planning middleware evaluates discovered remote services as alternative configurations for the functionalities required by a mobile application. MADAM framework is based on dynamic configuration of component frameworks, whereas MUSIC planning framework supports seamless configuration of component framework based on both local and remote components and services. The solution for adapting unanticipated changes on the functionalities of the mobile service component is not stated in MUSIC framework.

U-MUSIC – Unanticipated MUSIC Framework proposed by Khan (2010) as the extension of MUSIC framework, which provides the solution to address unanticipated adaptation, which is not addressed by the MADAM and MUSIC frameworks. The adaptability of unanticipated changes is achieved
through the configuration of services at run-time, which reduces the performance of a system.

Jadabs, an adaptive pervasive middleware architecture was proposed in (Frei 2005) which supports the ubiquitous computing. It is the lightweight middleware architecture, which allows transparently adapting and extending application components with new requirements. It uses the combination of service-oriented architecture and dynamic aspect-oriented programming approach for dynamic adaptation and it can run on small devices like mobile phones. The underlying feature of this middleware is dynamic heterogeneous service composition, which enables an autonomous dynamic adaptation of running services. The dynamic service composition allows adding or removing devices and services using different resource descriptions available for devices, platforms and services. The services are matched against the device and platform using the service meta-data. Matching services can then be downloaded and activated automatically. In Jadabs middleware, aspect is used for performing service composition to dynamically adapt the services according to the context whereas it does not address the adaptation of adding or modifying the operations in the services.

A solution to improve the modularization using aspect-oriented approach was proposed in the paper (Aldrich 2011). In that work, they proposed the formal model for specifying the module that is named as an Open Module, which ensures that clients cannot be affected due to the changes in the internal implementation of a module. Also in their work, they defined TinyAspect construct, a minimal core language for describing the Open Module. The above mentioned work defined the aspect interface for allowing the external functions to access the interface functions, whereas it did not focus on using aspects for adapting dynamic changes. An Aspect-oriented framework for designing and developing the open concurrent object-
oriented systems with the features of reusability and adaptability objects was proposed in the works (Constantinides et al 2000, Constantinides et al 2001). The scheduling and synchronization techniques required for efficient way of accessing the concurrent objects were considered as aspects in the work (Constantinides 2000). Handling the dynamic changes using aspect-oriented approach was not focused in the above stated work. The work stated in (Sanen 2005) proposed the solution for dynamic adaptability using aspect-oriented paradigm by separating the roles of the objects from the core classes and representing them as aspects. The quality of the aspects are improved by designing them using the aspect-oriented design principles stated in (Wampler 2007), which are derived from object-oriented design principles.

The work (Brice 2009) proposed a solution for building dynamically adaptive systems using aspects to represent the anticipated dynamic variants in the system and validating the dynamic configuration using dynamic meta-models. But in that work, the aspect was not representing the dynamic unanticipated changes in the functions of the system. Finally, it is concluded that all the works discussed above did not focus on providing a general solution for adapting the dynamic changes in the requirements, which are not necessarily cross-cutting concerns using aspect-oriented approach.

2.4.3 Augmentation of Reflective Middleware with Aspect-Oriented Programming

Reflective middleware has been proposed as an effective way to enhance adaptability of component-oriented middleware architectures. To be effectively adaptable, the implementation of reflective middleware needs to be modular. The mobile and pervasive systems have required support for the conception of a minimal middleware, which imposed more stringent modularity requirements to the middleware design. The key problem is that fundamental mechanisms for decomposing reflective middleware
implementations such as object-oriented, suffer from not providing the proper means to achieve the required level of localizing reflection-specific concerns. Highly-adaptable implementations of such systems have often been shown as a challenge because many of the core reflective features typically crosscut the modular object-oriented decomposition of the middleware architecture. Some examples are the mechanisms dedicated to support causal connection, state recovery of objects and meta-objects and introspection about the binding process. These features have a broadly-scoped effect over the target component model, thereby both limiting the conception of a minimal middleware and reducing its modularity and adaptability.

The authors of the paper (Cacho et al 2006) presented a systematic investigation on aspect-oriented programming to improve modularity of typical reflection-specific crosscutting concerns. The reflective middleware approach has two main drawbacks such as reflective APIs are typically complex that impose a steep learning curve on developers and the introduction of unconstrained openness places too much expressive power in the hands of developers. To address these drawbacks, the application of Aspect-Oriented Programming (AOP) techniques to middleware has been suggested in (Surajbali et al 2007).

OpenCOM is the reflective adaptable middleware that supports runtime configuration or reconfiguration for loading or unloading the low level components. GRIDKit is a highly configurable reflective middleware platform that is built as a composed set of reconfigurable component frameworks. These are used to extend the basic OpenCOM programming model to support distribution. As these middleware models suffered from the limitations of reflective middleware, AOP is used to represent the concerns distributed across the OpenCOM.
AOP provides support for defining distributed dynamic aspects that can be dynamically deployed across a distributed system on the basis of pointcut expressions that are inherently distributed in nature and also support the composition of remote advices. These aspects are realized as OpenCOM components, and their composition is performed using OpenCOM connectors. Hence, augmenting the reflective middleware with an aspect orientation support layer for better modularization of services is provided by OpenCOM, a component model and GRIDKit, an OpenCOM based middleware. In this model, the components of the aspect oriented support layer placed on top of the OpenCOM reflective layer and the distributed aspect oriented services like aspect composition are placed over the GRIDKit middleware. This design approach shows that the adaptability provided by this model should be confined to the limitations of OpenCOM and GRIDKit frameworks.

The AspectOpenCOM is a reflective component model proposed in (Grace et al 2008), which provides a meta-object protocol capable of fine-grained adaptation of deployed aspects and it suffers from the limitations of meta-object protocol. TRAP is a programming model, which supports dynamic adaptation in the existing object-oriented programs transparently to produce a family of adaptable programs (Cohen 2007). TRAP/J employs a systematic approach to dynamic adaptation. It generates the required reflective infrastructure and weaves adaptive code into an existing program through reflective classes. Fractal aspect component model has been designed for aspects and components (Pessemier et al 2006). This model is mapped on the Fractal component model, which is a general reflective component model and it supports dynamic binding. It improves dynamic software adaptation by taking benefits from aspect oriented software development, component based software development and reflection.
2.5 ADAPTABILITY METRICS

The following are the discussions on the various research works carried out on determining the measures for representing the modularity and adaptability of the system.

The paper (Przyblek 2010) presented a comparative study on modularizing the systems using object-oriented and aspect-oriented approaches. In that work, the systems were evaluated using the Coupling between Object Classes (CBO) and Lack of Cohesion in Methods (LCOM), which are the general metrics used for assessing modularity of the object-oriented systems. These general metrics were refined and proposed as the metrics for assessing the modularity of the aspect-oriented system. The CBO metric principles stated in the above work are not specifying the coupling between aspects and classes. The semantic information shared between the elements of the source code of the classes was defined as the conceptual coupling between the classes, which was proposed as the coupling measure for object-oriented systems in (Poshyvanyk et al 2006). It is deduced that the semantic information shared between the classes and aspects using point-cut specification at the architectural level is to be considered.

The metrics for adaptability described below were proposed in (Subramanian and Chung 1999).

- Element Adaptability Index (EAI), where EAI = 1 for an adaptable element and EAI = 0 for a non-adaptable element.

- Architecture Adaptability Index (AAI) = EAI for all elements of architecture/Total number of elements.
• Software Adaptability Index (SAI) = AAI for all architecture of the software/Total number of architectures for that software.

These metrics were used to evaluate the adaptability at the architectural level. In the above metrics, the semantic coherence existing between the methods across the classes and the semantic coherence between the aspects and the methods of different classes were not considered even though they are required to evaluate the adaptability at the architectural level.

Making use of Chidamber and Kemerer (1991) metrics suite for object-oriented system for evaluating the aspect-oriented system was analyzed in the work proposed in (Zakaria and Hosny 2003). The Chidamber and Kemerer (C&K) metric suite includes Weighted Methods Per Class (WMC), Depth of Inheritance Tree (DIT), Number of Children (NOC), Lack of Cohesion of Methods (LOCM), Coupling Between Object Classes (CBO) and Response For a Class (RFC). Based on the effect of aspect-oriented approach on C&K metrics, it is deduced that the impact of coupling is used to create adaptability metrics.

Measuring the impact of aspect-oriented approach in maintainability using coupling metrics was proposed by (Rachel et al 2010a) in their work. The coupling metrics stated in that work are coupling on advice execution and number of degree diffusion point-cuts. These metrics are used to measure the fault-proneness in the aspect-oriented systems (Rachel et al 2010).

(Haupt and Mezini 2004) proposed micro measurements for dynamic AOP systems, which include the cost of dynamic (un)weaving, the cost of executing the method along with the advice and the cost of passing the
advised method’s parameters to an advice. It implies the performance measure of the dynamism involved in the AOP system at the execution level.

AOP framework to encapsulate the software measurement process without affecting the software under analysis was proposed by (Cazzola et al 2008). This framework can be extended to measure the adaptability efficiency of the system. The evaluation of changeability of an aspect-oriented system is computed through the impact of changes made to modules of the system. The work stated in (Avadhesh Kumar et al 2007) evaluated the change on modules occurred due to the syntax change in code through accessing the impact of changes in scope, data, functions and operation signatures of a class/aspect on other classes/aspects. Garg et al (2011) stated the Unified Modeling Language (UML) diagrams for aspect-oriented system as the special design to count the dependencies between aspects and classes and also between aspects and interfaces. These dependency counts can be used to measure the coupling between classes, aspects and interfaces.

2.6 SUMMARY

The concepts that are supporting this thesis in the context of software adaptation were described in this chapter. The discussion on the need of adaptation in a software system determines the objective of the research work stated in this thesis. The software adaptability techniques described in this chapter were considered as the core in the adaptability approach proposed in the present work. The summarization of the features of adaptive middleware models discussed in Section 2.4 is shown in Table 2.1. The features include the compositional, parameterization, aspect-oriented programming, reflection and interception approaches followed in achieving adaptability; the portability of the models and the type of changes adapted by those solutions. The features of these middleware models were utilized in the design of the adaptable middleware proposed in this thesis.
Table 2.1 Consolidated View of Adaptive Middleware Models

<table>
<thead>
<tr>
<th>Adaptive Middleware Models</th>
<th>Reconfiguration</th>
<th>Compositional</th>
<th>Parameterization</th>
<th>AOP</th>
<th>Reflection</th>
<th>Interception</th>
<th>Portability</th>
<th>Un anticipated Changes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prism-MW</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>OpenCOM</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REFLEX</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OpenORB</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ReMMoC</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>FAMOUS</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AspectModeator</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ASPECTJ2EE</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACT</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MADAM</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>MUSIC</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UMUSIC</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jadabs</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPENCOM + GRIDKit</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AspectOpenCOM</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>