UNIVERSITY OF SINDH

M.Phil. Thesis

POLICY-BASED CONTEXT-AWARE ARCHITECTURAL ADAPTATION IN PERVASIVE COMPUTING

THESIS SUBMITTED TOWARDS THE PARTIAL FULFILMENT OF THE REQUIREMENT OF THE UNIVERSITY OF SINDH, FOR THE AWARD OF MASTER OF PHILOSOPHY DEGREE IN INFORMATION TECHNOLOGY

GULSHER LAGHARI

Institute of Information and Communication Technology
University of Sindh, Jamshoro

2014

CERTIFICATE

This is to certify that the work present in this thesis entitled "Policy-based Context-aware Architectural Adaptation in Pervasive Computing" has been carried out by Gulsher Laghari under our supervision. The work is genuine, original and, in our opinion, suitable for submission to the University of Sindh for the award of degree of Master of Philosophy in Information Technology.

SUPERVISOR

_____

Dr. Lachhman Das Dhomeja
Professor
Institute of Information and Communication Technology
University of Sindh, Jamshoro Pakistan

CO-SUPERVISOR

_____

Dr. Yasir Arfat Malkani
Assistant Professor
Institute of Mathematics & Computer Science
University of Sindh, Jamshoro Pakistan

DEDICATION

I dedicate

this thesis

to my whole family,

specially my beloved

mother.

ACKNOWLEDGEMENTS

First of all, I would like to show my greatest appreciation and thank to my supervisor Prof. Lachhman Das Dhomeja for his continuous support, guidance, care and patience during the course of this work. The good advice and support of my co-supervisor Dr. Yasir Arfat Malkani also deserves special mention here that has ever been invaluable to me. I would like to express my sincere gratitude to him for his patience, motivation and valuable feedback. This thesis would not have been possible without the help and support of both of them.

I am also thankful to all of my teachers, friends and people who have always remained supportive to me.

Last but not least, I am highly indebted to my family and in-laws for their love, care, patience and being with me through all the ups and downs of life.

DECLARATION

I hereby declare that the work present in this thesis has been carried out by me and that this thesis has not been and will not be, submitted in whole or in part to another University for the award of degree of Master of Philosophy in Information Technology or any other degree.

CONTENTS

LIST OF FIGURES

GLOSSARY

ADL	Architecture Description Language
AEM	Architectural Evolution Manager
CHAM	Chemical Abstract Machine
ECA	Event-Condition-Action
JESS	Java Expert System Shell
LBMD	Location Based Message Delivery
PCAA	Policy-based Context-aware Architectural Adaptation
SNB	Smart Notice Board

ABSTRACT

The primary goal of pervasive computing is to support user tasks, satisfy user needs and enrich user experience with minimal or no user distraction. Context-awareness in general and context-aware adaptation in particular is central to achieving this goal. Context-aware adaptation is a process of obtaining contextual information, reasoning about it and adapting the application.

The main argument of our thesis is that in existing adaptation approaches, various concerns involved in adaptation process – adaptation policies and adaptation mechanisms are tightly coupled with an application being adapted, making applications difficult to build and modify at runtime. We address this issue and propose our policy and architecture centric approach to context-aware adaptation in which both adaptation policies and adaptation mechanisms are separate and external to the application being adapted. In particular, adaptation policies are high-level declarative Event-Condition-Action (ECA) rules, which are strongly decoupled from rest of the application code and dynamically modifiable.

The thesis provides design goals of Policy-based Context-aware Architectural Adaptation (PCAA) infrastructure, discusses its main design elements and implementation. The PCAA infrastructure allows developing and executing context-aware adaptive applications at software architectural level and using ECA policies. The infrastructure supports specification of application in a small configuration language, initialization of application from its specification and encapsulates compositional adaptation mechanism to adapt the application dynamically. We, finally, evaluate performance of PCAA infrastructure and support for dynamic modifiability of adaptation policies.

CHAPTER 1

INTRODUCTION

In this chapter we provide motivation and contributions of this thesis and at the end of the chapter, we present structure of the rest of the thesis.

1.1 MOTIVATION

In 1991, Mark Weiser in his seminal paper (Weiser September 1991) introduced the notion of pervasive computing in which he predicted that computing will move beyond desktop and become ubiquitous and invisible to the user. Satyanarayanan (Satyanarayanan 2001) attributes the invisibility as "minimal user distractions". For pervasive computing applications to be able to perform user tasks with minimal user distractions, they need to adapt themselves in response to context. This makes context-aware adaptation as a fundamental requirement for many pervasive computing systems.

In pervasive computing environments, context-aware adaptation is initiated by a particular context event or a set of context events with an aim to satisfy user needs and preferences or to enrich user experience. This requires that the applications monitor their environment (contexts), reason upon context changes (adaptation policies) and adapt accordingly. There exist a number of approaches to context-aware adaptation, in which adaptation support is provided in the form of programming languages, middleware and software architectures (detailed discussion of this is presented in Section 2.3 and Section 2.4). While adaptation approaches based on these categories have contributed towards this goal, our literature survey reveals that software architecture based approach is more promising, as it provides a clean separation of adaptation support from an application being adapted, and it operates at a higher level of abstraction—software architecture level.

Existing architectural adaptation approaches have received more focus on architectural reconfiguration (i-e. specific methods, technologies, tool support, etc.), while a little attention has been paid on another important facet of context-aware adaptation—adaptation policies. Adaptation policy support in existing approaches is limited in that (1) the adaptation concerns are tightly bound to application code and (2) dynamically un-modifiable, thus making applications difficult to build and modify at runtime.

We address aforementioned limitations and present our approach to context-aware adaptation by developing mechanisms and supporting toolset. The core of our approach is the use of policies and software architectures for development and execution of adaptive context-aware applications. In our approach, application is specified as a configuration of software components and software connectors. The adaptation concerns (when and how an application adapts) are specified as Event-Condition-Action (ECA) policies. An ECA policy subscribes to a context event. When context event occurs and the condition is true, the policy is enforced. The aim of our work is to have a modular and flexible approach for development and execution of adaptive context-aware applications. Towards this end, our approach is based on software architectures and adaptation policies by following separation of concerns principle, in which all concerns involved in adaptive context-aware application (adaptation policies, adaptation mechanisms) are separate and external to an application being adapted. Proposed approach requires writing a configuration code (i-e. expression of an application at software architecture level) and then running it. Once the application is running, adaptation concerns expressed as ECA polices are later added to the application. In particular, ECA policies are dynamically modifiable. Separation of concerns and the support for dynamic modifiability of adaption policies provide ease of development and support dynamic programmability of applications, which is an essential requirement for applications running in pervasive computing environments.

1.4 CONTRIBUTIONS OF THE THESIS

In this thesis, we present a Policy-based Context-aware Architectural Adaptation (PCAA) infrastructure for development and execution of adaptive context-aware applications running in pervasive computing environments. The contributions of the thesis can be described in terms of the features of the PCAA infrastructure, which include:

- **Software architectural adaptation support:** This includes design and implementation of runtime support for application initialization and its adaptation. Application initialization involves transforming an initial software architecture description of the application into running system. Adaptation support involves reconfiguring the architecture, thereby adapting the running application.

- **Dynamic programmability of context-aware applications:** Dynamic programmability of applications is achieved with support of dynamic modifiability of adaptation policies. Adaptation policies are specified separately of the application configuration code and can be modified dynamically.

- **Separation of concerns:** In our approach, all the adaption concerns involved in adaptation process (i-e. application being adapted, adaptation mechanisms and adaptation policies) are handled separately from each other. This greatly contributes towards reducing complexity involved in the development of adaptive context-aware applications.

1.5 STRUCTURE OF THE THESIS

Chapter 2 presents the definition of context, elaborates the adaptation process which involves three sub phases: context monitoring, adaptation policies and adaptation acting. It then discusses two types of adaptations: Parameter adaptation and Compositional adaptation along with discussion on approaches to achieving compositional adaptation. Finally, it provides the review of state-of-the-art architectural adaptation approaches.

Chapter 3 is organized as follows: Section 3.1 provides the design goals of the Policy-based Context-aware Architectural Adaptation (PCAA) infrastructure. Section 3.2 introduces the PCAA infrastructure. Section 3.3 describes the main design elements of PCAA infrastructure. Section 3.4 describes how does PCAA infrastructure work and finally Section 3.5 discusses the capabilities and limitations of PCAA infrastructure.

Chapter 4 presents the high level architecture of the system and describes implementation of each design element of the PCAA infrastructure. It provides the syntax and description of commands present in small configuration language that we have developed as part of this work. Finally, the chapter ends with discussion on PCAA component model.

Chapter 5 presents the design, development and execution of some hypothetical example adaptive applications using PCAA infrastructure. It also discusses and presents the policy specifications for the scenarios where these hypothetical applications need to adapt at runtime in response to change in their contexts. Finally, it provides the evaluation of the thesis.

Chapter 6 provides the conclusions of this thesis and outlines some future research directions.

CHAPTER 2

BACKGROUND AND RELATED WORK

In this chapter, we present the definition of context, elaborate the adaptation process, discuss adaptation types and explore the approaches to achieving compositional adaptation. Since our context-aware adaptation approach is based on software architectures, this chapter finally reviews state-of-the-art systems focusing on software architecture-based adaptation.

2.1 DEFINITION OF CONTEXT

While many researchers (Schilit, Adams et al. 1994, Brown, Bovey et al. 1997, Chen and Kotz 2000, Chalmers, Dulay et al. 2004) have defined context, we take the definition of context given by (Abowd, Dey et al. 1999), which is widely accepted. It defines context as:

"Any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the applications themselves".

Based on this definition, contextual information can include any information that characterizes the situation of a participant in an interaction, be that resource variation, mobility aspects, user location, preferences of the user, user activity, lighting, noise level, temperature, etc.

2.2 ADAPTATION PROCESS

Adaptive context-aware applications are required to monitor their environment to acquire contextual information, reason about context changes and adapt

accordingly. Adaptation process (Figure 2.1) comprises of following three sub-phases.

- **Context monitoring:** In this sub-phase, contextual information (such as environmental or user context information) is acquired through sensors installed in the environment. This information is further translated into high level context events which may initiate adaptation process.

- **Adaptation policies:** In this sub-phase, adaptation concerns (such as when and how to adapt the application) are specified. This may involve making decisions about what adaptation actions to execute, in response to changes in contextual information received from context monitoring sub-phase.

- **Adaptation Acting:** In this sub-phase adaptation decisions are implemented using suitable adaptation mechanisms (such as parametric adaptation, reconfiguration or code mobility, etc.) to adapt the application.

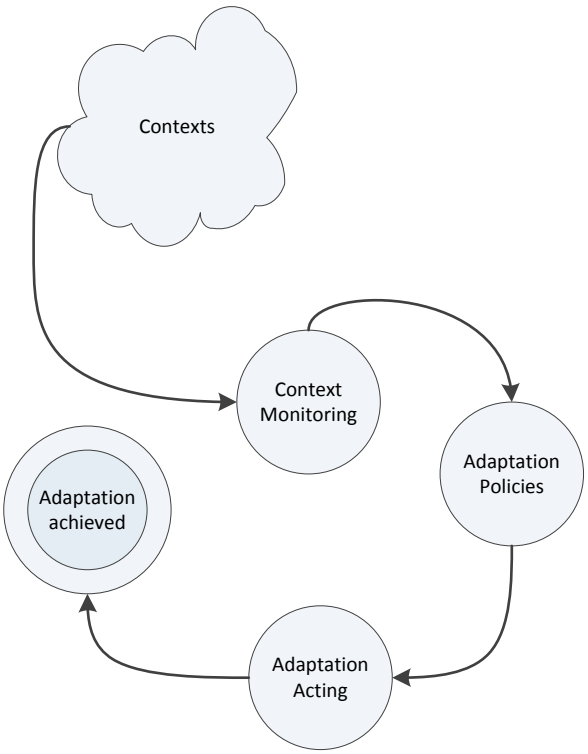

Figure 2.1 Adaptation process

2.3 ADAPTATION TYPES

In the literature (McKinley, Sadjadi et al. 2004, Fox and Clarke 2009), two approaches have been examined and employed to realize dynamic adaptation: parameter adaptation and compositional adaptation. In parametric adaptation, an application is fine-tuned by adjusting parameter values. For example, decreasing or increasing the sound of a music service by adjusting volume property, at the time the user is having a conversation on the telephone. This method of adapting an application has been extensively employed in various approaches. The recent approaches making use of this adaptation method to achieve dynamic application adaptation include (Salber, Dey et al. 1999, Sousaand and Garlan 2002, David and Ledoux 2003, Dhomeja 2011, Floch, Frà et al. 2013). The parametric adaptation approach is limited in a sense that new behaviors / algorithms, unforeseen during application development, cannot be adopted. It only allows adjusting the values of application properties or switching between existing algorithms to adapt the behavior of the application (McKinley, Sadjadi et al. 2004). Unlike parametric adaptation, in compositional adaptation, the application is reconfigured by modifying architectural topology of application. Since structural parts of an application are decoupled, it allows its structural parts to be added, detached, or replaced. As new behaviors and algorithms, unforeseen at application design time, can dynamically be adopted in the application, the compositional adaptation approach is more flexible than parametric adaptation. Different terms are interchangeably used for compositional adaptation such as reconfiguration, structural adaptation, and application code adaptation.

A number of approaches have been proposed in the literature to achieving context-aware adaptation based on compositional adaptation. A survey of these approaches can be found in (Aksit and Choukair 2003, McKinley, Sadjadi et al. 2004, Mukhija 2007). These can be classified as approaches providing programming language features, approaches based on middleware and approaches exploiting software architectures (Figure 2.2). Adaptation approach based on software architecture is more promising, as it provides a clean separation of adaptation support from an application being adapted. Also it operates at a greater level of abstraction—software architecture level).

The architecture of a software system represents the system as an organization of computational elements (components) and their interconnection (connectors) (Shaw and Garlan 1996). In software architectural adaptation approaches, the software architecture of an application is used to reason about and make changes in the application. The software architecture is kept and deployed along with application and the adaptation is carried out by making changes in software architecture of the application, these changes are also reified in the running application. Since architectural adaptation approach separates and externalizes adaptation mechanisms from application code and operates at software architectural level, it allows the developers to focus on system structure than a set of program statements (Oreizy, Medvidovic et al. 1998). This contributes to easing development and modification of adaptive applications.

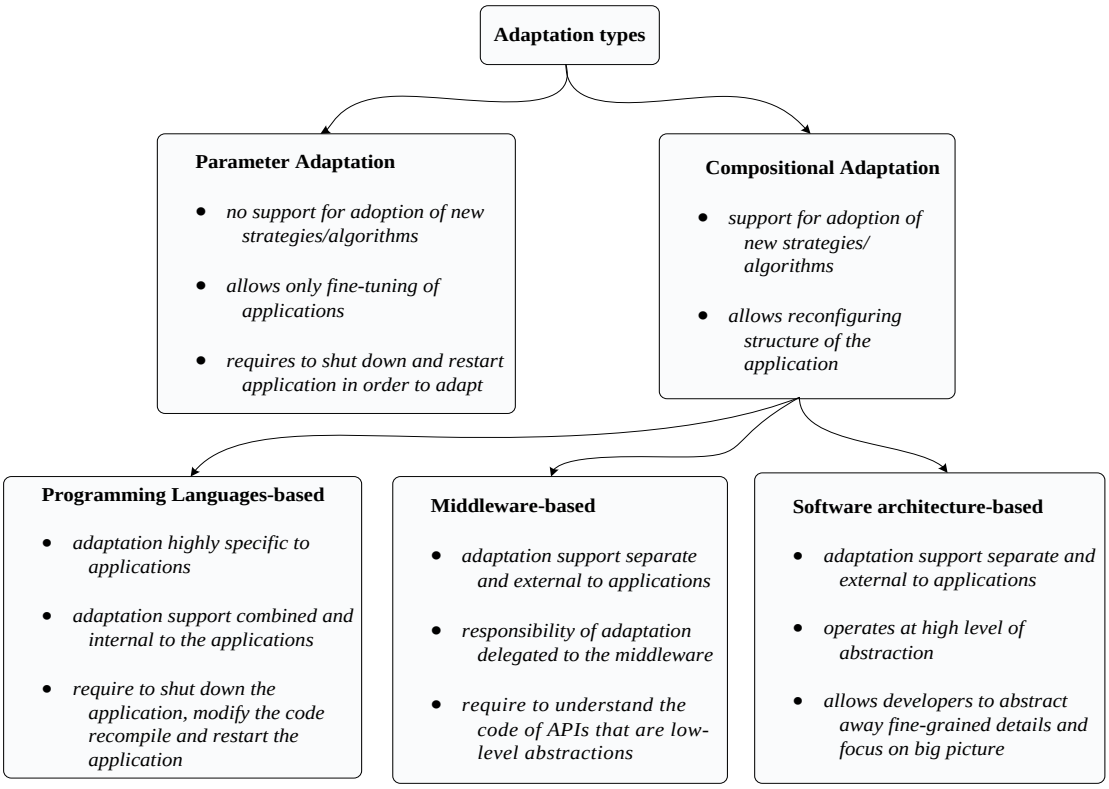

Figure 2.2 Adaptation types and approaches

2.4 APPROACHES TO REALIZING COMPOSITIONAL ADAPTATION

There exists a large body of research supporting compositional adaptation and the approaches providing adaptation support in the form of (1) programming languages, (2) middleware and (3) software architectures. Programming

languages supporting compositional adaptation include CLOS and Python. Later, ContextL (Costanza and Hirschfeld 2005), ContextPy (Schubert 2008) and PyContext (Löwis, Denker et al. 2007). Other languages supporting compositional adaptation include ContextS (Hirschfeld, Costanza et al. 2008), ContextR (Schmidt 2008) and ContextJS that extend SmallTalk, Ruby and JavaScript respectively. There is another class of programming languages that extend a Java programming language to support compositional adaptation. These include Open Java (Tatsubori, Chiba et al. 2000), R-Java (de Oliveira Guimarães 1998), Handi-Wrap (Baker and Hsieh 2002), Adaptive Java (Kasten, McKinley et al. 2002), ContextJ (Appeltauer, Hirschfeld et al. 2011), ContextAJ (Appeltauer, Hirschfeld et al. 2008). A detailed survey of these languages can be found in (McKinley, Sadjadi et al. 2004, Appeltauer, Hirschfeld et al. 2009). In approaches providing programming language features, the adaptation mechanisms are very specific to applications and strongly coupled with application source code. Moreover, no attention has been paid on another important component of adaptive applications called adaptation policies, in that the policies are tightly bound with application code, hence cannot be modified dynamically. These limitations of language-based approaches pose inflexibility in the sense that context-aware applications are both difficult to write and modify dynamically.

Middleware / runtime systems such as (Hallsteinsen, Floch et al. 2005, Gjørven, Eliassen et al. 2006, Mukhija 2007, Dhomeja 2011) offer an alternate solution to runtime adaptation, where adaptation support is delegated to middleware, hence separate and external to the application. This separation provides application transparent adaptation and contributes towards reducing development efforts. However, these solutions provide APIs to code applications that are low-level abstractions, requiring a fair amount of system knowledge to code the applications. A detailed survey and discussion on these approaches is provided in (McKinley, Sadjadi et al. 2004, Mukhija 2007).

Another approach used is software architecture based which exploits software architectures to achieve dynamic adaptation. As in middleware based approaches, this also separates and externalizes adaptation support from the

application code. Early dynamic architectural adaptation approaches include CHAM (Inverardi and Wolf 1995), graph grammars (Le Métayer 1998) and architectural description language (ADL) based (e.g. Rapide (Luckham and Vera 1995), Darwin (Magee and Kramer 1996), Dynamic Wright (Allen, Douence et al. 1998)). They were not widely accepted owing to two main reasons: (1) they lacked associated tool support and (2) dynamic adaptation support was limited which only allowed, for instance, replicating existing components a number of times (Oreizy, Medvidovic et al. 2008). Later adaptation approaches based on software architectures better addressed these limitations with tool support. The prominent ones include (Oreizy, Medvidovic et al. 1998, Oreizy and Taylor 1998, Dashofy, Hoek et al. 2002, Garlan, Cheng et al. 2004).

2.5 STATE-OF-THE-ART

(Oreizy, Medvidovic et al. 1998) present an approach to evolving software at runtime using software architectures. The software system is described as configuration of software components and connectors. The architectural style is event-based and layered that uses connectors to mediate communication between components. The style is called C2-style (Taylor, Medvidovic et al. 1995). They use software architectural model of software to reason about and make changes in the software at runtime. The model is explicitly maintained and deployed along with software system implementation and causally connected to it. The model contains description of components, connectors, their interconnection and their mapping to implementation units. The system is evolved through modifying architectural model. The modifications are achieved by applying architectural changes. The architectural changes may be adding a software component (attaching new behaviour), removing software component (excluding existing behaviour), replacing old components with other new component (modifying existing behaviour) or making structural changes to rearrange the composition of components and connectors. The tools are provided to introduce architectural changes. Agro allows making changes as graphical manipulations. Text based modification commands can be issued through ArchShell. While, Extension Wizard allows to execute modification scripts. The work is an initial attempt towards runtime software adaptation using software architectures; however it

does not address the specification and dynamic management of adaptation policies.

(Oreizy, Gorlick et al. 1999) present an architecture-centric approach to runtime software self-adaptation. In this, self- adaptation process comprises of adaptation management and evolution management. Adaptation management involves monitoring the running application, application operating environment and planning the changes that need to be made to the running application. Evolution management includes mechanisms for runtime software adaptation through software architectures as described in (Oreizy, Medvidovic et al. 1998). The approach sketches a basic framework for runtime software self-adaptation at software architectural-level. However, it does not provide details on specifying adaptation concerns and modifying them dynamically.

(Dashofy, Hoek et al. 2002) present an approach to self-healing software using software architectures. The software architecture is specified in xADL (Dashofy, Hoek et al. 2002) that is an extensible architectural description language defined as a set of XML schemas. The detailed information about xADL is available at http://isr.uci.edu/projects/xarchuci. The application is initialized by loading software components and interconnecting them based on its architectural description. The application is monitored for any faults and repaired while executing. Application repair, which is an architectural difference between initial architecture and architecture after repair, is represented as xADL schema. The repair is also called "diff" and is applied to the application to repairing it. This approach focuses more on system repair based on software architecture than addressing adaptation concerns (adaptation policies).

PBAAM (Georgas and Taylor 2008, Georgas and Taylor 2009) is a policy-based approach to architectural adaptation management in robotics domain. The adaptive behaviour (capturing what actions to perform in response to events indicating them) is decoupled from software and is expressed as condition-action rules. Third party system called Java Expert System Shell (JESS) (Hill 2003) has been used for expression and management of rules. The rules are specified in xADL (Dashofy, Hoek et al. 2005) and consist of observations and responses. The observations represent system information while

responses indicate system modifications. As adaptive behavior is specified in xADL, which is based on XML schema, it requires a lot of code to specify polices, hence polices are difficult to read and debug. Also the approach does not consider adaption in response to contexts external to the application.

Rainbow (Garlan, Cheng et al. 2004) is a framework, which provides a reusable infrastructure to support self-adaptation using software architectures. It uses architectural model of the application to reason about and make modifications in the running application. The model captures constraints placed on the application, and upon constraint violations, adaptation strategies are invoked to adapt the running application. A constraint may require that a response time to a client's request shall always be less than some threshold. A repair strategy may be applied to adapt the application, if the response time increases from threshold. The infrastructure requires the developers to write adaption operators that the system invokes at runtime to adapt the application. The developers then write adaption strategies which encapsulate system properties, their constraints and adaption operators when constraints are violated. As it requires pre-defining all adaption operators, unplanned adaptations cannot be dealt with using Rainbow, which constraints dynamic modifiability of adaptation strategies.

ACCADA (Gui, Florio et al. 2011) framework supports runtime component composition. The framework consists of five modules, Event Monitor, Structural Modeler, Context-specific Modeler, Context Reasoner and Adaptation Actuator. Event Monitor observes properties of the running system. Structural Modeler deals with knowledge about functional constraints and Context-specific Modeler deals with knowledge about context specific constraints (adaptation concerns). Context Reasoner selects the matching Context-specific Modeler based on current context. Adaptation Actuator carries out system adaptation. The adaptation process involves verification of structural violations of all the installed components in the system and context-specific violations and taking the adaptation actions to correcting the system. Adaptation action taken may trigger another round of adaptation process and so on. This design feature degrades system performance as number of components increase.

Also the adaptation concerns are not expressed at declarative level rather specified by writing code for Context-specific modelers. Transformer (Gui, De Florio et al. 2013) framework is very similar to ACCADA framework.

In summary, existing architectural adaptation approaches have received more focus on architectural reconfiguration (i.e. specific methods, technologies, tool support, etc.), while a little attention has been paid on another important facet of context-aware adaptation (i.e. adaptation policies). The support in existing architecture-based approaches is limited in that the adaptation concerns are tightly bound to application code, if not tightly bound, the adaptation concerns cannot be modified dynamically. We address these limitations in existing approaches based on software architectures and present a policy-based context-aware adaptation approach based on software architectures targeting pervasive computing environments.

2.6 SUMMARY

Various researchers have provided the definition of context but a broader definition of context has been given by Dey and so we have adopted his definition. In this chapter we have elaborated the adaptation process having three phases (1) Context monitoring and processing, (2) Adaptation policies and (3) Adaptation acting. We then discussed the two adaptation types: parameter adaptation and compositional adaptation. Parameter adaption involves fine tuning the application by modifying its parameters. Compositional adaptation involves reconfiguring the structural parts of the application and allows adopting new strategies and algorithms to unforeseen during design time. This chapter also explored various approaches to achieving compositional adaptation such as programming language based, middleware-based and software architecture based. This chapter also discussed the strengths of software architecture-based approaches as they externalize the adaptation mechanisms from application code and operate at higher level of abstraction. Finally the chapter provided discussion on state-of-the-art architectural adaptation approaches. In summary, we build our policy based approach on software architectural adaptation to develop and execute adaptive context-aware applications.

CHAPTER 3

PCAA INFRASTRUCTURE

In this chapter, we present the details of the proposed approach. First, we discuss the design goals and the main design elements of the Policy-based Context-aware Architectural Adaptation (PCAA) infrastructure. We then describe the working of PCAA infrastructure and finally discuss the capabilities and limitations of PCAA infrastructure.

3.1 DESIGN GOALS OF PCAA INFRASTRUCTURE

The design of the PCAA infrastructure is characterized by making all the phases (elements) in adaptation process separate and external to each other following the software engineering's separation of concerns principle. Application development is software architecture based with support of dynamic modification of adaptation policies.

3.1.1 SEPARATION OF CONCERNS

We treat all the adaption concerns involved in adaptation process (i.e. application being adapted, adaptation mechanisms and adaptation policies) separately from each other.

In our approach to developing and executing adaptive context-aware applications at software architectural level, an application is developed as a composition of independent software components. Each software component encapsulates a functional behaviour of an application and may require services of other components to accomplish the task. A component does not contain any code for adaptation concerns. An application is expressed at software architectural level by writing configuration code. Application is initialized and adapted by *reusable reconfiguration management component.* The adaptation

mechanism (software architectural reconfiguration) is a part of the *reusable reconfiguration management component*. Adaptation concerns are addressed and managed by *policy system* as adaptation policies. Adaptation policies are separately specified and added to the application dynamically, which means adaptation policies can be added to the application at runtime while it is running without stopping and restarting it.

This *separation of concerns* reduces complexity involved in development of adaptive context-aware applications and lets the developers focus only on one of these concerns at various levels of life cycle of an application. For example, this helps developers focus on main aspects of application core business concerns when developing application components while paying no attention on adaptation concerns.

3.1.2 SOFTWARE ARCHITECTURE BASED ADAPTATION

The application development in our approach is based on software architecture. The architecture of a software system represents the system as an organization of computational elements (components) and their interconnection (connectors) (Shaw and Garlan 1996, Garlan 2000). Traditionally, software architecture has been described as box-and-line diagrams in which software architecture is represented as a graph of interconnected nodes. The nodes represent computational components (processing elements, data stores etc.) and the edges (arcs) represent pathways of interaction between components (Garlan and Schmerl 2002). The shortcomings in traditional box-and-line diagrams include (1) the lines in between the nodes do not clearly show the type of interaction between the node components and leaves the developer to her intuition to make inference (2) when the system is implemented, it is difficult to tell that system implementation truly conforms to its initial architecture. Software architecture views software construction as configuration of components (encapsulating system's functionality) and connectors (regulating interactions among components) (Taylor, Medvidovic et al. 2009). Describing system as a gross organization of interacting components holds promise to ensure that system satisfies the requirements in terms of performance, reliability, portability, scalability, and interoperability (Garlan 2000). Software architectures can

provide a basis for runtime software adaptation by focusing, largely, on system structure to reason about and make changes in the software (Oreizy, Medvidovic et al. 1998, Oreizy and Taylor 1998). Also, adaptation support at software architectural level offers great flexibility to reconfigure software systems as components are arranged in a loosely coupled fashion which allows modifying system structure by rearranging them (Dashofy, Hoek et al. 2002).

Similar to approaches based on middleware, adaptation approach based on software architectures also separates and makes adaptation support external to the application code. In software architectural adaptation approaches, software architecture of the running application is used to adapt the executing application dynamically. The description of software architecture the application is kept alongside the running application. The application is adapted dynamically by manipulating software architecture (such as adding, removing or replacing components in the architecture). The changes made in the architecture are then reified in the running application. Besides the separation of adaptation support from the application, the software architecture-centric adaptation approach works at a greater detail abstraction, allowing the developer to view the software as a set of interconnected components rather than a set of program statements. This provides more abstraction by eliminating fine-grained details and focusing on components, their interconnection and runtime change (Oreizy, Medvidovic et al. 1998). This suggests that architecture-based adaptation is more flexible and provides abstraction to achieve the adaptation as performing architectural actions, such as addition of new software components in the system to achieve new application behavior, or replacement of components for modifying the application behavior.

Since the approach is based on software architecture, developing application requires assembling together coarse-grained software components (computational components and connectors), resulting in reduced development efforts.

3.1.3 DYNAMIC MODIFICATION OF ADAPTATION POLICIES

An adaptation policy encapsulates adaptation decision logic, i.e. containing information as what adaptation actions to perform, when to apply the adaptation actions to adapt the application and under what conditions to apply the actions. Following *separation of concerns* principle, offers great flexibility in dynamically modifying adaptation policies. Adaptation policies are managed by *policy system* and it makes it possible to dynamically modify adaptation policies such as to add a new policy, remove or modify an existing policy.

Dynamic modification of adaptation policies is desirable for applications running in pervasive computing environments since environmental conditions (such as resource variability) and user preferences or needs may change over time that require change in application behaviour at runtime. This also allows adding new adaptation behaviors that were not predicted or foreseen at the time of application development.

3.2 INTRODUCTION TO PCAA INFRASTRUCTURE

PCAA infrastructure is the development and execution infrastructure for developing and executing architecture-based adaptive context-aware applications. While developing these applications, the adaptation process involved comprises three sub-phases. The first phase is monitoring contexts so that contextual information is obtained and sent in the form of context events. In second phase adaptation decisions (that is making decisions on when and how an application is adapted) in response to changes in application contexts are defined. Finally, in the third phase adaptation acting is done using an appropriate adaptation mechanism by implementing adaptation decisions made in second phase.

In PCAA infrastructure, all these three sub-phases of adaptation process are handled separately. Software components only encapsulate main business logic and perform required functionality. They do not contain any code for adaptation decision logic and code to obtain contextual information. By taking adaptation decision logic out of the component boundary frees developers from

focusing on adaptation concerns and lets them concentrate only on core business logic while writing software components. Application is described as initial software architecture of the application by writing configuration code, which specifies the components and connectors used in the application and their bindings. Reusable reconfiguration management component is a sub-element of PCAA infrastructure that deals with application execution and carries out application adaptation. Application is initialized and loaded from the description of its initial software architecture. When the application is loaded, reusable reconfiguration management component also maintains an in-memory model of software architecture of the application. It contains references to executing components and always represents current architecture of the application. It evolves over time whenever application is adapted in response to changing contexts. Adaptation involves reconfiguring this model. The changes in model are enacted in the running application.

Adaptation decision logic is defined as specification of high-level declarative Event-Condition-Action (ECA) adaptation policies, which subscribe to a specific context event or a set of context events and encapsulate specifications for adaptation actions as architectural changes. When context event occurs and the condition is true, an appropriate policy will be triggered and executed. This will cause reusable reconfiguration management component to make an architectural change in in-memory architectural model, which is then enacted in the running system. The ECA policies are external to application and are separately specified (independently of configuration code of the application), and independently and dynamically managed (added to and removed from the system dynamically at any time throughout the life cycle of application). This provides a clean separation of concerns between adaptation policies and other aspects of architecture-centric adaptation (i.e. adaptation mechanisms, adaptation policies and application being adapted are all separate and external to each other). This separation of concerns reduces complexity involved in application development and supports dynamic programmability of applications. Dynamic modifiability of adaptation policies is an important feature, which is needed to meet the dynamic nature of pervasive computing environments.

In summary, the steps involved in application initialization and adaptation in PCAA infrastructure are following:

Application initialization

1. Application is initialized and loaded from description of its initial software architecture.
2. In-memory model of the software architecture of the application is maintained.

Application adaptation

1. Contextual information, in the form of high level context events, is sent from Context Simulators.
2. Context event(s) trigger adaptation policies that have subscribed to the event(s).
3. Adaptation actions of the triggered policies will be executed if the conditions are met. These actions are, in fact, architectural actions which are sent to reusable reconfiguration management component.
4. The reusable reconfiguration management component invokes the architectural actions which make changes in in-memory model of the software architecture. The changes in in-memory model are also reified into the running application that changes behaviour of the application. This is how the application is adapted in response to change in application contexts.

The steps for application initialization and adaptation are also shown as a sequence diagram in upper and lower halves in Figure 3.1 respectively.

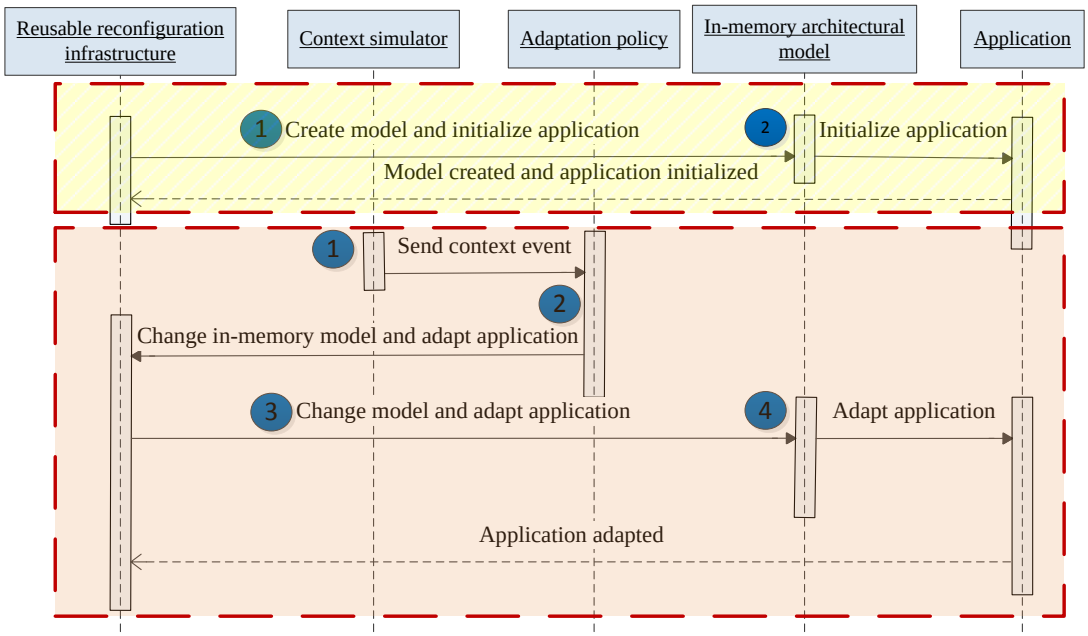

Figure 3.1 Working mechanism of the PCAA infrastructure

3.3 MAIN ELEMENTS OF PCAA INFRASTRUCTURE

The design of PCAA infrastructure is modular in which each phase of adaptation process is handled as a separate entity and external to others. As stated earlier, the infrastructure comprises three distinct elements and each element is responsible for a particular task. The high level architecture of the PCCA infrastructure is depicted in Figure 3.2.

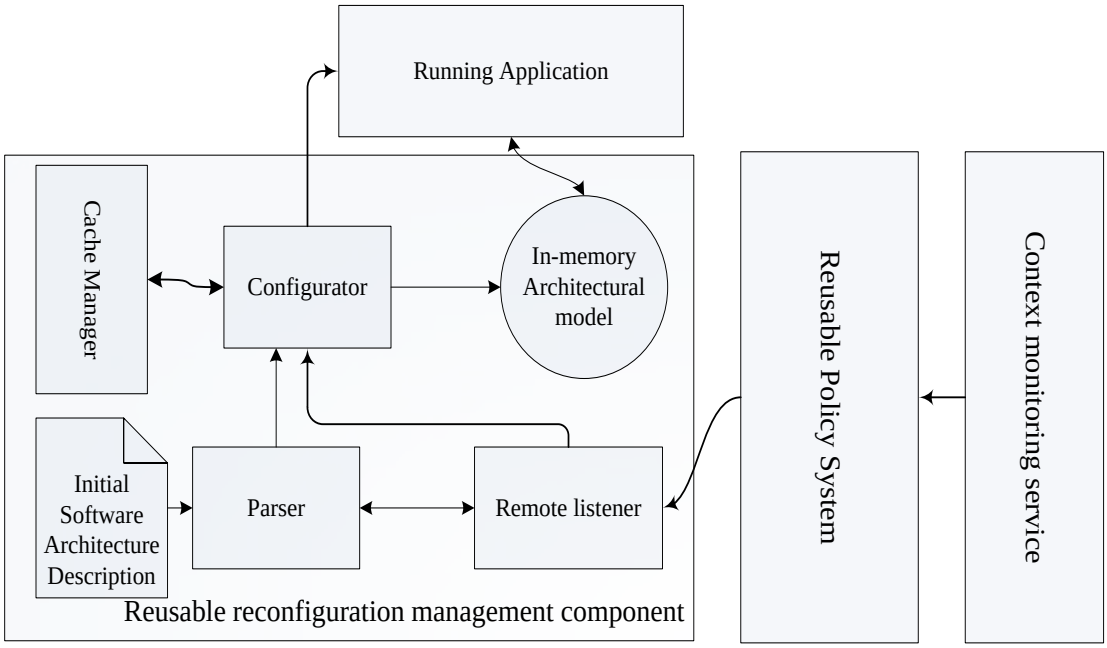

Figure 3.2 High level architecture of PCAA infrastructure

3.3.1 REUSABLE RECONFIGURATION MANAGEMENT COMPONENT

The reusable reconfiguration management component is a main element of the PCAA infrastructure, which encapsulates adaptation mechanisms. It is responsible for two things: (1) initialization of the application from its initial architecture (configuration) and (2) carrying out of application adaptation to modify the application behaviour at runtime.

Application development in PCAA infrastructure is a process of specifying initial software architecture (configuration) of the application using high-level declarative notations. Reusable reconfiguration management component initializes the application from this initial software architecture description by loading software components in the memory and interconnecting them as specified in the initial description. In addition, the reusable reconfiguration management component also maintains an in-memory model of the software architecture of the application which is causally connected to application implementation units. This in-memory model always represents the current architecture of the application and may differ from initial application

architecture during the course of application execution as the application is adapted.

There are three ways by which an application can be adapted: (1) Adding entirely new software component in the application that encapsulates different strategies / algorithms that provide the behaviour required in response to the changing contexts, (2) Removing existing components whose functionality is not required any more or (3) Replacing existing software components with new ones which implement the desired application behaviour. The reusable reconfiguration management component encapsulates this adaptation mechanism and carries out application adaptation by altering the in-memory model of the software architecture of the application through these architectural actions i-e adding new components, removing or replacing existing components. Any changes in the in-memory model are also reflected in the running application thereby achieving new application behaviour. The reusable reconfiguration management component receives adaptation actions in the form of architectural actions from *policy system* that result when a particular policy is triggered in response to the context changes (context events).

3.3.2 POLICY SYSTEM

The policy system provides the support for specification and dynamic management of adaptation policies. Adaptation policies are high level Event-Condition-Action (ECA) rules for adaptation decisions. An adaptation policy is an independent unit of execution and comprises three parts: (1) subscription to event or set of events, (2) a condition or set of conditions and (3) an action or set of actions. An event is contextual information that is sent from *context simulator widgets*, a condition is a check point to make sure that an action can be taken and the action part involves architectural actions to modify the in-memory software architectural model of the application to achieve new application behaviour.

As adaptation policies are independent units of execution, these may be added to, removed from or replaced at any point during the course of the application execution. This allows to dynamically changing the adaptation logic

at any point during application execution in order to specify new adaptation concerns without taking the application offline.

3.3.3 CONTEXT SIMULATOR WIDGETS

As the primary focus of our research is providing a flexible support for adaptation concerns and software architecture based adaptation, we do not address the issues related to obtaining context from real sensors. We, therefore, have implemented context simulator widgets that send contextual information to the policy system.

3.4 WORKING OF PCAA INFRASTRUCTURE

PCAA infrastructure has three sub-components (elements), each of which runs independently. Reusable reconfiguration management component provides the support for specification of an initial architecture of an application and its execution, and runtime support for architectural adaptation. It takes the initial architecture specification (configuration) as an input, reads and parses it, loads the components, interconnects them and initializes the application implementation. It also builds an in-memory model of the software architecture of the application which is connected to the application implementation units.

The policies are specified and stored in files that the policy system reads, parses and loads them as separate units of execution. When context simulator widget generates and sends a context event, it is sent to policy system which triggers all policies that have subscribed to the context event. When a policy is triggered, the specified conditions are checked and if conditions are true, its action part is executed. An action part of the policy contains adaptation messages (expressed as architectural commands) which are sent to reconfiguration management component. The reconfiguration management component applies the architectural commands to the in-memory software architectural model of the application to achieve new architecture of the application. The corresponding changes in the architectural model are also reflected in the running application. This results in application being adapted to meet the requirements of current contexts.

3.5 CAPABILITIES AND LIMITATIONS OF PCAA INFRASTRUCTURE

3.5.1 CAPABILITIES

- One of the strengths of our PCAA infrastructure comes from the use of software architecture as a basis of application adaptation. The benefits of software architecture are twofold: (1) it provides a loosely coupled structure, allowing the adaptation to be achieved by just rearranging its structural parts and (2) it operates at high level of abstraction. Other strength of our approach is reusability of architecture based adaptation support (adaptation mechanism) as part of the infrastructure, which means our architectural adaptation support can be used across various applications.

- Another main strength of PCAA infrastructure is flexible support for adaptation policies in that polices are separately specified (using high-level declarative notations) from rest of application code, run independently of application and are dynamically modifiable.

- Separation of concerns (i.e. application being adapted, adaptation mechanisms and adaptation policies being treated separately of each other) reduces complexity involved in application development and eases the development process.

3.5.2 LIMITATIONS

- One limitation of PCAA is that currently all components to be used in application must be written in JAVA language following PCAA component model. Components written in any other language cannot be used without having any wrapper components for them.

- In the current implementation, services a component, written following PCAA component model, can provide are limited. A component can only

provide one required service to other components. However a component can require (or use) services of other components as many as needed.

- Small configuration language developed is in its infancy. It supports a limited number of architectural commands just to test the proposed approach. The list of commands needs be expanded to include more commands. The syntax and reference naming for components needs be improved so that there is no naming conflict in policies.

3.6 SUMMARY

In this chapter, we have presented the details of the proposed policy based approach to developing and executing the adaptive context-aware applications at software architectural level. The goals of our research are to reduce complexity involved in development process and providing support for dynamic programmability of applications. We have provided design goals of PCAA infrastructure, which are separation of concerns, software architecture based adaptation and dynamic modification of adaptation policies. We have also provided an overview of PCAA infrastructure, description of its main elements and working of it. Finally, we have discussed its capabilities and limitations.

CHAPTER 4

PROTOTYPE SYSTEM IMPLEMENTATION

In previous chapter, we presented the design goals of PCAA infrastructure, description of its main design elements and explained its working along with discussion on the capabilities and limitations of PCAA infrastructure.

The design of PCAA infrastructure is modular and has separate and independent sub components. In this chapter, we discuss implementation details of each of the elements with description of its functionality.

4.1 REUSABLE RECONFIGURATION MANAGEMENT COMPONENT

This component encapsulates adaptation mechanisms that provide the support for application adaptation at architectural level. The reusable reconfiguration management component has been implemented in JAVA programming language. It has a simple GUI user interface and comprises several sub-components: Parser, Configurator, Cache Manager and Remote Listener.

4.1.1 USER INTERFACE

A simple GUI based user interface is designed, which has four sections stacked vertically (Figure 4.1). The top section has a check box labeled as “Enable Cache Support”. When it is checked before the application has started, the cache support is enabled to increase system performance. Next to checkbox is a text box, which displays the path of the file containing description of initial software architecture of the application. At last, the button labeled as “Select File and Run” is used to browse the file containing description of initial software architecture of the application stored on local disk. When a file is selected, the application is initialized from this initial description. Next section has a text pane with white background, which is used to capture and display the text from

standard output console. Third section has text pane with black background that displays the messages for actions that the reconfiguration management component performs. The bottom section comprises a text box and a button labeled as “Modify architecture”. Writing architectural commands in text box and then clicking the button (Modify architecture) will reconfigure the in-memory architectural model, thereby adapting application. Through this mechanism, we can check our architecture based adaptation support without using policies.

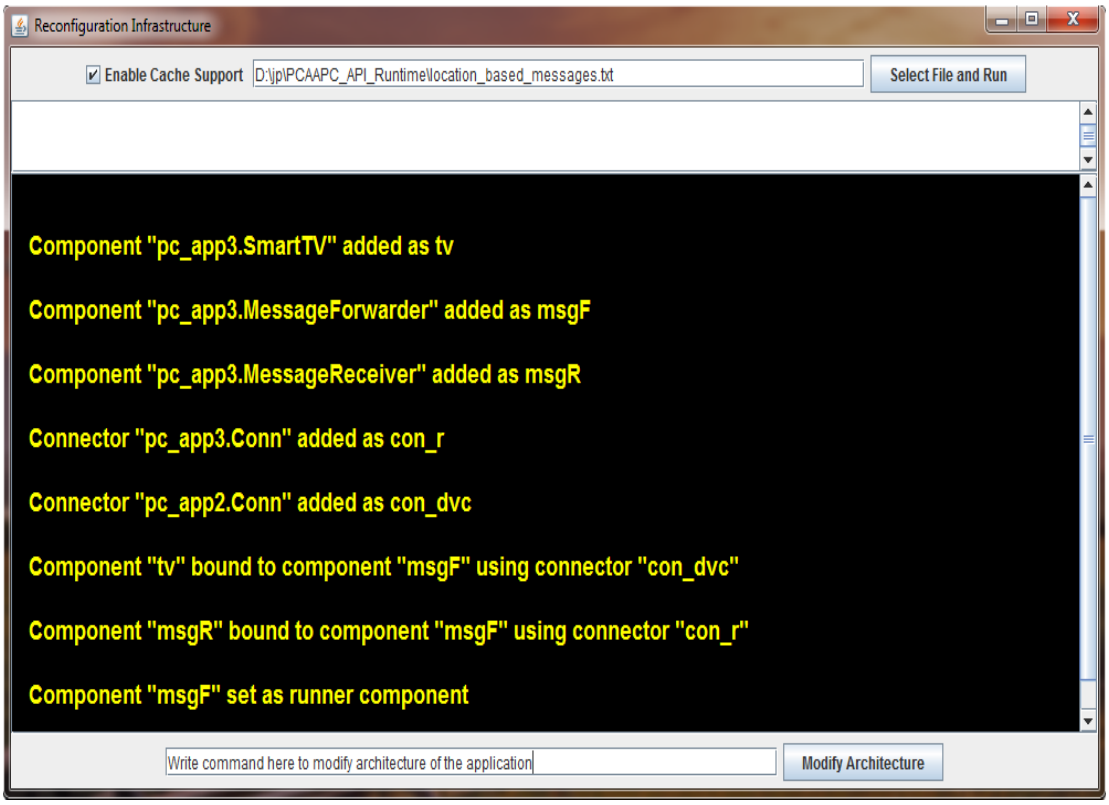

Figure 4.1 The user interface of reconfiguration management component

4.1.2 PARSER

This component of reconfiguration management is responsible to check the syntax of configuration (architectural) commands (Section 4.1.7) and produces a list of commands that are executed by the Configurator component. At the time of application initialization, when the file with specification of initial architecture of the application written in our own configuration language is

selected, the Parser component reads the file from the disk, loads its contents and goes through line by line of the specification file to check its syntax. If there is a syntax error in an architectural command, the Parser stops and reports the error. Once the file is completely checked against syntax errors, the Parser generates a list of configuration commands. The Configurator component executes each command in the list and builds an in-memory model of the software architecture of the application.

The Parser also parses the commands received through Remote Listener component. These commands are usually addition of new component in the architecture or replacement of an existing component with another new component with a similar interface.

4.1.3 CONFIGURATOR

The Configurator component is one of the core components of reconfiguration management that loads components in memory, initializes them and binds them together in order to initialize the application. It also builds an in-memory model of the software architecture of the application that is causally connected to the application implementation units.

The Configurator has two main roles. First, at application startup when Parser component gives it a list of configuration commands, it executes those commands. The commands are, in fact, software architectural commands listed and described in (Section 4.1.7). For add commands, the Configurator loads the classes in memory and instantiates the objects. For bind command, the Configurator binds the components together through connector by setting appropriate references. For start command, the Configurator calls the "*run( )*" method of component and the component gains the execution control. While initializing the application, the Configurator also creates the In-memory architectural model of the application.

The second crucial role Configurator plays is carrying out application adaptation. Application adaptation may involve loading new component in memory and replacing the old one with new ones by rebinding. Application

adaptation is triggered when Remote Listener component receives adaptation message (comprising architectural commands) which are meant to modify the in-memory architectural model and thereby application implementation.

4.1.4 IN-MEMORY SOFTWARE ARCHITECTURAL MODEL

This in-memory model of the software architecture of the application, initialized from the description of initial software architecture of the application, is stored in memory. It always represents current architecture of the application and contains references to application executing units. It evolves over time whenever application is adapted in response to changing contexts. In our approach, application adaptation is realized through this in-memory architectural model which is causally connected to application implementation units. Application adaptation is defined in terms of architectural manipulations (addition, removal or replacement of components). Application adaptation is always an architectural change for which in-memory architectural model is modified to reflect new architecture of the application. Any changes in architectural model are also reified in application implementation. This model is created and maintained by Configurator component.

4.1.5 REMOTE LISTENER

The Remote Listener is actually a java RMI service which is exported by reconfiguration management component so that policy system can interact with it. It is a means to receive adaptation message in the form of architectural commands from action part of the policy. When it receives the architectural commands, it sends them to Parser. If Parser successfully parses the commands then Remote Listener invokes Configurator component which executes the commands to carry out application adaptation and the application is reconfigured to meet the requirements of the current context.

4.1.6 CACHE MANAGER

As a result of application adaptation, new components may be added in the in-memory architectural model and removed from it. When a component is

removed, its reference is also removed from the model and resultantly, the component memory is reclaimed. When the same component needs to be added into the application later on, it is reloaded in memory and used in the application. Component reloading incurs cost in terms of time required to load the component. When cache support is enabled in reconfiguration management component, the component memory is not reclaimed and the component is cached by Cache Manager.

Cache Manager is used when caching support is enabled in reconfiguration management. It is invoked when components are removed from the in-memory model. It maintains references to the components removed from in-memory model, so the components are not garbage collected. When a component removed from the model is required in the application later on, its reference is obtained from Cache Manager and the component is used in the application without requiring reload. This is how cache improves performance of the system.

4.1.7 CONFIGURATION LANGUAGE

The approach to application adaptation in PCAA infrastructure is software architecture based, in which application composition and application adaptation (reconfiguration) is achieved through the use of software architecture. Initial application composition (specification) is expressed in a small configuration language that we have developed as part of the infrastructure. Initial software architecture of the application specifies the components and connectors used and their interconnection. At application runtime when application is adapted, the software architecture of the application is reconfigured to change the behaviour of the application. The specification of adaptation message is also expressed in the small configuration language.

The configuration language is a declarative language and has some basic constructs as architectural commands. The list of commands is short just to meet the requirements of the PCAA infrastructure. The basic commands are meant to build and modify the software architecture of the application. Here is the description of some of the commands:

4.1.7.1 ADD COMMAND

The add command is for addition of component or connector in the architecture. This command is used to specify the initial architecture of the application and also used at application runtime while application needs to be adapted. At runtime application adaptation, add command dynamically adds the component in architecture. The syntax of the command to add a component in the architecture is shown in (Figure 4.2).

add component *className* as *identifier*

Figure 4.2 The syntax of add command for adding a component

In the above command *"add"*, *"component"* and *"as"* are keywords whereas *"className"* is the fully qualified name of the JAVA class encapsulating the implementation of component and *"identifier"* is the reference name given to the component.

Similarly to add a connector in the architecture, the add command is used. The syntax for the command is shown in (Figure 4.3) below.

add connector *className* as *identifier*

Figure 4.3 The syntax of add command for adding a connector

In the above command *"add"*, *"connector"* and *"as"* are keywords whereas *"className"* is the fully qualified name of the JAVA class encapsulating the implementation of connector and *"identifier"* is the reference name given to the connector.

4.1.7.2 BIND COMMAND

The bind command is used to specify the interconnection of components. It binds two components together through a connector. One component provides the service that another component requires and connector is the mediator through

which component requiring the service uses the service that another component provides. The syntax of the command is given in (Figure 4.4).

bind *identifier1* at *port1* to *identifier2* at *port2* using *identifier3*

Figure 4.4 The syntax of bind command

In the above command *"bind"*, *"at"*, *"to"* and *"using"* are keywords whereas "*identifier*" is the reference name of the component or connector and *"portN"* is name of the port where the service is required / provided. The *"identifier1"* refers to the component that provides the service at *"port1"*, *"identifier2"* refers to the component that requires the service at *"port2"* and *"identifier3"* is the reference name of the connector mediating the communication between component using the service and component providing the service.

4.1.7.3 REPLACE COMMAND

The replace command is used for application adaptation where old components or connectors need to be replaced with new components or connectors respectively, to adapt the application to change its behaviour in response to change in the context. The syntax of the command is given in (Figure 4.5).

replace component *identifier1* with *identifier2*

Figure 4.5 The syntax of replace command

In the above command *"replace"*, *"component"* and *"with"* are keywords whereas *"identifier1"* is reference name for the component that needs to be replaced and *"identifier2"* refers to the new component replacing the old one.

4.1.7.4 START COMMAND

The start command starts execution of the application. Application startup takes from the component that implements IRunner interface. The interface tags the component as the entry point of the application. The syntax of the command is shown in (Figure 4.6).

start *identifier*

Figure 4.6 The syntax of start command

In the above command *"start"*, is the keyword whereas *"identifier"* is a reference name of the startup component.

4.2 POLICY SYSTEM

One of the core requirements of our proposed approach is a provision of flexible support for an important facet of overall adaptation process, i.e. adaptation policies. To realize this requirement, we have adopted a third party policy system called Ponder2 (Twidle, Lupu et al. 2008, Twidle, Dulay et al. 2009). We choose Ponder2, as it is a light-weight, self-contained and extensible policy system that can be used across all devices from small resource-constrained devices to complex environments. In addition, Ponder2 uses a declarative language called PonderTalk to specify polices, which provides better transparency to the developers, as policies are specified at higher-level of abstractions (Dhomeja 2011).

4.2.1 PONDER2

Ponder2 is a light-weight policy system for specification and enforcement of policies. Ponder2 supports both authorization and obligation policies (ECA rules). We use Ponder2 ECA rules for adaptation policies, which are expressed in PonderTalk language (a declarative language) provided by Ponder2 system. In Ponder2 everything is implemented as managed object and controlled through PonderTalk message keywords. A managed object is implemented in JAVA and its methods are annotated like @Ponder2op ("reconfig:"). The annotations are used for establishing link between JAVA method and PonderTalk message keyword. Ponder2 software and its documentation are available at: http://www.ponder2.net.

4.3 CONTEXT SIMULATOR WIDGETS

In current implementation of the PCAA infrastructure, there are no practical mechanisms to interact with the sensors deployed in the environment to acquire the contextual information. However, to serve the purpose of testing hypothetical example scenarios to be run in PCAA infrastructure, we have designed several simulators as the context widgets to provide the contextual information for the applications. The use of simulated context monitors, contrary to practical integration with real sensors, has no effect on our research objectives, as the primary focus of our research is providing a flexible support for adaptation concerns and software architecture based adaptation.

The simulators are GUI components implemented as Ponder2 managed objects and generate Ponder2 events to trigger Ponder2 actions. The Location Context Widget and User Context Widget are shown below (Figures 4.7 and 4.8).

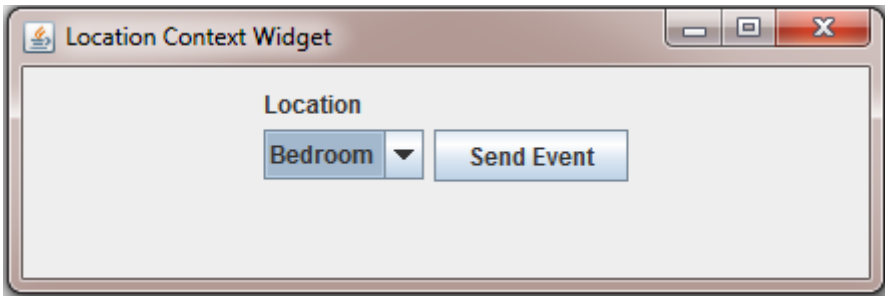

Figure 4.7 Location Context Widget

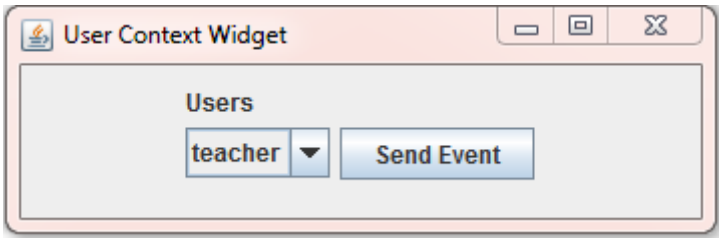

Figure 4.8 User Context Widget

4.4 COMPONENT MODEL IN PCAA INFRASTRUCTURE

Application in PCAA infrastructure is a composition of software components and connectors. Software components encapsulate core business logic of the application. Each component is in charge of a particular task in the application and may provide service to other components and require services of other components. Software connector mediates communication between components. The components do not interact with each other directly, rather they communicate through software connectors. A component exports its service through provided interface and uses services of other components at required interface. A component can require as many services as needed but can provide only one service to other components. A connector binds two components together. It has two interfaces, at one interface a component providing the service is attached and at other interface, the component requiring the service is attached. The connector can bind together only two components, one component providing the service and other requiring the service. The communication between components is achieved through method invocation via connector. Both the component and connector are first class entities. We have provided a JAVA API which includes basic interfaces and classes for implementing components and connectors following PCAA infrastructure component model.

4.4.1 PCAA INFRASTRUCTURE COMPONENT MODEL API

We have provided an API for writing components and connectors following PCAA infrastructure component model. The API comprises some basic interfaces and classes which are written in JAVA.

4.4.1.1 COMPONENT INTERFACE AND CLASS

The *"IComponent"* interface (Appendix A, Section A.1) in package pcaapc.api is the basic interface for component specification. The method *"void initialize( )"* is aimed for component initialization, once it is loaded. In this method, component initialization tasks, such as establishing connections with database server etc. are accomplished. A component can provide service to other components that they require by implementing *"Output doRequired(String port,*

Input in)" method, where *port* is connection point and *in* is an object of type Input used to pass data to the method. The method returns data through object of type Output. Both Input and Output are interfaces for tagging purpose only. Similarly, a component can use services of other components by implementing the *"void doProvided(String port)"*. An object is bound to another component through connector component. Connectors are attached to the component through method *"void setConnector(IConnector connector, String port)"*, where *connector* refers to Connector instance and *port* specifies connection point. A list of all the attached connectors to the component can be accessed through method *"public Hashtable<String, IConnector> getConnectors( )"*. The API, however, provides a *"Component"* class (Appendix A, Section A.2) in package pcaapc.api that provides implementation of all the methods in *"IComponent"* interface. The programmers can extend this class to define their own components.

4.4.1.2 CONNECTOR INTERFACE AND CLASS

The *"IConnector"* interface (Appendix A, Section A.3) in package pcaapc.api is the basic interface for connector specification. A connector binds together two components, one providing the service and the other requiring the service. The component providing the service is attached through method *"void setProvided(IComponent provided)*", where *provided* refers to the Component instance. Similarly, the component requiring the service is attached through method *"void setRequireded(IComponent required)*", where *required* refers to the Component instance. The attached components, requiring and providing the service can be accessed through *"IComponent getRequired( )"* and *"IComponent getProvided( )"* methods respectively. The interface also declares two methods *"public Output doRequired(String port, Input in)* and *"void doProvided(String port)"* for delegating method invokes from one component to other component. The API, however, provides a *"Connector"* class (Appendix A, Section A.4) in package pcaapc.api that provides implementation of all the methods in *"IConnector"* interface. The programmers should extend this class to define their own connectors.

4.4.1.3 INPUT AND OUTPUT INTERFACES

These interfaces are used for just tagging purpose. If an object is to be passed to method *"Output doRequired(String port, Input in)"*, it must implement *"Input"* interface (Appendix A, Section A.5) to tag it to type *"Input"*. The method returns objects which implement *"Output"* interface (see Appendix A, Section A.6).

4.4.1.4 IRUNNER INTERFACE

The *"IRunner"* interface extends *"Runnable"* interface (Appendix A, Section A.7). The component that must get control when an application is initialized, it must implement the *"IRunner"* interface. The component provides implementation of *"run ( )"* method and controlling code is placed inside it.

4.5 SUMMARY

In this chapter we have provided implementation details of one of the core components of PCAA infrastructure, reusable reconfiguration management component, which performs two main tasks: application initialization and its adaptation. The application is initialized from the description of its initial architecture and an In-memory architectural model of the application is created. The application is adapted by reconfiguring the In-memory architectural model and the changes made in the model are enacted in the application. The adaptation concerns are expressed as ECA adaptation polices specified separately from application configuration code and added to the application once it is running. The policies are dynamically modifiable. The chapter also provides details on small configuration language, its basic constructs and syntax. Finally, the description of PCAA infrastructure component model is presented, along with discussion on API provided as part of the infrastructure.

CHAPTER 5

EXAMPLE APPLICATIONS AND EVALUATION

In this chapter we present the description, design, development and execution of some hypothetical adaptive context-aware applications using PCAA infrastructure. Finally, we evaluate the performance of the system and dynamic modifiability of adaptation policies.

5.1 LOCATION BASED MESSAGE DELIVERY

Location based message delivery (LBMD) is the application in which the messages for a user from remote source are presented to the user through the nearest device available to the user. For example the user may prefer to receive the messages on smart TV, if she is watching TV in the TV hall and may be interested to receive the messages on the smart phone when in the bedroom.

The application is composed of three software components (Figure 5.1). The MessageReceiver component receives messages from remote sources; it has only one provided interface where it provides the messages received from remote source. The code for the component is presented in (Appendix B, Section B.1.2). The MessageForwarder component (Appendix B, Section B.1.1) reads messages from MessageReceiver component and sends to the bound device for display. It has two required interfaces. On one required interface, it reads messages through provided interface of MessageReceiver component. On other required interface, it forwards the messages through provided interface of SmartTV or SmartPhone. The third component is the device component. In the demonstration application, we implement only two components SmartTV (Appendix B, Section B.1.4) and SmartPhone (Appendix B, Section B.1.3). The SmartTV or SmartPhone

components imitate display devices and show the messages. They have only one provided interface for displaying the data.

Figure 5.1 High level diagram of LBMD application

The initial software architecture of the application expressed in the small configuration language is given below (Figure 5.2).

```
1. add component pcaa_app3.SmartTV as tv

2. add component pcaa_app3.MessageForwarder as msgF

3. add component pcaa_app3.MessageReceiver as msgR

4. add connector pcaa_app3.Connector as con_r

5. add connector pcaa_app2.Connector as con_dvc

6. bind tv at mf to msgF at mf using con_dvc

7. bind msgR at mr to msgF at mr using con_r

8. start msgF
```

Figure 5.2 Initial software architecture of the LBMD application

To demonstrate the runtime change in the behaviour of Location based message delivery application, we consider two possibilities. If the user is in TV hall watching television then the messages are delivered to and displayed at Smart TV. If the user is in bedroom then the messages are forwarded and displayed at Smart Phone.

We write two policies, one for the TV hall and the other for Bedroom. Both policies subscribe to user location event. In a policy for a TV hall, we specify that if the location of the user is TV hall then Display Device component is replaced with SmartTV component. TV hall policy (Figure 5.3) subscribes to user location context event (line 2), when user location context event occurs and the location is "TV Hall" (line 3), action part of the policy is performed. The action part includes sending reconfiguration message (lines 4, 5 and 6) to reusable reconfiguration management component, which eventually adapts the application. In a policy for bedroom, if the user is in bedroom then display device component is replaced with SmartPhone component (Figure 5.4). First the SmartPhone component is added to the system and then SmartTV component is replaced with SmartPhone component. The reconfiguration process is shown in (Figure 5.5). When the location event occurs and if the user location is TV hall, all the messages are forwarded and displayed on smart TV. If the location is bedroom then messages are forwarded and displayed on smart phone.

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/locationevent;
3. condition: [ :type :value | value == "TV Hall" ];
4. action: [ :type :value |
5. config reconfig: "add component pcaa_app3.SmartTV as tv;
6  replace component sp with tv".
7. active: true.
```

Figure 5.3 TV policy specification

1. policy := root/factory/ecapolicy create.
2. policy event: root/event/locationevent;
3. condition: [:type :value | value == “Bedroom”];
4. action: [:type :value |
5. config reconfig: “add component pcaa_app3.SmartPhone as sp;
6. replace component tv with sp”.
7. active: true.

Figure 5.4 Bedroom policy specification

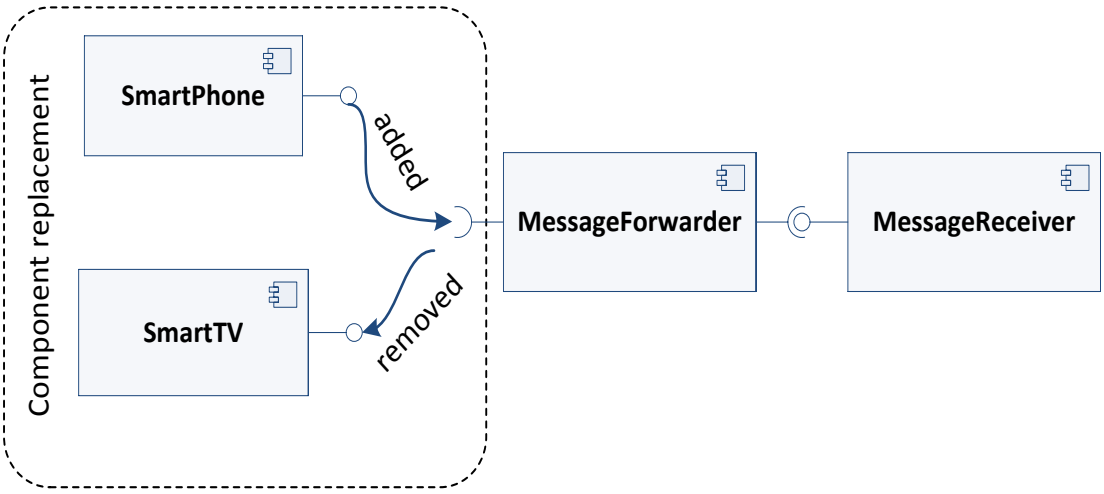

Figure 5.5 Adaptation in LBMD application as replacement of components

5.2 SMART NOTICE BOARD

In this example scenario, we present an automated smart version of a traditional notice board. In an academic institution, a traditional notice board is a means where up-to-date academic information relating to students or teachers is available. Teachers or students locate and read the information relevant to them.

On the contrary, a Smart Notice Board (SNB) is a hypothetical application in which the notice board is smart enough that it provides the information relevant to the person who is in front of the notice board. If a teacher is in front of the board, information relating to the teacher is displayed or if a student is there, student related information is presented. It is also capable of presenting information in different views. Some people may prefer the information to be displayed in tabular form while the others might be interested in seeing the charts, or graphical view.

The application is composed of three software components. The Data component (either StudentData or TeacherData) provides information related to teachers or students. It may be noted that there can be as many Data components as needed such as HoDData (for Head of Department data), but we currently implement and discuss only two components viz TeacherData (Appendix B, Section B.2.3) and StudentData (Appendix B, Section B.2.2) to simplify the application. Data component has only one provided interface where it exports the data. The View component renders the data into particular form, it has one required interface where it requires the data through provided interface of the Data component and one provided interface where it exports formatted data (rendered in a particular form such as charts etc.) to be displayed by SmartNoticeBoard component. The code for the component is presented in (Appendix B, Section B.2.1). The SmartNoticeBoard is the component that displays the data rendered in particular view. It has one required interface where it requires the data rendered in particular view through provided interface of View component and one provided interface where it provides the data for display (Figure 5.6). The View component formats the data into particular view. The implementation code for two View components is presented in (Appendix B, Sections B.2.4 and B.2.5).

Figure 5.6 High level diagram of smart notice board application

The initial software architecture of the application coded with small configuration language is given below (Figure 5.7).

```
1. add component pcaa_app2.SmartNoticeBoard as nb
2. add component pcaa_app2.View1 as v1
3. add component pcaa_app2.StudentData as st
4. add connector pcaa_app2.Connector as con_d
5. add connector pcaa_app2.Connector as con_v
6. bind st at dp to v1 at dp using con_d
7. bind v1 at vp to nb at vp using con_v
8. start nb
```

Figure 5.7 Initial software architecture of the SNB application

To demonstrate the runtime change in the behaviour of Smart Notice Board application, we consider two possibilities. If the person standing in front of the notice board is a teacher then the smart notice board displays data relevant to the teacher. In second case, if the person standing in front of the notice board is a student then the notice board displays the data relevant to the student.

There are two policies involved in the scenario, one for the teacher and the other for student. Both policies subscribe to a user presence event. In a policy for a teacher, we specify that if the user is teacher, then Data component is replaced with TeacherData component (Figure 5.8). In a policy for student, if the user standing before smart notice board is a student, then Data component is replaced with StudentData component (Figure 5.9). When the user presence event occurs and if the user is a teacher, the TeacherData component is added in the system and StudentData component is replaced with TeacherData component (Figure 5.10). This, in effect, changes the behaviour of the notice board, as now the data relevant to the teacher is displayed. On the other hand, if the user is student then Data component is replaced with StudentData component. This results in the notice board presenting information concerning the student.

1. policy := root/factory/ecapolicy create.
2. policy event: root/event/userpresenceevent;
3. condition: [:type :value | value == “teacher”];
4. action: [:type :value |
5. config reconfig: “add component pcaa_app2.TeacherData as teach;
6. replace component st with teach”.
7. active: true.

Figure 5.8 Teacher policy specification

```
1. policy := root/factory/ecapolicy create.

2. policy event: root/event/userevent;

3. condition: [ :type :value | value == “student” ];

4. action: [ :type :value |

5. config reconfig: “add component pcaa_app2.StudentData as st;

6. replace component teach with st”.

7. active: true.
```

Figure 5.9 Student policy specification

In the above code (Figure 5.8 and Figure 5.9), an ECA policy is created which subscribes to user presence event. If the user presence event occurs, the policy is triggered. If the user is a teacher, Data component is replaced with TeacherData component and if the user is a student then Data component is replaced with StudentData. This is how the Smart Notice Board application is adapted in response to change in its current context (user presence).

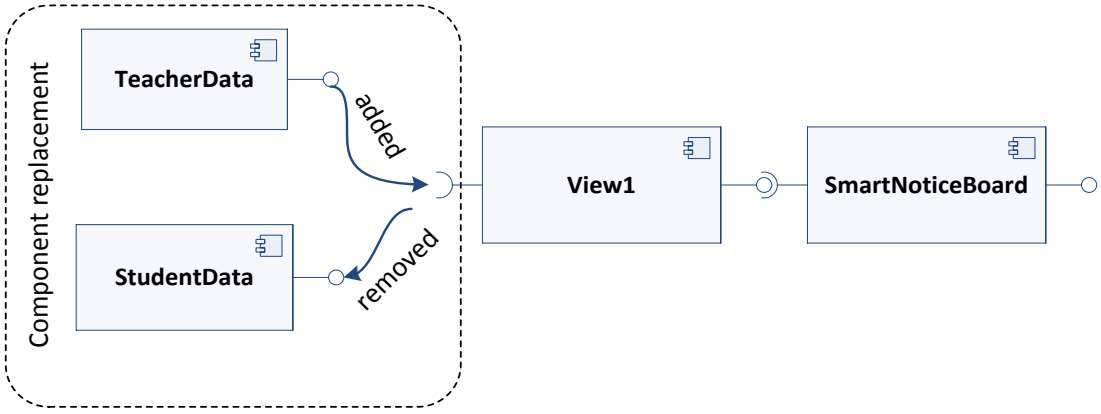

Figure 5.10 Adaptation in SNB application as replacement of components

5.3 CONTEXT-AWARE COMPRESSION SERVER

In this example scenario, there is a server application that sends data to remote clients using some compression technique. The main software components of the server application include DataStore component (Appendix B, Section B.3.1), Compressor component (Appendix B, Sections B.3.3 and B.3.4) and the Provider component (Appendix B, Section B.3.2). The high-level diagram of the application is shown in (Figure 5.11). The DataStore component represents the source of data. It has only one provided interface where it provides the data. The Compressor component provides the compression service, it has only one provided interface where it exports compression service. The Provider component is the main component whose task is reading data from DataStore and compressing the data through Compressor and then sending the compressed data to remote client. The Provider component has two required interfaces and one provided interface. One required interface requires the service provided by the DataStore component and the other required interface requires the service provided by the Compressor component, whereas one provided interface provides the compressed data to remote clients.

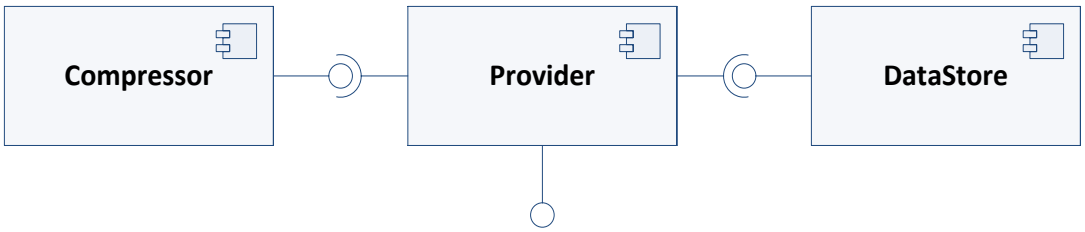

Figure 5.11 High level diagram of compression server application

The initial software architecture of the application, expressed in small configuration language, is given below (Figure 5.12).

1. add component pcaa_app.Compressor as compR
2. add component pcaa_app.DataStore as dataR
3. add component pcaa_app.Provider as pro
4. add connector pcaa_app.Connector as con_dc
5. add connector pcaa_app.Connector as con_ds
6. bind compR at dc to pro at dc using con_dc
7. bind dataR at ds to pro at ds using con_ds
8. start pro

Figure 5.12 Initial software architecture of the server application

To demonstrate the runtime change in the behaviour of server application, we consider the possibility of the change in bandwidth. If the bandwidth falls below some threshold, the server must increase compression ratio so as more data can be sent in the packets to compensate the fall of bandwidth. We write a policy where we specify that when the bandwidth falls below some threshold (for instance, less than 100), the Compressor component is replaced with another Compressor component providing more compression ratio. The policy is subscribed to bandwidth event. When the bandwidth event occurs, the policy is triggered and if the condition is true (bandwidth less than 100), the Compressor component is replaced with another Compressor component. The policy description is given below (Figure 5.13). Similarly, to decrease the computational load on server, a Compressor component with less compression ratio can be added when bandwidth exceeds the threshold (greater than 100). The policy description is given in (Figure 5.14).

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/bandwidthevent;
3. condition: [ :type :value | value < 100 ];
4. action: [ :type :value |
5. config reconfig:"add component pcaa_app.ComplexCompressor as comp2;
6. replace component compR with comp2".
7. active: true.
```

Figure 5.13 Policy specification for bandwidth less than 100

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/bandwidthevent;
3. condition: [ :type :value | value > 100 ];
4. action: [ :type :value |
5. config reconfig:"add component pcaa_app.Compressor as compR;
6. replace component comp2 with compR".
7. active: true.
```

Figure 5.14 Policy specification for bandwidth greater than 100

In the (Figure 5.13), a policy is created that subscribes to bandwidth event. If the bandwidth event occurs, the policy is triggered and if the bandwidth is less than 100, the new Compressor component is added to the application that

is capable of having more compression ratio. The new component then replaces the old Compressor component that is already in use (Figure 5.15). This is how the server application is adapted in response to change in its current context (fall in bandwidth).

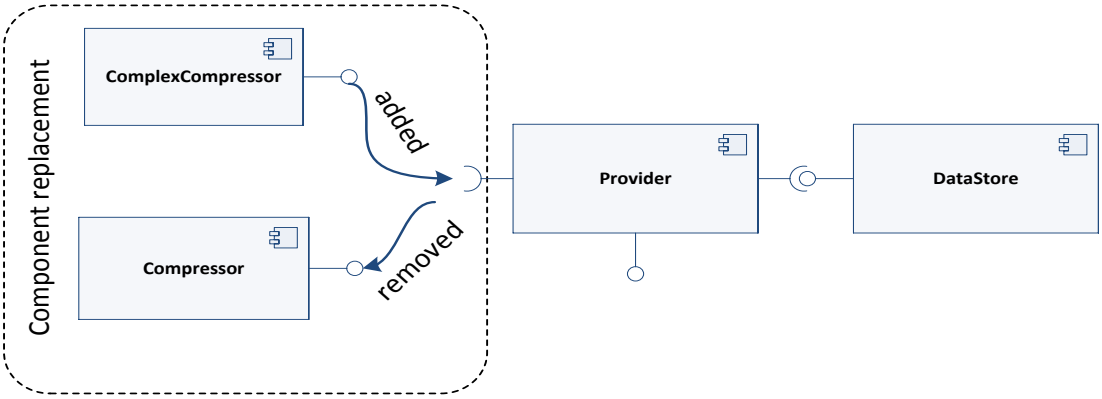

Figure 5.15 Adaptation in server application as replacement of components

5.4 EVALUATION

In this section, we evaluate our proposed system through performance analysis and support for dynamic modifiability of adaptation concerns.

5.4.1 PERFORMANCE ANALYSIS

To study the performance of our system, we conduct tests by executing one of the hypothetical applications (described in this chapter) and measuring total adaptation time taken by our system. Total adaptation time is measured from a point context is sent by the context simulator widget to the policy, which has subscribed to it, until the application is adapted in response to policy evaluation. This time (t_{tadp}) is a sum of time (t_p) taken by policy system (for policy enforcement) and time (t_{adp}) taken by reconfiguration management component to adapt the application, which can be expressed as:

$$t_{tadp} = t_p + t_{adp}$$

Figure 5.16 Equation for total adaptation time

The time (t_{tadp}) taken by reconfiguration management component for application adaptation is a sum of time (t_{load}) in loading the new component in memory and time (t_r) in reconfiguring the architecture.

$$t_{adp} = t_{load} + t_r$$

Figure 5.17 Equation for reconfiguration time

Performance tests are conducted on a Windows platform (CPU Intel Core i3-3120M 2.50 GHz, RAM 2GB, OS Windows 7 Ultimate 64-bit Service Pack 1, Java version JDK1.7.0). All the infrastructural elements of the system run on a single machine. To achieve a better measurement, the adaptation policy is triggered thirty (30) times, which responds to user location context event.

The reported times, along with standard deviation and confidence intervals, are presented in (Figures 5.18 to 5.23). The graph (Figure 5.18) shows total adaptation time (t_{tadp}) along with time (t_p) for policy evaluation and enforcement and application adaptation time (t_{adp}). The graph (Figure 5.19) shows total adaptation time along with confidence intervals with different confidence levels that the intervals contain true mean. In (Figure 5.20) application adaptation time (t_{adp}) which is sum of component load time (t_{load}) and architectural reconfiguration time (t_r) is shown. The policy time (t_p) is independent of application whereas reconfiguration time (t_r) is variable from component to component. For example replacing a component having more bindings with other components may take more time, as it would require for the new component to establish bindings with all components that the replaced component is bound to. The pie chart (Figure 5.21) shows total adaptation time (t_{tadp}) split into the percentage of times taken by each: policy enforcement (t_p), component loading (t_{load}) and architectural reconfiguration (t_r). The pie chart indicates that the largest contribution in overall adaptation time is provided by component loading time (t_{load}), while performance overhead of adaptation policies (t_p) and architectural reconfiguration (t_r) is minimal. The graphs (Figure 5.22 and Figure 5.23) show execution times when caching support is enabled and indicates that caching support improves the performance by

avoiding component reloading (t_{load}), thereby minimizing the total adaptation time.

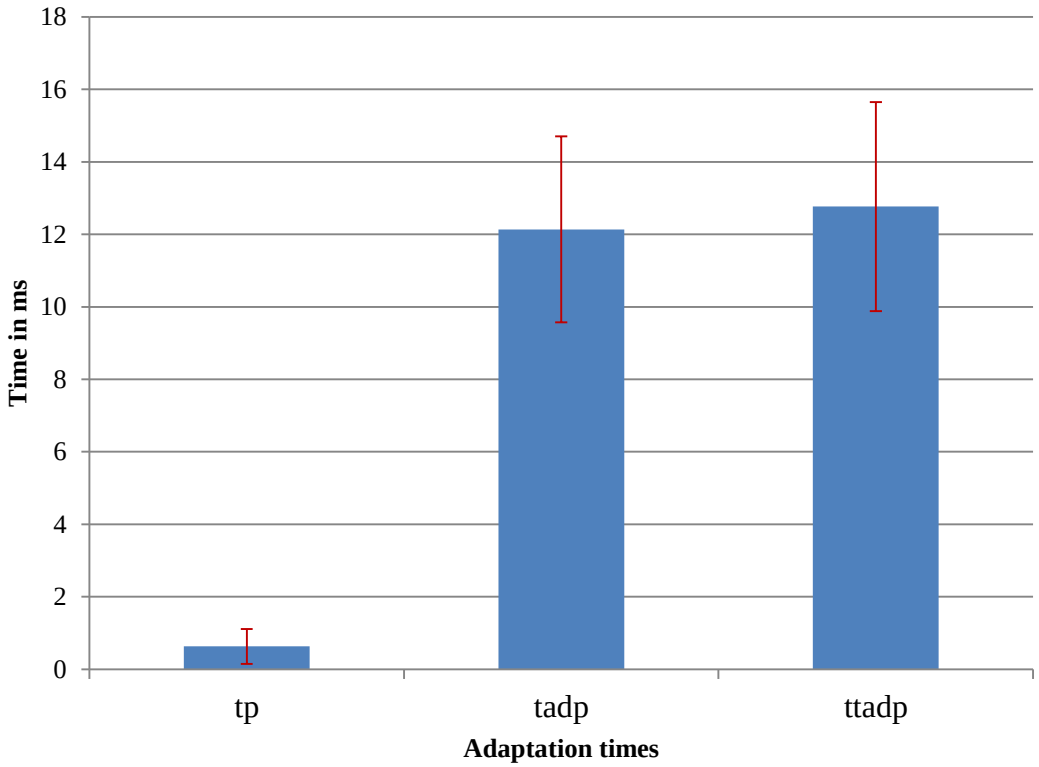

Figure 5.18 Graph showing total adaptation time ($t_{tadp} = t_p + t_{adp}$) along with standard deviation

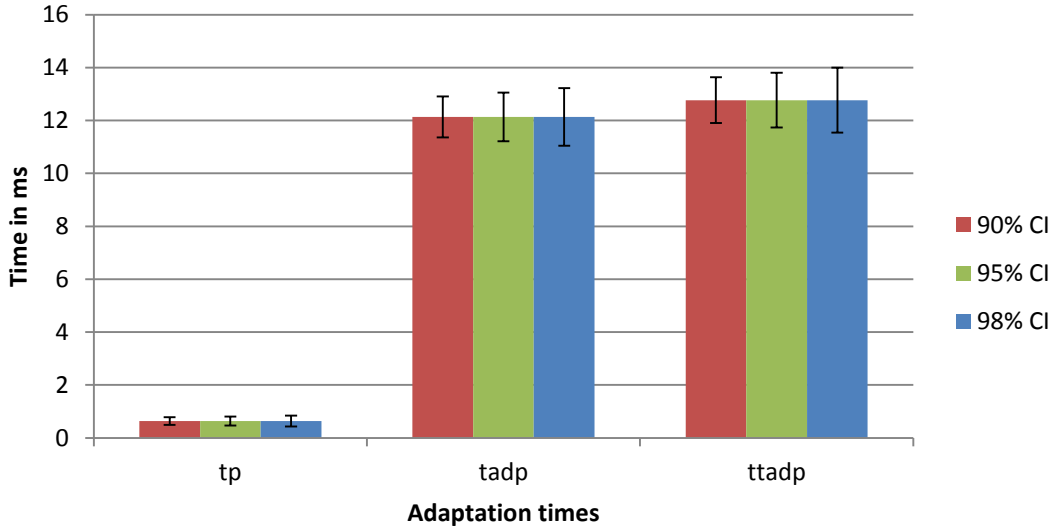

Figure 5.19 Graph showing total adaptation time ($t_{tadp} = t_p + t_{adp}$) along with confidence intervals

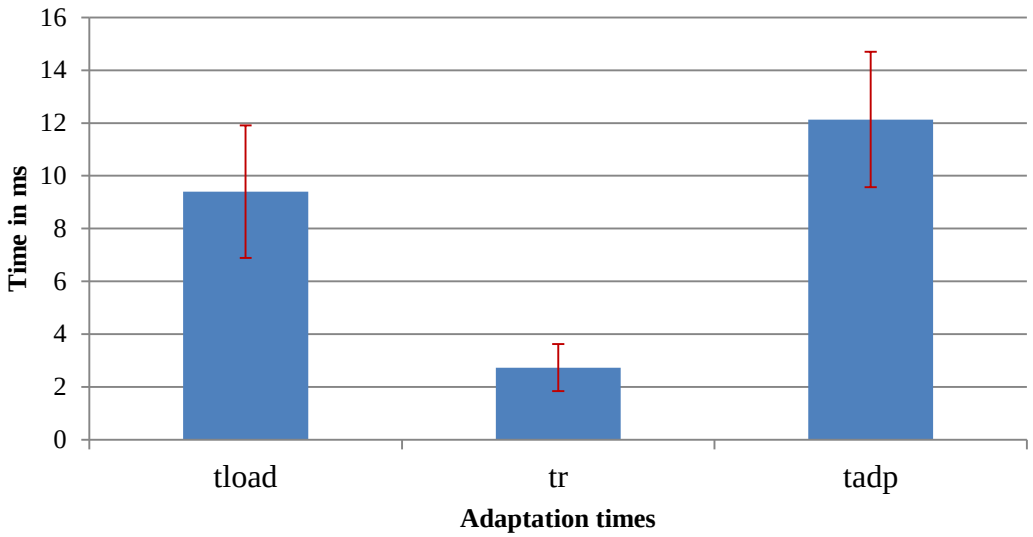

Figure 5.20 Graph showing application adaptation time ($t_{adp} = t_{load} + t_r$) along with standard deviation

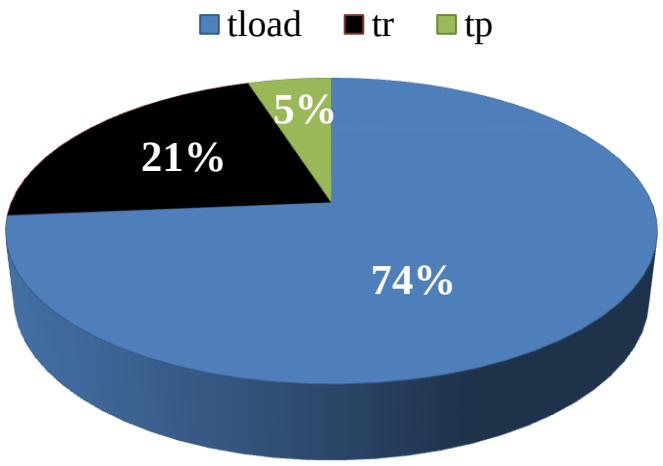

Figure 5.21 Pie chart showing percentage of different adaptation times

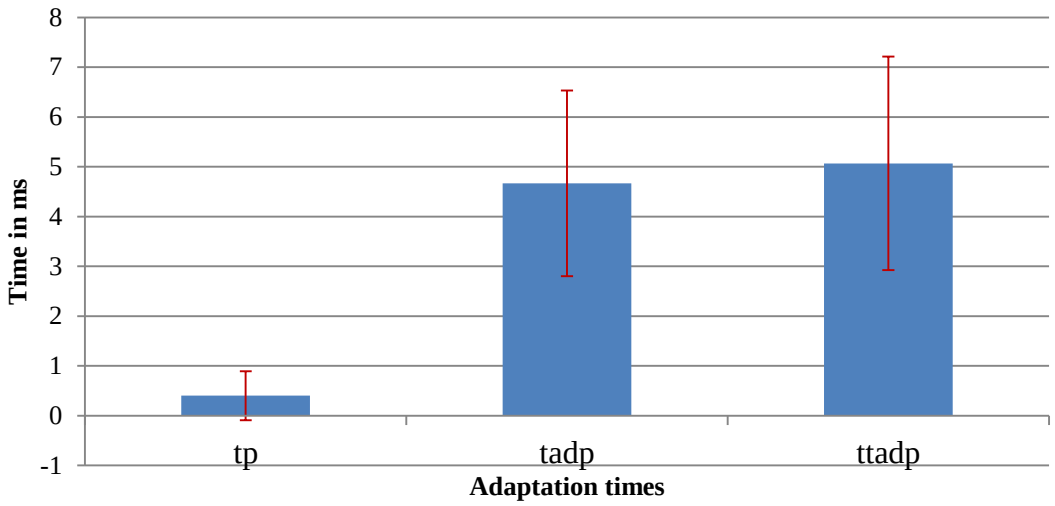

Figure 5.22 Graph showing total adaptation time ($t_{tadp} = t_p + t_{adp}$) along with standard deviation with cache enabled

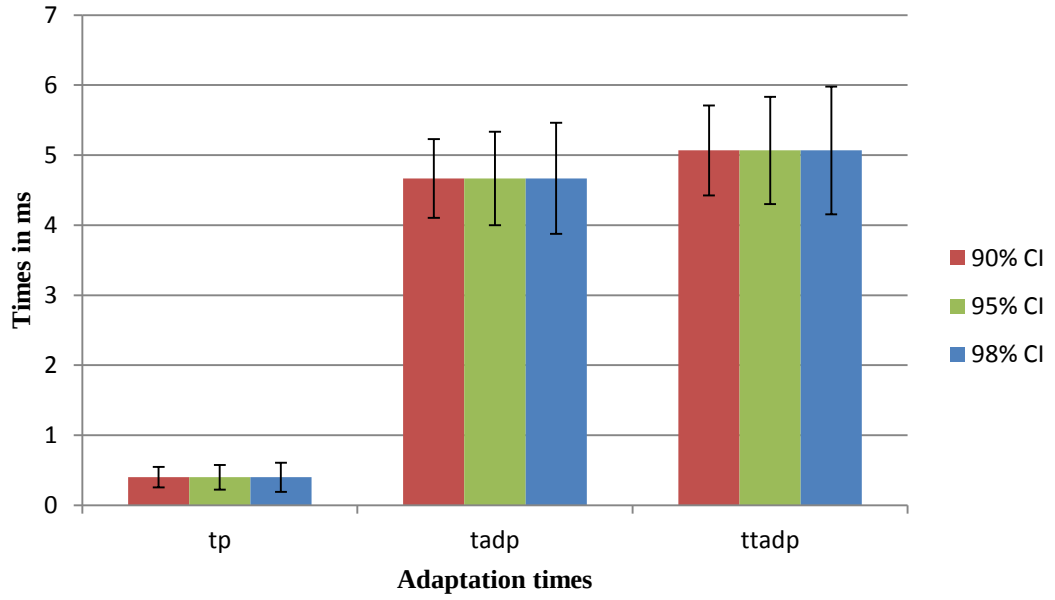

Figure 5.23 Graph showing total adaptation time ($t_{tadp} = t_p + t_{adp}$) along with confidence intervals with cache enabled

The results in the form of graphs showing means of different execution times along with standard deviation and confidence intervals are presented above. It is evident from the results that the use of polices in our approach towards software architecture based context-aware adaptation provides a greater flexibility in terms of dynamic programmability of applications with a minimal performance overhead.

5.4.2 DYNAMIC MODIFIABILITY OF ADAPTATION POLICIES

In this section we evaluate the main feature of our research that our proposed approach supports dynamic modifiability of adaptation policies. Through Location Based Message Delivery application presented in (Section 5.1), we show how policies are dynamically modified without taking the system offline. The demonstration application has two adaptation policies: TV hall policy and bedroom policy. We use bedroom policy and modify it to demonstrate dynamic modifiability of this policy.

Let us assume the user preference has changed. She is interested to receive messages into her email inbox instead of smart phone when she is in bedroom. This requires modifying lines 5 and 6 of the bedroom policy shown in (Figure 5.4) presented in (Section 5.1). The modified bedroom policy is shown in (Figure 5.24).

Next step is to load the modified policy through Ponder2 shell without shutting down and restarting the running application. Now in response to location context event (location = “Bedroom”), messages are forwarded to Email Inbox.

```
1. policy := root/factory/ecapolicy create.
2. policy event: root/event/locationevent;
3. condition: [ :type :value | value == “Bedroom” ];
4. action: [ :type :value |
5. config reconfig: “add component pcaa_app3.EmailInbox as ei;
6. replace component tv with ei”.
7. active: true.
```

Figure 5.24 Modified bedroom policy specification

Above discussion demonstrates that our proposed approach supports dynamic modification of adaptation concerns without shutting down and restarting the system.

5.5 SUMMARY

In this chapter, we have presented description of several hypothetical applications along with their design and implementation.

To substantiate our approach, we have conducted performance tests by executing example application designed and developed in this chapter. The results presented as graphs show that the largest contribution in overall adaptation time is provided by component loading time, while performance overhead of policies and architectural reconfiguration is minimal. Further caching support improves the performance by minimizing the total adaptation time. We have also demonstrated that our proposed approach supports dynamic modification of adaptation concerns without shutting down and restarting the system. We modified the adaptation policy and reloaded it without affecting the application while it was executing.

CHAPTER 6

CONCLUSION AND FUTURE WORK

This chapter concludes the thesis with discussion on the benefits of the proposed approach and also provides the summary of contributions of the thesis. Finally, it outlines direction for future work.

6.1 CONCLUSION

The primary goal of pervasive computing is to support user tasks, satisfy user needs and enrich user experience with minimal or no user distraction. Context-awareness in general and context-aware adaptation in particular is central to achieving this goal. Context-aware adaptation is a process in which applications acquire contextual information, reason upon it and adjust their behaviour accordingly. The development of adaptive context-aware applications is a challenging task and therefore various solutions have been proposed in the literature with an aim to simplify the development efforts. These solutions are provided in the form of programming languages, middleware and architecture based solutions. Towards this goal, architecture based solutions are more flexible in the sense that architecture of the application is a loosely coupled structure (allowing structural parts to be rearranged and hence providing easy means to achieve adaptation) and operate at higher level of abstraction. However, existing architecture based adaptation approaches do not provide flexible solution to another important facet of context-aware adaption process — adaptation policies. The policies, in these approaches, are tightly coupled with application code and dynamically un-modifiable. We have addressed this issue and provided a policy and software architecture based solution following separation of concerns principle in which all the concerns involved in adaptation process are separately treated and managed. The application being adapted is specified and executed separately from the adaptation policies. Adaptation policies are specified

separately and managed independently of other concerns. We have implemented our approach and the outcome of this is design and implementation of PCAA infrastructure. The infrastructure comprises three components: (1) Reusable reconfiguration management component, (2) Policy system and (3) Context simulator widgets. The reusable reconfiguration management component initializes the application and performs runtime adaption. The policy system is used for specification, enforcement and dynamic management of adaptation policies. Context simulator widgets are used to generate and send context events to the policy system. We have tested our system by developing and executing some hypothetical applications. Moreover, we have evaluated performance of our system and support for dynamic modifiability of adaptation policies. Results show that the use of polices in our proposed approach provides a greater flexibility (dynamic addition to and removal of adaptation policies from the system) with a minimal performance overhead.

6.1.1 SUMMARY OF CONTRIBUTIONS OF THIS THESIS

A summary of the research contributions is given below:

- **Software architectural adaptation support:** This includes implementation of reusable reconfiguration management component, which provides runtime support for both application initialization from the description of initial software architecture of the application and architectural adaptation.

- **Dynamic programmability of context-aware applications:** Dynamic programmability of applications, which is an essential requirement of the pervasive computing environments, is provided with support of dynamic modifiability of adaptation policies.

- **Separation of concerns:** Separating all the adaption concerns (i.e. application being adapted, adaptation mechanisms and adaptation policies) and making them external to each other provides ease of development and reduces complexity involved in the development of adaptive context-aware applications.

6.2 FUTURE WORK

The research work presented in this thesis has potential to be extended. In this regard, we outline some of the future directions below:

- Currently PCAA infrastructure supports one adaptation mechanism, compositional adaptation to adapt the application. Adaptation support provided by our PCAA can be broadened by integrating another adaptation mechanism, called parametric adaptation to fulfill adaptation needs of the application.

- In the current PCAA infrastructure component model, a component can provide only one service to other components, where as a component can require more than one services of other components. PCAA infrastructure component model should be refined to add support for a component to provide more than one services to other components.

- The small configuration language can be extended. The language currently supports only basic constructs for architectural actions. It requires extending the small configuration language to add more architectural commands.

- PCAA infrastructure currently provides context simulator widgets to send context events to the policy system. It can be extended by adding support for context monitoring service which will provide contextual information with real sensors deployed in the environment.

- One inherent disadvantage of policies is policy conflicts. There needs to be a provision for solution of policy conflicts.

REFERENCES

Abowd, G. D., A. K. Dey, P. J. Brown, N. Davies, M. Smith and P. Steggles (1999). Towards a Better Understanding of Context and Context-Awareness. Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing. Karlsruhe, Germany, Springer-Verlag**:** 304-307.

Aksit, M. and Z. Choukair (2003). Dynamic, adaptive and reconfigurable systems overview and prospective vision. Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on, IEEE.

Allen, R., R. Douence and D. Garlan (1998). Specifying and analyzing dynamic software architectures. Fundamental Approaches to Software Engineering. E. Astesiano, Springer Berlin Heidelberg. **1382:** 21-37.

Appeltauer, M., R. Hirschfeld, M. Haupt, J. Lincke and M. Perscheid (2009). A comparison of context-oriented programming languages. International Workshop on Context-Oriented Programming, ACM.

Appeltauer, M., R. Hirschfeld, M. Haupt and H. Masuhara (2011). "ContextJ: Context-oriented programming with Java." Information and Media Technologies **6**(2): 399-419.

Appeltauer, M., R. Hirschfeld and T. Rho (2008). Dedicated programming support for context-aware ubiquitous applications. Mobile Ubiquitous Computing, Systems, Services and Technologies, 2008. UBICOMM'08. The Second International Conference on, IEEE.

Baker, J. and W. Hsieh (2002). Runtime aspect weaving through metaprogramming. Proceedings of the 1st international conference on Aspect-oriented software development, ACM.

Brown, P. J., J. D. Bovey and C. Xian (1997). "Context-aware applications: from the laboratory to the marketplace." Personal Communications, IEEE **4**(5): 58-64.

Chalmers, D., N. Dulay and M. Sloman (2004). "A framework for contextual mediation in mobile and ubiquitous computing applied to the context-aware adaptation of maps." Personal Ubiquitous Comput. **8**(1): 1-18.

Chen, G. and D. Kotz (2000). A survey of context-aware mobile computing research, Technical Report TR2000-381, Dept. of Computer Science, Dartmouth College.

Costanza, P. and R. Hirschfeld (2005). Language constructs for context-oriented programming: an overview of ContextL. Proceedings of the 2005 symposium on Dynamic languages. San Diego, California, ACM**:** 1-10.

Dashofy, E. M., A. v. d. Hoek and R. N. Taylor (2002). An infrastructure for the rapid development of XML-based architecture description languages. Proceedings of the 24th International Conference on Software Engineering. Orlando, Florida, ACM**:** 266-276.

Dashofy, E. M., A. v. d. Hoek and R. N. Taylor (2002). Towards architecture-based self-healing systems. Proceedings of the first workshop on Self-healing systems. Charleston, South Carolina, ACM**:** 21-26.

Dashofy, E. M., A. v. d. Hoek and R. N. Taylor (2005). "A comprehensive approach for the development of modular software architecture description languages." ACM Transactions on Software Engineering and Methodology (TOSEM) **14**(2): 199-245.

David, P.-C. and T. Ledoux (2003). Towards a Framework for Self-adaptive Component-Based Applications. Distributed Applications and Interoperable Systems. J.-B. Stefani, I. Demeure and D. Hagimont, Springer Berlin Heidelberg. **2893:** 1-14.

de Oliveira Guimarães, J. (1998). Reflection for statically typed languages. ECOOP'98—Object-Oriented Programming, Springer**:** 440-461.

Dhomeja, L. D. (2011). Supporting policy-based contextual reconfiguration and adaptation in ubiquitous computing, PhD Thesis, University of Sussex.

Floch, J., C. Frà, R. Fricke, K. Geihs, M. Wagner, J. Lorenzo, E. Soladana, S. Mehlhase, N. Paspallis and H. Rahnama (2013). "Playing MUSIC—building context-aware and self-adaptive mobile applications." Software: Practice and Experience **43**(3): 359-388.

Fox, J. and S. Clarke (2009). Exploring approaches to dynamic adaptation. Proceedings of the 3rd International DiscCoTec Workshop on Middleware-Application Interaction. Lisbon, Portugal, ACM**:** 19-24.

Garlan, D. (2000). Software architecture: a roadmap. Proceedings of the Conference on The Future of Software Engineering. Limerick, Ireland, ACM**:** 91-101.

Garlan, D., S.-W. Cheng, A.-C. Huang, B. Schmerl and P. Steenkiste (2004). "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure." Computer **37**(10): 46-54.

Garlan, D. and B. Schmerl (2002). Model-based adaptation for self-healing systems. Proceedings of the first workshop on Self-healing systems. Charleston, South Carolina, ACM**:** 27-32.

Georgas, J. C. and R. N. Taylor (2008). Policy-based self-adaptive architectures: a feasibility study in the robotics domain. Proceedings of the 2008 international workshop on Software engineering for adaptive and self-managing systems. Leipzig, Germany, ACM**:** 105-112.

Georgas, J. C. and R. N. Taylor (2009). Policy-based architectural adaptation management: Robotics domain case studies. Software Engineering for Self-Adaptive Systems, Springer**:** 89-108.

Gjørven, E., F. Eliassen, K. Lund, V. S. W. Eide and R. Staehli (2006). Self-adaptive systems: A middleware managed approach. Self-Managed Networks, Systems, and Services, Springer**:** 15-27.

Gui, N., V. De Florio and T. Holvoet (2013). "Transformer: an adaptation framework supporting contextual adaptation behavior composition." Software: Practice and Experience **43**(8): 937-967.

Gui, N., V. D. Florio, H. Sun and C. Blondia (2011). "Toward architecture-based context-aware deployment and adaptation." J. Syst. Softw. **84**(2): 185-197.

Hallsteinsen, S., J. Floch and E. Stav (2005). A middleware centric approach to building self-adapting systems. Software Engineering and Middleware, Springer**:** 107-122.

Hill, E. F. (2003). Jess in Action: Rule-Based Systems in Java Manning Publications Co, Greenwich, C, USA.

Hirschfeld, R., P. Costanza and M. Haupt (2008). An Introduction to Context-Oriented Programming with ContextS. Generative and Transformational Techniques in Software Engineering II. L. Ralf, mmel, V. Joost, Jo and S. o, Springer-Verlag**:** 396-407.

Inverardi, P. and A. L. Wolf (1995). "Formal specification and analysis of software architectures using the chemical abstract machine model." Software Engineering, IEEE Transactions on **21**(4): 373-386.

Kasten, E. P., P. K. McKinley, S. Sadjadi and R. Stirewalt (2002). Separating introspection and intercession to support metamorphic distributed systems. Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on, IEEE.

Le Métayer, D. (1998). "Describing software architecture styles using graph grammars." Software Engineering, IEEE Transactions on **24**(7): 521-533.

Löwis, M. v., M. Denker and O. Nierstrasz (2007). Context-oriented programming: beyond layers. Proceedings of the 2007 international conference on Dynamic languages: in conjunction with the 15th International Smalltalk Joint Conference 2007. Lugano, Switzerland, ACM**:** 143-156.

Luckham, D. C. and J. Vera (1995). "An event-based architecture definition language." Software Engineering, IEEE Transactions on **21**(9): 717-734.

Magee, J. and J. Kramer (1996). "Dynamic structure in software architectures." SIGSOFT Softw. Eng. Notes **21**(6): 3-14.

McKinley, P. K., S. M. Sadjadi, E. P. Kasten and B. H. Cheng (2004). "A taxonomy of compositional adaptation." Rapport Technique numéroMSU-CSE-04-17, juillet.

Mukhija, A. (2007). CASA A Framework for Dynamic Adaptive Applications, PhD thesis, University of Zurich, Switzerland.

Oreizy, P., M. M. Gorlick, R. N. Taylor, D. Heimbigner, G. Johnson, N. Medvidovic, A. Quilici, D. S. Rosenblum and A. L. Wolf (1999). "An Architecture-Based Approach to Self-Adaptive Software." IEEE Intelligent Systems **14**(3): 54-62.

Oreizy, P., N. Medvidovic and R. N. Taylor (1998). Architecture-based runtime software evolution. Proceedings of the 20th international conference on Software engineering. Kyoto, Japan, IEEE Computer Society**:** 177-186.

Oreizy, P., N. Medvidovic and R. N. Taylor (2008). Runtime software adaptation: framework, approaches, and styles. Companion of the 30th international conference on Software engineering, ACM.

Oreizy, P. and R. Taylor (1998). On the Role of Software Architectures in Runtime System Reconfiguration. Proceedings of the International Conference on Configurable Distributed Systems, IEEE Computer Society**:** 61.

Salber, D., A. K. Dey and G. D. Abowd (1999). The context toolkit: aiding the development of context-enabled applications. Proceedings of the SIGCHI conference on Human Factors in Computing Systems. Pittsburgh, Pennsylvania, United States, ACM**:** 434-441.

Satyanarayanan, M. (2001). "Pervasive computing: vision and challenges." Personal Communications, IEEE **8**(4): 10-17.

Schilit, B., N. Adams and R. Want (1994). Context-Aware Computing Applications. Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications, IEEE Computer Society**:** 85-90.

Schmidt, G. (2008). "ContextR & ContextWiki, Master's thesis." Hasso-Plattner-Institut, Potsdam.

Schubert, C. (2008). "ContextPy & PyDCL – Dynamic Contract Layers for Python, Master's thesis." Hasso-Plattner-Institut, Potsdam.

Shaw, M. and D. Garlan (1996). "Software architecture: perspectives on an emerging discipline."

Shaw, M. and D. Garlan (1996). Software architecture: perspectives on an emerging discipline, Prentice Hall Englewood Cliffs.

Sousaand, J. and D. Garlan (2002). Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance, Kluwer, B.V.**:** 29-43.

Tatsubori, M., S. Chiba, M.-O. Killijian and K. Itano (2000). OpenJava: A Class-Based Macro System for Java. Reflection and Software Engineering. W. Cazzola, R. Stroud and F. Tisato, Springer Berlin Heidelberg. **1826:** 117-133.

Taylor, R. N., N. Medvidovic, K. M. Anderson, J. E. James Whitehead and J. E. Robbins (1995). A component- and message-based architectural style for GUI software. Proceedings of the 17th international conference on Software engineering. Seattle, Washington, United States, ACM**:** 295-304.

Taylor, R. N., N. Medvidovic and E. M. Dashofy (2009). Software Architecture: Foundations, Theory, and Practice, Wiley Publishing.

Twidle, K., N. Dulay, E. Lupu and M. Sloman (2009). Ponder2: A policy system for autonomous pervasive environments. Autonomic and Autonomous Systems, 2009. ICAS'09. Fifth International Conference on, IEEE.

Twidle, K., E. Lupu, N. Dulay and M. Sloman (2008). Ponder2-a policy environment for autonomous pervasive systems. Policies for Distributed Systems and Networks, 2008. POLICY 2008. IEEE Workshop on, IEEE.

Weiser, M. (September 1991). "The computer for the 21st century." Scientific American. **265**: 94-104.

APPENDIX A

PCAA INFRASTRUCTURE COMPONENT MODEL API

A.1 COMPONENT INTERFACE

```
package pcaapc.api;

import java.util.Hashtable;

public interface IComponent {

	public void initialize();

	public Output doRequired(String port, Input in);

	public void doProvided(String port);

	public void setConnector(IConnector connector, String port);

	public Hashtable<String, IConnector> getConnectors();

}
```

A.2 COMPONENT CLASS

```
package pcaapc.api;

import java.util.Hashtable;

public class Component implements IComponent {

	Hashtable<String, IConnector> connectors;

	public Component() {
```

```
            this.connectors = new java.util.Hashtable<String, IConnector>();
        }
        public void initialize() {
        }
        public Output doRequired(String port, Input in) {
            Output o;
            o = this.connectors.get(port).doRequired(port, in);
            return o;
        }
        public void doProvided(String port) {
            this.connectors.get(port).doProvided(port);
        }
        public void setConnector(IConnector connector, String port) {
            this.connectors.put(port, connector);
        }
        public Hashtable<String, IConnector> getConnectors() {
            return this.connectors;
        }
} // end class
```

A.3 CONNECTOR INTERFACE

```
package pcaapc.api;
```

```
public interface IConnector {
	public Output doRequired(String port, Input in);
	public void doProvided(String port);
	public void setRequired(IComponent required);
	public void setProvided(IComponent provided);
	public IComponent getRequired();
	public IComponent getProvided();
}
```

A.4 CONNECTOR CLASS

```
package pcaapc.api;
public class Connector implements IConnector {
	private IComponent required;
	private IComponent provided;
	public Connector() {
	}
	public Output doRequired(String port, Input in) {
		Output o;
		o = this.required.doRequired(port, in);
		return o;
	}
	public void doProvided(String details) {
```

```
            this.provided.doProvided(details);
        }
        public void setRequired(IComponent required) {
            this.required = required;
        }
        public void setProvided(IComponent provided) {
            this.provided = provided;
        }
        public IComponent getRequired() {
            return this.required;
        }
        public IComponent getProvided() {
            return this.provided;
        }
} // end class
```

A.5 INPUT INTERFACE

```
package pcaapc.api;
public interface Input {
}
```

A.6 OUTPUT INTERFACE

```
package pcaapc.api;
```

```
public interface Output {

}
```

A.7 IRUNNER INTERFACE

```
public interface IRunner extends Runnable {

	//public void start();

}
```

APPENDIX B

CODE OF HYPOTHETICAL EXAMPLE APPLICATIONS

B.1 LOCATION BASED MESSAGE DELIVERY APPLICATION

B.1.1 MESSAGEFORWARDER COMPONENT

```
package pc_app3;

import pcaapc.api.*;

public class MessageForwarder extends Component  implements IRunner {

        public void doProvided(String details) {

                forwardMessage();

        }

        public void forwardMessage() {

         do {

                // get messages from receiver

                Output o = doRequired("mr", new Input(){});

                Data d = (Data)o;

                DataIn di = new DataIn(d.getData());

                doRequired("mf", di); // Send to device for display

                delay();
```

```
        } while(true);
    }
    public void delay() {
        try {
            Thread.sleep(2000);
        }catch(Exception ex) {}
    }
    public void run() {
        doProvided("");
    }
}
```

B.1.2 MESSAGERECEIVER COMPONENT

```
package pc_app3;
import pcaapc.api.*;
public class MessageReceiver extends Component {
    private int i=1;
    // This method imitates data to be fetched from remote source
    public Data getData() {
        return new Data("Message "+i++);
    }
```

```
	public Output doRequired(String port, Input in) {
		return getData();
	}
}
```

B.1.3 SMARTPHONE COMPONENT

```
package pc_app3;
import java.awt.BorderLayout;
import java.awt.Font;
import pcaapc.api.Component;
import pcaapc.api.Input;
import pcaapc.api.Output;
import javax.swing.*;
public class SmartPhone extends Component {
	JFrame f = null;
	JLabel label = null;
	public SmartPhone() {
		f = new JFrame();
		f.setTitle("Smart Phone");
		this.label = new JLabel();
		this.label.setFont(new Font("Arial", 25, 25));
		f.getContentPane().add(BorderLayout.CENTER,this.label);
```

```
                f.pack();
                f.setSize(400, 200);
                f.setVisible(true);
        }
        private void display(Input in) {
                DataIn d = (DataIn)in;
                this.label.setText(d.getData());
        }
        public Output doRequired(String port, Input in) {
                display(in);
                return null;
        }
}
```

B.1.4 SMARTTV COMPONENT

```
package pc_app3;
import java.awt.BorderLayout;
import java.awt.Font;
import pcaapc.api.Component;
import pcaapc.api.Input;
import pcaapc.api.Output;
import javax.swing.*;
```

```
public class SmartTV extends Component {
	JFrame f = null;
	JLabel label = null;
	public SmartTV() {
		f = new JFrame();
		f.setTitle("Smart TV");
		this.label = new JLabel();
		this.label.setFont(new Font("Arial", 25, 25));
		f.getContentPane().add(BorderLayout.CENTER,this.label);
		f.pack();
		f.setSize(400, 200);
		f.setVisible(true);
	}
	private void display(Input in) {
		DataIn d = (DataIn)in;
		this.label.setText(d.getData());
	}
	public Output doRequired(String port, Input in) {
		display(in);
		return null;
	}
}
```

B.1.5 CONNECTOR

```
package pc_app3;

import pcaapc.api.Connector;

public class Conn extends Connector {

}
```

B.1.6 DATA CLASS

```
package pc_app3;

import pcaapc.api.Output;

public class Data implements Output {

	String s;

	Data(String s) {

		this.s=s;

	}

	String getData() {

		return this.s;

	}

}
```

B.1.7 DATAIN CLASS

```
package pc_app3;

import pcaapc.api.Input;
```

```
public class DataIn implements Input {
	String s;
	DataIn(String s) {
		this.s=s;
	}
	String getData() {
		return this.s;
	}
}
```

B.1.8 RENDEREDDATA CLASS

```
package pc_app3;
import pcaapc.api.Output;
public class RenderedData implements Output {
	String s;
	RenderedData(String s) {
		this.s=s;
	}
	String getData() {
		return this.s;
	}
}
```

B.2 SMART NOTICE BOARD APPLICATION

B.2.1 SMARTNOTICEBOARD COMPONENT

```
package pc_app2;

import java.awt.BorderLayout;

import java.awt.Font;

import pcaapc.api.Component;

import pcaapc.api.IRunner;

import pcaapc.api.Input;

import pcaapc.api.Output;

import javax.swing.*;

public class SmartNoticeBoard extends Component implements IRunner {

	JFrame f = null;

	JLabel label = null;

	public SmartNoticeBoard() {

		f = new JFrame();

		f.setTitle("Smart Notice Board");

		this.label = new JLabel();

		this.label.setFont(new Font("Arial", 25, 25));

		f.getContentPane().add(BorderLayout.CENTER,this.label);

		f.pack();

		f.setSize(400, 200);
```

```
            f.setVisible(true);
        }
        private void display() {
            Output o = doRequired("vp", new Input(){});
            RenderedData d = (RenderedData)o;
            this.label.setText(d.getData());
        }
        public void doProvided(String details) {
            while(true) {
                display();
                try { Thread.sleep(1000);}catch(Exception e){}
            }
        }
        public void run() {
            doProvided("");
        }
}
```

B.2.2 STUDENT DATA COMPONENT

```
package pc_app2;
import pcaapc.api.*;
public class StudentData extends Component {
```

```
	public Data sendData() {
		int i = (int)(Math.random()*100);
		return new Data("Student Data " + i);
	}
	public Output doRequired(String port, Input in) {
		return sendData();
	}
}
```

B.2.3 TEACHERDATA COMPONENT

```
package pc_app2;
import pcaapc.api.*;
public class TeacherData extends Component {
	public Data sendData() {
		int i = (int)(Math.random()*100);
		return new Data("Teacher Data " + i);
	}
	public Output doRequired(String port, Input in) {
		return sendData();
	}
}
```

B.2.4 VIEW COMPONENT 1

```
package pc_app2;

import pcaapc.api.*;

public class View1 extends Component {

	public RenderedData renderData() {

		Output o = super.doRequired("dp", new Input(){});

		Data d = (Data)o;

		return new RenderedData("View 1 => (" + d.getData() + ")");

	}

	public Output doRequired(String port, Input in) {

		return renderData();

	}

}
```

B.2.5 VIEW COMPONENT 2

```
package pc_app2;

import pcaapc.api.*;

public class View2 extends Component {

	public RenderedData renderData() {

		Output o = super.doRequired("dp", new Input(){});

		Data d = (Data)o;

		return new RenderedData("View 2 => [" + d.getData() + "]");

	}
```

```
	public Output doRequired(String port, Input in) {
		return renderData();
	}
}
```

B.2.6 CONNECTOR

```
package pc_app2;
import pcaapc.api.Connector;
public class Conn extends Connector {
}
```

B.2.7 DATA CLASS

```
package pc_app2;
import pcaapc.api.Output;
public class Data implements Output {
	String s;
	Data(String s) {
		this.s=s;
	}
	String getData() {
		return this.s;
	}
}
```

B.2.8 RENDEREDDATA CLASS

```
package pc_app2;

import pcaapc.api.Output;

public class RenderedData implements Output {

	String s;

	RenderedData(String s) {

		this.s=s;

	}

	String getData() {

		return this.s;

	}

}
```

B.3 CONTEXT-AWARE COMPRESSION SERVER

B.3.1 DATASTORE COMPONENT

```
package pc_app;

import pcaapc.api.Component;

import pcaapc.api.Input;

import pcaapc.api.Output;

public class DataStore extends Component {

	private String data[] = {"Data1","Data2","Data3","Data4","Data5"};

	private int pointer = 0;
```

```
	private int limit = 5;
	public String getData() {
		if(pointer == limit)
			pointer = 0;
		return "DS-1["+data[pointer++]+"]";
	}
	public Output doRequired(String port, Input in) {
		DataOut d = new DataOut(getData());
		return d;
	}
}
```

B.3.2 PROVIDER COMPONENT

```
package pc_app;
import java.awt.BorderLayout;
import java.awt.Font;
import javax.swing.JFrame;
import javax.swing.JLabel;
import pcaapc.api.*;
public class Provider extends Component implements IRunner {
	JFrame f = null;
	JLabel label = null;
```

```
public Provider() {
        f = new JFrame();
        f.setTitle("Server application");
        this.label = new JLabel();
        this.label.setFont(new Font("Arial", 25, 25));
        f.getContentPane().add(BorderLayout.CENTER,this.label);
        f.pack();
        f.setSize(1400, 200);
        f.setVisible(true);
}
public void process() {
        Output o = doRequired("ds", new Input(){});
        DataOut d = (DataOut) o;
        ForCompress fc = new ForCompress(d.getData());
        Output o2 = doRequired("dc", fc);
        Compressed c = (Compressed) o2;
        String data = c.getData();
        this.label.setText(data);
        //System.out.println(data);
}
public void provide() {
```

```
		while(true) {
			process();
			try { Thread.sleep(1000);}catch(Exception e){}
		}
	}

	public void doProvided(String details) {
			provide();
	}
	public void run() {
		doProvided("");
	}
}
```

B.3.3 COMPRESSOR COMPONENT 1

```
package pc_app;
import pcaapc.api.Component;
import pcaapc.api.Output;
import pcaapc.api.Input;
public class Compressor extends Component {
	public String compress(String data) {
		return "Compressor 1 => Compressed "+data+" with compression
		 ratio 2 (20MB to 10MB)";
```

```
	}

	public Output doRequired(String port, Input in) {

		ForCompress fc = (ForCompress)in;

		String d = compress(fc.getData());

		return new Compressed(d);

	}

}
```

B.3.4 COMPRESSOR COMPONENT 2

```
package pc_app;

import pcaapc.api.Component;

import pcaapc.api.Output;

import pcaapc.api.Input;

public class Compressor2 extends Component {

	public String compress(String data) {

		return "Compressor 2 => Compressed "+data+" with compression
			ratio 5 (20MB to 4MB)";

	}

	public Output doRequired(String port, Input in) {

		ForCompress fc = (ForCompress)in;

		String d = compress(fc.getData());

		return new Compressed(d);

	}
```

```
}
```

B.3.5 CONNECTOR

```
package pc_app;

import pcaapc.api.Connector;

public class Conn extends Connector {

}
```

B.3.6 DATAOUT CLASS

```
package pc_app;

import pcaapc.api.Output;

public class DataOut implements Output {

	String s;

	DataOut(String s) {

		this.s=s;

	}

	String getData() {

		return this.s;

	}

}
```

B.3.7 COMPRESSED CLASS

```
package pc_app;

import pcaapc.api.Output;
```

```
public class Compressed implements Output {
        String s;
        Compressed(String s) {
                this.s=s;
        }
        String getData() {
                return this.s;
        }
} // end class
```

B.3.8 FORCOMPRESS CLASS

```
package pc_app;
import pcaapc.api.Input;
public class ForCompress implements Input {
        String s;
        ForCompress(String s) {
                this.s=s;
        }
        String getData() {
                return this.s;
        }
} // end class
```