ALPH: A DSAL-based Programming Model for Complexity Management in Pervasive Healthcare Applications

Jennifer Munnelly

A thesis submitted to the University of Dublin, Trinity College

in fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

November 2009

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work. I agree that Trinity College Library may lend or copy this thesis upon request.

Jennifer Munnelly

Dated: May 23, 2010

Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

____
Jennifer Munnelly

Dated: May 23, 2010

Abstract

Healthcare information, for example patient records, must be available to appropriate professionals at all times. The application of pervasive technology to healthcare means that those professionals are using such technology while mobile, but still, healthcare information must remain available at all times and across multiple locations. "Pervasive healthcare" has emerged as the field that is concerned with the application of pervasive computing to healthcare, increasing productivity, reducing human error and increasing interoperability between various healthcare areas and facilities.

However, the development of pervasive healthcare applications has proved to be significantly more complex than traditional healthcare applications. Two concerns are of particular interest to this thesis. Firstly, these applications must support the storage and exchange of healthcare information in a mobile, distributed environment. Secondly, adapting to such a changing environment requires the handling of contextual information, such as location information or device heterogeneity. Incorporating these concerns into the development process increases its complexity both for new applications and for existing applications to be upgraded with pervasive functionality.

The focus of this thesis is to reduce the level of complexity in pervasive healthcare application code. The complexity is analysed against two dimensions; difficulties with modularity and inappropriate levels of abstraction. Poor modularity emerges because many pervasive healthcare concerns cut across the entire system. Such concerns are referred to as "crosscutting" and are difficult to encapsulate using traditional programming models, leading to complicated, unmanageable code. Inappropriate levels of abstraction in the implementation of pervasive healthcare applications emerge when using general purpose languages (GPLs),

whose constructs tend to be at a low-level of abstraction. This means that developers need significant domain knowledge to produce the required verbose, low-level code that is neither expressive nor semantically intuitive.

Aspect-oriented programming (AOP) provides modularisation capabilities for crosscutting concerns and has successfully been applied to the modularisation of a selection of pervasive computing concerns. However, it has not been considered for the broad set of concerns required by pervasive healthcare applications. Domain-specific languages (DSLs) provide high-level, expressive constructs that encapsulate domain knowledge, reducing the requirement for domain knowledge, but its application in healthcare has been limited. One notable exception is MUMPS, a language that provides database functionality that was previously applied to healthcare information. However, it does not provide any constructs specific to healthcare, mobility or adaptation.

In this thesis we combine these two software engineering techniques in a programming language called ALPH (Aspect Language for Pervasive Healthcare). The research question addressed was whether the collaboration of aspect-oriented programming and domain-specific languages significantly reduces complexity in pervasive healthcare application code. ALPH provides a set of constructs for thirteen abstractions derived from analysis of the pervasive healthcare domain. These abstractions model concerns that reoccur in pervasive healthcare applications and that have exhibited crosscutting characteristics. To modularise these concerns, they are implemented using an aspect-oriented language and assembled in a library of modular pervasive healthcare aspects. To achieve the benefits of a domain-specific language, the ALPH language syntax and semantics have been formally specified as a grammar and a compiler has been created from this grammar. ALPH programs are constructed from these high-level, expressive domain-specific constructs, parameterised according to application requirements. The programs are then parsed by the compiler and the generative compilation process triggers the use of code from the library of modular aspects. A configured aspect implementation is produced and woven into the base application at relevant points using the aspect language weaver. The result is a complete compiled and executable pervasive healthcare application. In the development process, developers are aware only of the base

application and the high-level ALPH constructs reducing the requirement for domain knowledge.

ALPH has been empirically evaluated through comparative studies between standard object-oriented and ALPH implementations of multiple applications. Applications were selected based on their inclusion of pervasive healthcare concerns and on their base language suitability. Metrics were used to measure variations in complexity. Common AOP code metrics were used to measure modularity. Common code metrics were used to measure abstraction and an appropriate metric was selected to measure DSL expressiveness. Results show reductions in elements of complexity. The ALPH model improves abstraction. However, although modularity is improved in the base application, dependencies introduced by AOP negatively impact modularity when viewed from a larger perspective.

Publications Related to this Ph.D.

[1] Jennifer Munnelly and Siobhán Clarke. HL7 Healthcare Information Management Using Aspect-Oriented Programming. In *Proceedings of the 22nd IEEE Symposium on Computer-Based Medical Systems* . (CBMS 2009), IEEE, Albequerque, NM, USA, August 2009.

[2] Jennifer Munnelly and Siobhán Clarke. Domain-Specific Language for Ubiquitous Healthcare. In *Proceedings of the Third International Conference on Pervasive Computing and Applications*, (ICPCA 2008), IEEE, Alexandria, Egypt, October 2008.

[3] Jennifer Munnelly and Siobhán Clarke. Infrastructure for Ubiquitous Computing: Improving Quality with Modularisation. In *the 7th Workshop on Aspects, Components, and Patterns For Infrastructure Software*, (ACP4IS '08 (AOSD 08)), ACM, Brussels, Belgium, April 2008.

[4] Jennifer Munnelly and Siobhán Clarke. ALPH: A Domain-Specific Language for Crosscutting Pervasive Healthcare Concerns. In *the 2nd Workshop on Domain Specific Aspect Languages.*, (DSAL '07 (AOSD 07)), ACM, Vancouver, British Columbia, Canada, March 2007.

[5] Jennifer Munnelly, Serena Fritsch, and Siobhán Clarke. An Aspect-Oriented Approach to the Modularisation of Context. In *Proceedings of the Fifth Annual IEEE International Conference on Pervasive Computing and Communications.*, (PerCom 2007)), IEEE, White Plains, NY, USA, March 2007.

[6] Patricia Gómez Bello, Ignacio Aedo, Fausto Sainz, Paloma Daz, Jennifer Munnelly and Siobhán Clarke. Improving Communication for Mobile Devices in Disaster Response. In *the International Workshop on Mobile Information Technology for Emergency Response.*, (MobileResponse 2007)), Springer, LNCS 4458.

[7] Serena Fritsch, Jennifer Munnelly and Siobhán Clarke. Towards a Domain-Specific AOP language for Ubiquitous Computing. In *the 1st Workshop on Open and Dynamic Aspect Languages.*, (ODAL 2006 (AOSD 2006))), ACM, Brussels, Belgium, 2006.

[8] F. Sanen, E. Truyen, B. De Win, W. Joosen, N. Boucke, T. Holvoet, N. Loughran, A. Rashid, R. Chitchyan, N. Leidenfrost, J. Fabry, N. Cacho, A. Garcia, A. Jackson, N. Hatton, J. Munnelly, S. Fritsch, S. Clarke, M. Amor, L. Fuentes, and C. Canal. A Domain Analysis Of Key Concerns Known And New Candidates. In *AOSD Europe Deliverable 45.*, March, 2006.

Other Presentations

[i] Jennifer Munnelly and Siobhán Clarke. ALPH: A Domain-Specific Aspect Language for Pervasive Healthcare. Participation & Poster, At *ACM and Microsoft Research Student Research Competition (SRC) at the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation.* (PLDI 2009).

[ii] Jennifer Munnelly and Siobhán Clarke. HL7 Healthcare Information Management Using Aspect-Oriented Programming. Poster, At *Proceedings of the 22nd IEEE Symposium on Computer-Based Medical Systems* . (CBMS 2009).

[iii] Jennifer Munnelly, and Siobhán Clarke. A Domain-Specific Language for Crosscutting Pervasive Healthcare Concerns, Poster, At *Sixth International Conference on Aspect-Oriented Software Development.* (AOSD 2007).

[iv] Jennifer Munnelly, Serena Fritsch and Siobhán Clarke. A Domain-Specific Aspect-Oriented Framework for Ubiquitous Computing, Poster, At *Fifth International Conference on Aspect-Oriented Software Development.* (AOSD 2006).

Contents

Abstract **iv**

List of Tables **xviii**

List of Figures **xix**

Chapter 1 Introduction **1**

1.1 Background 1

1.2 Pervasive Computing 2

1.2.1 Mobility 2

1.2.2 Context-Awareness 3

1.3 Pervasive Healthcare 4

1.4 Motivation 6

1.4.1 Difficulties with Modularity 7

1.4.2 Inappropriate Levels of Abstraction 7

1.5 Approach 8

1.5.1 Complexity Approaches 8

1.5.1.1 Modularity 8

1.5.1.2 Abstraction 9

1.5.2 Modular Programming 9

1.5.3 Aspect-Oriented Programming 9

1.5.4 Domain-Specific Languages 10

1.5.5 ALPH Model 10
1.6 Contributions 14
1.7 Thesis Outline 15

Chapter 2 Related Work **16**
2.1 Pervasive Healthcare Application Development 17
2.1.1 Centre for Pervasive Healthcare 17
2.1.2 HL7 18
2.1.3 Electronic Health Records 19
2.2 Modularisation in Pervasive Healthcare 20
2.2.0.1 Separation of concerns 21
2.2.0.2 Crosscutting concerns 21
2.2.1 Aspect-Oriented Programming 22
2.2.2 AOP in Pervasive Healthcare 23
2.2.3 Context Adaptation 23
2.2.4 Distribution 24
2.2.5 Persistence 24
2.2.6 Quality of Service 25
2.2.7 Mobility 25
2.3 Abstraction in Pervasive Healthcare 25
2.3.1 Domain-Specific Languages 26
2.3.1.1 DSL Benefits 26
2.3.1.2 Limitations 28
2.3.2 DSLs in Pervasive Healthcare 28
2.3.2.1 MUMPS 29
2.3.2.2 Reformatting of HL7 Messages 30
2.3.2.3 Apache Camel 30
2.3.3 DSLs in Pervasive Computing 30
2.3.3.1 YABS 31
2.3.3.2 AmbientTalk 31

2.3.3.3 Indus 32
2.3.3.4 PLUE 32
2.3.3.5 Pantaxou 33
2.3.3.6 DiaSpec 33
2.3.3.7 Scooby 34
2.3.3.8 PerIDL 34
2.3.3.9 Olympus 34
2.3.3.10 Context Adaptation 35
2.4 Domain-Specific Aspect Languages 35
2.4.0.11 AOPAmI 36
2.4.0.12 Distributed Definition Language 37
2.4.0.13 D Language Framework 37
2.4.0.14 AWED 37
2.5 Middleware 38
2.6 Summary 39

Chapter 3 Domain Analysis **43**
3.1 Overview of Model 43
3.2 Methodology 44
3.2.1 ALPH Design 45
3.3 Domain Analysis 46
3.3.1 Requirements for Pervasive Healthcare Applications 47
3.3.2 Application Literature Review 49
3.3.3 Pervasive Healthcare Case Studies 49
3.3.4 Pervasive Healthcare Codebases 51
3.4 Crosscutting concerns 53
3.5 Concern Selection 55
3.6 Pervasive Healthcare Concerns 56
3.6.1 Mobility 57
3.6.1.1 Distribution 57

3.6.1.2 Communication 58
3.6.1.3 Network Roaming 58
3.6.1.4 Software Roaming 58
3.6.1.5 Service Discovery 59
3.6.1.6 Device Discovery 59
3.6.1.7 Limited Connectivity 59
3.6.1.8 Location 60
3.6.1.9 Quality of Service 60
3.6.2 Context 60
3.6.2.1 Device Context 61
3.6.2.2 Location Context 61
3.6.2.3 Other Context Types 62
3.6.3 Healthcare 63
3.6.3.1 HL7 63
3.6.3.2 EHR 64
3.6.3.3 Persistence 64
3.7 Summary 65

Chapter 4 Pervasive Healthcare Aspect Library **67**

4.1 Library Overview 67
4.2 Language and Tools 69
4.2.1 Theme/UML 69
4.2.2 AspectJ 70
4.3 Concern Implementations 71
4.3.1 Distribution 71
4.3.2 Network Roaming 75
4.3.3 Quality of Service 78
4.3.4 Software Roaming 81
4.3.5 Service Discovery 83
4.3.6 Communication 85

4.3.7 Device Discovery 87
4.3.8 Limited Connectivity 89
4.3.9 Device Context 90
4.3.9.1 Reasoning 91
4.3.10 Location 93
4.3.11 HL7 96
4.3.12 EHR 98
4.3.13 Persistence 100
4.4 Summary 102

Chapter 5 ALPH Language 104

5.1 Overview 104
5.2 DSL Implementation 106
5.3 ALPHc Compiler 107
5.3.1 T-Diagrams 107
5.3.2 Compiler Generation 108
5.4 Language Processing 109
5.4.1 Components 109
5.4.2 Parsing Overview 110
5.4.3 Definition 112
5.5 ALPH Language Constructs 113
5.5.1 Terminology 113
5.5.2 Constructs 113
5.5.2.1 Distribute 114
5.5.2.2 AdaptDevice 114
5.5.2.3 HL7 115
5.5.2.4 Location 115
5.5.2.5 SoftwareRoaming 116
5.5.2.6 NetworkRoaming 116
5.5.2.7 Persist 117

5.5.2.8 QualityOfService 117
5.5.2.9 ServiceDiscovery 118
5.5.2.10 DeviceDiscovery 118
5.5.2.11 LimitedConnectivity 118
5.5.2.12 EHR 119
5.5.2.13 Communication 119
5.5.3 Parameter Values 120
5.6 Summary 120

Chapter 6 Evaluation **122**
6.1 Complexity 122
6.2 Goal-Question-Metric Approach 123
6.3 Modularisation 125
6.3.1 Conceptual Level Goals 125
6.3.1.1 Manageability 126
6.3.1.2 Maintainability 126
6.3.1.3 Comprehensibility 126
6.3.2 Operational Level Questions 127
6.3.2.1 Coupling 127
6.3.2.2 Cohesion 127
6.3.2.3 Independence 128
6.3.3 Quantitative Level Metrics 128
6.3.3.1 Coupling 129
6.3.3.2 Cohesion 129
6.3.3.3 Independence 129
6.4 Abstraction 130
6.4.1 Abstraction Benefits 130
6.4.2 Conceptual Level Goals 131
6.4.2.1 Expressiveness 131
6.4.2.2 Conciseness 132

6.4.3 Operational Level Questions 132
6.4.3.1 Syntactic Expressiveness 132
6.4.3.2 Size 132
6.4.4 Quantitative Level Metrics 133
6.4.4.1 Syntactic Expressiveness Valuation 133
6.4.4.2 Lines of Code 135
6.5 Tools 135
6.6 Applications 135
6.6.1 DBay 135
6.6.2 HL7Browser 136
6.6.3 MedHCP 137
6.6.4 Healthwatcher 139
6.6.5 Rococo 140
6.7 Results 140
6.7.1 Presentation of results 141
6.7.2 Coupling 141
6.7.3 Cohesion 149
6.7.4 Independence 152
6.7.5 Size 156
6.7.6 Expressiveness 159
6.7.7 Comparison to Context Toolkit 162
6.8 Discussion 162
6.9 Summary 164

Chapter 7 Conclusions and Future Work 166

7.1 Achievements 166
7.2 Research Question 171
7.3 Discussion 172
7.4 Future Work 173
7.4.1 Extensibility 174

7.4.2 Semantic Expressiveness 174
7.5 Summary 175

Bibliography **176**

Appendix A Generated Construct Output **206**

Appendix B ALPH Language Definition **221**

List of Tables

2.1 Summary of state of the art approaches to pervasive healthcare concerns . . . 42

3.1 Pervasive healthcare requirements in domain analysis 48
3.2 Pervasive healthcare literature analysis 50
3.3 Pervasive healthcare case study analysis 50
3.4 Pervasive Healthcare codebase analysis 52
3.5 Concern Summary 65

6.1 DBay Concerns 136
6.2 HL7Browser Concerns 137
6.3 MedHCP Concerns 138
6.4 Healthwatcher Concerns 139
6.5 Rococo Concerns 140
6.6 Frequency of lexical terms 160
6.7 Results Summary 164

List of Figures

1.1 ALPH Overview 11

2.1 Combination of areas to reduce complexity 17

3.1 Development Methodology Steps, Phases and Chapters 45
3.2 Domain Analysis 47
3.3 Concern Terms 47
3.4 Codebase with Highlighted Crosscutting Concerns 54

4.1 Library in the ALPH Model 69
4.2 Distribution Module Class Diagram 73
4.3 Distribution on Server Construction 74
4.4 Distribution on Method Invocation 75
4.5 Network Roaming Class Diagram 76
4.6 Network Roaming Module 77
4.7 Quality of Service Class Diagram 79
4.8 Quality of Service & Network Roaming 79
4.9 Quality of Service Module 80
4.10 Software Roaming Class Diagram 82
4.11 Software Roaming Module 83
4.12 Service Discovery Class Diagram 84
4.13 Service Discovery Module 85
4.14 Communication Class Diagram 86

4.15 Communication Module 87
4.16 Device Discovery Class Diagram 88
4.17 Device Discovery Module 89
4.18 Limited Connectivity Class Diagram 90
4.19 Limited Connectivity Module 91
4.20 Device Context Class Diagram 92
4.21 Device Context Module 92
4.22 Location Class Diagram 94
4.23 Location Module 95
4.24 HL7 Class Diagram 96
4.25 HL7 Module 97
4.26 EHR Package Diagram 99
4.27 Archetype Definition Language 100
4.28 EHR Module 101
4.29 Persistence Class Diagram 102
4.30 Persistence Module 103

5.1 ALPH Language in the ALPH Model 105
5.2 T Diagram Concept 107
5.3 T Diagram 107
5.4 ALPH T Diagram 108
5.5 Compiler Class Diagram 109
5.6 Compiler Sequence Diagram 111
5.7 Compiler Mapping 111
5.8 ALPH Language Construct Parameters 121

6.1 Goal-Question-Metric Approach 124
6.2 Syntactic Expressiveness 134
6.3 Mobile Clinical Assistant 138
6.4 DBay Coupling Between Modules 142

6.5 DBay Efferent Coupling 142
6.6 DBay Afferent Coupling 143
6.7 HL7Browser Efferent Coupling 144
6.8 HL7Browser Afferent Coupling 144
6.9 MedHCP Coupling between Modules 145
6.10 MedHCP Afferent Coupling 146
6.11 MedHCP Efferent Coupling 146
6.12 Healthwatcher Coupling Between Modules 147
6.13 Healthwatcher Efferent Coupling on all Packages 147
6.14 Healthwatcher Efferent Coupling on Reduced Packages 148
6.15 Healthwatcher Afferent Coupling 149
6.16 Rococo Coupling Between Modules 149
6.17 Rococo Afferent Coupling 150
6.18 Rococo Efferent Coupling 150
6.19 HL7Browser Lack of Cohesion 151
6.20 MedHCP Lack of Cohesion 151
6.21 DBay Response for a Module 152
6.22 DBay Instability 153
6.23 HL7Browser Instability 153
6.24 HL7Browser Response for a Module 154
6.25 MedHCP Response for a Module 155
6.26 MedHCP Instability 155
6.27 Healthwatcher Response for a Module 156
6.28 Healthwatcher Instability 157
6.29 Healthwatcher Instability Reductions 157
6.30 Rococo Response for a Module 158
6.31 Rococo Instability 158
6.32 DBay Lines of Code 159
6.33 Rococo Lines of Code 160

6.34 Syntactical Expressiveness 161

Chapter 1

Introduction

Pervasive healthcare applications are required to make healthcare information available to multiple users and systems in a mobile, adaptable environment [192]. These applications have proved to be complex both in their development [139] [185] and in their resulting code with poor modularity and inappropriate levels of abstraction as attributing factors. This thesis describes an approach to address these difficulties, with the goal of reducing application complexity. The proposed approach provides a means to create modular pervasive healthcare applications using a high-level, domain-specific language called ALPH (Aspect Language for Pervasive Healthcare) that reduces developer requirement for domain knowledge [187]. This introductory chapter presents the background and motivation to this work and examines challenges in pervasive healthcare application development. ALPH is introduced along with the contributions of this thesis and an outline for the remainder of this document.

1.1 Background

Like many industries, healthcare has recognised the advantages to be gained by the applied use of technology [209] [218]. Globally, technology has reengineered the healthcare industry resulting in reduced human error [17], reduced cost [262] and increased interoperability between various healthcare areas and facilities. Technology applied in the field of pervasive healthcare can improve the productivity of healthcare professionals and greatly facilitate the

delivery of a wider range of medical services to a broad spectrum of users [262].

Pervasive healthcare is a growing scientific discipline of considerable technological breadth [153] with its many and varied research questions, agendas, approaches, and methods [34]. The field of pervasive healthcare is a relatively nascent one and therefore its definition continues to evolve with the progression of the field [193]. We define "pervasive healthcare" as the application of pervasive computing technology to applications within the healthcare domain. Research in the area of pervasive healthcare is imperative and has a strong impact on the quality and effectiveness of healthcare in an aging society [153].

1.2 Pervasive Computing

To understand the implications of applying pervasive computing technologies to healthcare applications we must first understand the field of pervasive computing itself. Pervasive computing is defined as a "computing environment in which each person is continually interacting with hundreds of nearby wirelessly interconnected computers" [270]. The proliferation of mobile devices has propelled the use of pervasive computing technologies to support users via mobile applications [63]. The application of pervasive computing technology enables the development of intelligent environments with applications unconstrained by either geographical location or by incapable hardware [223]. Mobile devices have advanced capabilities to communicate with other devices and resources within the environment [235]. Supporting user activities in such environments requires that applications are capable of interpreting and reacting to information from the environment [255]. The ability for applications to react to their environment in this way is known as "context-awareness" [228] [281]. Context-awareness along with the ability to be mobile within the environment are the two key requirements for pervasive applications.

1.2.1 Mobility

The central theme in pervasive computing is the ability to incorporate mobility in applications [226]. Mobility in applications overcomes previous geographical location constraints

and enables the user to carry out their activities in a mobile environment. Two enabling requirements make this possible; mobile devices and wireless communications technology [235].

Mobile devices are generally small, portable computing devices such as mobile phones, laptops and PDAs [224]. These devices contain embedded technologies that enable them to communicate wirelessly with the environment around them [106]. The environment encompasses other devices along with services, data stores and information from sensors embedded in the physical environment. The full scope of development work with mobile devices spans a spectrum of domains and has evolved from early work in Xerox with ParcTabs and ParcPads [268] to the creation of intelligent workspaces with mobile devices [137] to the application of user based services to mobile devices e.g., tourist guides [76] up to very recent work implementing verification using the latest camera enabled mobile phones [177]. Wireless communications technologies are concerned with the ability for mobile devices to communicate with each other and with other elements in their environment. The transfer of information between mobile devices is imperative in the development of mobile applications. Wireless technology itself is the underlying enabler of communication in mobile devices [224]. Wifi, Bluetooth and Infrared provide channels of communication with differing range and network features. These wireless communication networks encourage and enable the ubiquitous nature of pervasive computing.

1.2.2 Context-Awareness

Applications adaptation to information, or context, within its mobile environment is key to pervasive computing [62]. Context has been defined 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 applications themselves" [16]. Systems are required to observe and understand contextual information in the environment and to use this information to adapt their behaviour consequently. Context-aware applications can tailor or customise their behaviour according to user and application requirements, making context-awareness a powerful tool in supporting

user activities in pervasive environments [228]. Contextual information is generally acquired by the interpretation of data from sensors within the environment. Sensors capture many different types of data from the environment e.g., location data. Applications that handle various context types can use contextual data to their advantage [190]. The applicability of context-awareness can be seen in a chronological selection of applications across many domains from early mapping applications [268] using location information, to interactive shopping assistants [47], location based tourist guides [15], applications to analyse and aid sport [23] [237] [182], automotive aids [97] and recent computer gaming applications [57].

1.3 Pervasive Healthcare

Pervasive healthcare addresses the application of pervasive computing technology to applications in the healthcare domain. Healthcare applications encompass myriad systems addressing multiple areas of healthcare from administrative activities to lab-based diagnosis systems [192]. Pervasive technologies can be applied to the healthcare domain in many forms e.g., monitoring and body sensor networks, pervasive assistive technologies and pervasive computing for hospitals [34]. The applications in these areas may be located either at home or in a healthcare facility environment to assist both patients and healthcare professionals in the delivery of healthcare services [262].

The incorporation of mobility and context in pervasive healthcare applications are essential to achieving the two focal aims of pervasive healthcare as identified by Korhonen and Bardram [153].

- To enable access to healthcare information anytime, anywhere in a mobile environment.
- To apply pervasive computing technology in order to create intelligent context-aware applications that are not constrained by physical factors such as geographic location or heterogeneous devices.

Healthcare applications have one crucial element in common; the use of and requirement for healthcare information. Healthcare information must be available at all times and in

multiple locations in a pervasive healthcare environment [65]. Information in healthcare applications varies according to application-specific details, but centres around patient based information [262] [126] [100]. The treatment of a single patient can involve many devices, such as X-Ray machines, people, such as doctors and laboratory personnel, and systems, such as administrative hospital systems [185]. Patient-based information is involved in each of these phases in the delivery of healthcare services. Information management in a hospital setting requires significant collaboration and mobility [139]. Patient health information is increasingly being transferred to electronic formats, e.g., electronic health records [74] and must remain available and accessible to the appropriate professional regardless of its format, location or accessing device. Healthcare information in a pervasive environment should be exchanged and shared in an interoperable manner enabling multiple systems or professionals access to the same data. In this thesis, we investigate the use of the International Health Level 7 [8] standard for interoperable healthcare information communication in pervasive healthcare applications.

In a pervasive environment, mobile devices can be used to access healthcare information. Applications executing on such devices require functionality relating to mobility to be considered. Mobility encompasses functionality including limited connectivity, distribution, discovery and quality of service. The incorporation of mobility into pervasive healthcare applications alleviates the location constraints of access to information in traditional healthcare systems [123]. Mobile devices executing mobile pervasive healthcare applications can access healthcare information when required within hospital and home environments [46].

Pervasive healthcare's second aim addresses the requirement for applications to be intelligent and unconstrained by physical factors such as geographic location or heterogeneous devices. Incorporating context-awareness in healthcare applications in a pervasive environment facilitates the use of contextual information from the user's environment to customise the application to best suit the needs of the user [54]. The need for context-awareness in pervasive healthcare applications has been recognised [262] [191] [236]. Customisation of pervasive healthcare applications according to contextual information e.g., situations, people and hardware, create systems that significantly improve the usefulness of such services to the

user [185].

1.4 Motivation

As described, pervasive healthcare applications are beneficial to both healthcare facilities and to patients themselves. However, the development of pervasive healthcare applications has proved to be significantly more complex than traditional healthcare applications [139] [185]. Incorporating pervasive technologies into healthcare systems to enable their execution in a distributed, mobile environment is a complex transition with many difficulties [262]. This transition involves integrating the necessary underlying infrastructure required for a fully pervasive healthcare system [262] e.g., distributed communication technology, network connectivity, interoperable information formatting.

Incorporating context handling and mobility in applications involves the consideration of a new set of concerns in the pervasive healthcare application development process. Mobility involves technical challenges that arise from an unstable environment caused by factors such as variable network connectivity quality, service availability and user/device mobility [204]. Context involves the handling of various context types and the adaptation of the application behaviour relative to the current context [229]. Application developers must consider these concerns in application development to support users and make healthcare information available in a mobile, context-aware environment. Incorporating these concerns into the development process increases its complexity [185] both for new applications and for existing applications to be upgraded with pervasive functionality [262]. New pervasive healthcare applications require complex designs for the inclusion of knowledge from healthcare, mobility and context domains. Existing healthcare applications require significant modification and/or refactoring to upgrade to support a distributed, mobile environment. “This process (of upgrading) should not interfere with the basic functioning of the current system” [262].

The focus of this thesis is to reduce the level of complexity in pervasive healthcare application code and in the development process. The complexity is analysed against two dimensions; difficulties with modularity and inappropriate levels of abstraction.

1.4.1 Difficulties with Modularity

Modularisation involves the breaking up of an application into smaller, more independent elements known as modules. Modular code reduces the complexity of applications and enables the modules to be developed in isolation as each concentrates and addresses a separate concern [201]. Modularisation promotes the use of well defined, independent modules to increase the maintainability, manageability and comprehensibility of applications [201], which in turn results in reduced complexity. Poor modularity in pervasive healthcare applications emerges because many pervasive healthcare concerns cut across the entire system. These concerns affect multiple parts of the application and are scattered and tangled with the base application functionality. Such concerns are referred to as "crosscutting" and are difficult to encapsulate using traditional programming models i.e., object-oriented programming [138]. Poor modularisation leads to highly coupled modules that present as complicated, unmanageable code [129] [201]. Pervasive healthcare applications are generally poorly modularised due to the crosscutting nature of pervasive healthcare concerns. Incorporating pervasive computing technology into healthcare applications creates a complex web of tangled code.

1.4.2 Inappropriate Levels of Abstraction

Inappropriate levels of abstraction in the implementation of pervasive healthcare applications emerge when using general purpose languages (GPLs). As pervasive healthcare applications are typically developed using traditional programming techniques, pervasive healthcare concerns are generally developed using GPLs. The constructs available in GPLs are low-level, syntactical elements that, together, provide a comprehensive language that supports the implementation of solutions to a wide number of varied problems. Developers can implement multiple applications using a single familiar language. However, the broad scope of GPLs becomes inefficient when developing applications in a specific domain [81] [132]. Implementation of domain-specific functionality using GPLs requires programming at a low-level of abstraction. It is difficult to identify the semantics of the targeted domain from the resulting verbose code. The code also lacks any intuitive syntactic or semantic relation to the target domain. This lack of expressiveness places the onus of comprehensive domain knowledge on

the application developer [179].

1.5 Approach

Software engineering approaches are applied in application development to achieve the goal of generating high-quality software [257]. This includes approaches appropriate for controlling and reducing complexity in software programs. A common set of contributing factors to complexity are accepted [27], [144], [145], [77], [51]: namely size, composition, abstraction and modularity.

1.5.1 Complexity Approaches

Complex systems require a "divide and conquer" approach to manage complexity [194]. "Today, it is generally accepted that we can cope effectively with complexity in software design through abstraction and decomposition" [176]. The acceptance of abstraction and decomposition, or modularisation, in domain literature illustrates the effectiveness of these two approaches on application complexity. Studies establishing this position [194][176] reiterate early programming analysis findings [82] [80] [277] and recent findings [241], [107], [66]. Basic principles for system construction outline abstraction and decomposition as fundamental techniques within application development [50] and combined, they have been found to facilitate application development [66]. To address the identified complexity factors we adopt the software engineering approaches of modularity and abstraction.

1.5.1.1 Modularity

The modularisation of a system into reasonably sized, encapsulated components, is a beneficial approach to managing application complexity [27]. The division of a system into modules and the maximising of module independence from other modules is a universal means of reducing complexity in any system [200]. To achieve good modularity, concerns must be modularly encapsulated i.e., including a single behaviour or goal functionality in a module. Modularity in applications allows programmers to avoid the "waterfall effect" of unwanted

side effects when modifying a particular module. Modular encapsulation reduces complexity by making sure that changes to the internal operation of a module are contained within that module [254] [252]. Modularity helps to maximise cohesion and to minimise coupling between modules [200] [232]. The effect of this is increasingly independent modules. Maximising the independence of modules is useful in reducing the complexity of a software program [200] and in the improvement of the overall application structure.

1.5.1.2 Abstraction

Abstraction is widely used as a technique to ease the development process [81]. Abstraction is a key technique in providing clarity through the selection and structuring of relevant information for the developer and is an effective approach to combating complexity [176]. Through abstraction, higher levels of programming can be provided to application developers. A provision of high-level abstractions reduce application size [179], therefore reducing complexity. An examination of major complexity metrics reveals that most of them confound the complexity of a program with its size [27] [98] [71]. Smaller modules are easier understand and are an indicator of a well designed program structure [200].

1.5.2 Modular Programming

Modular programming promotes the decomposition of systems (i.e., Parnas' notion of modularity [201]) and the separation of behaviours (i.e., Dijkstra's "separation of concerns" [83]). Object-oriented techniques support the modularisation of behaviours as classes. Advanced modularity techniques have emerged to improve the separation of concerns by introducing new units of modularity including subjects in subject-oriented programming [124] and aspects in aspect-oriented programming [138].

1.5.3 Aspect-Oriented Programming

The aspect-oriented programming (AOP) [138] paradigm provides advanced modularisation capabilities for crosscutting concerns. The encapsulation of code for a particular concern or objective within modular components or "aspects" results in the removal of duplicated code

and the separation of this code from the base functionality of the application [157]. The modularisation of this code removes the concern code that would previously have been scattered and tangled throughout the entire system, reducing coupling and increasing cohesion. Increased modularity improves code quality [56] and results in more manageable, maintainable and understandable code [201].

1.5.4 Domain-Specific Languages

Difficulties with inappropriate levels of abstraction can be addressed by the use of Domain-Specific Languages (DSL) for the target domain [81]. DSLs address the problem of inadequate semantic abstractions in GPLs by providing expressive semantic notations and constructs tailored towards a particular application domain. High-level, expressive constructs model domain tasks and entities that enable application developers to use language that is semantically intuitive of the domain [179]. This reduces the requirement for application developers to have significant domain knowledge. The resulting code is more concise, expressive and modular within the domain [81].

1.5.5 ALPH Model

To tackle the complexity produced by poor modularity and inappropriate levels of abstraction we combine two software engineering techniques, AOP and DSLs, in the ALPH model. By combining the benefits of AOP and DSLs, the model provides a means to develop applications within the pervasive healthcare domain at a high-level of abstraction and in a modular manner. Mobility, context-awareness and healthcare concerns are combined to create a set of pervasive healthcare concerns. These concerns emerge following an investigation into recurring crosscutting concerns in pervasive healthcare applications. ALPH models the pervasive healthcare domain by providing modular abstractions for each concern. To ensure a modular composition, each concern is implemented using an aspect-oriented language, AspectJ [147]. The functionality for each concern is encapsulated in appropriate modules i.e., aspects and classes. These modules are assembled within a library of modular pervasive healthcare functionality. It should be noted that the scope and depth of the modules are such that

the required behaviour is delivered to a sufficient degree to enable the investigation of the research question posed in this thesis.

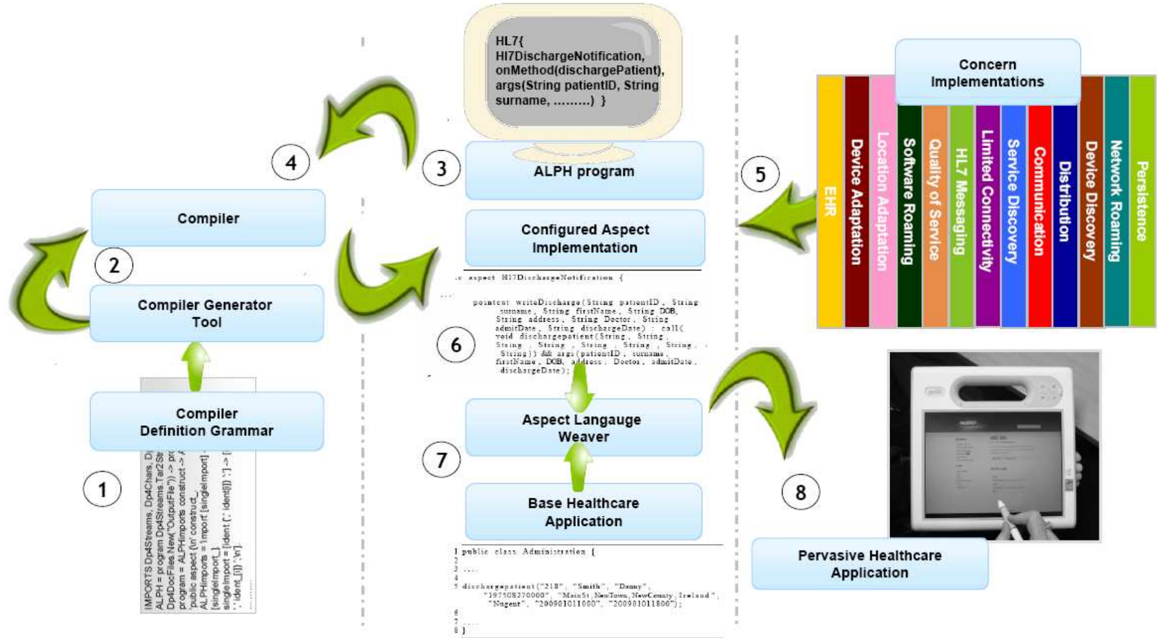

Fig. 1.1: ALPH Overview

To achieve the benefits of DSLs, the ALPH model makes the library of aspects available by the provision of domain-specific constructs in a programming language called ALPH (Aspect Language for Pervasive Healthcare) [187]. These constructs relate to each of the concerns implemented in the library of pervasive healthcare functionality. The syntax and semantics of the ALPH language have been formally specified as a grammar in EBNF notation, as illustrated in point 1 of figure 1.1. The ALPH compiler and compilation process is created from the defined grammar, as shown in point 2.

To include the pervasive healthcare concerns provided by ALPH into a base application a programmer must create an ALPH program defining the concerns to be included and application-specific implementation details associated with each concern i.e., where the concern behaviour should be applied. ALPH programs are constructed from the high-level, expressive, domain-specific constructs provided with application-specific details being provided

as parameters to the constructs, as shown in point 3 of figure 1.1. The ALPH programs are then parsed by the ALPH compiler, shown in point 4, and the generative compilation process triggers the inclusion of code from the library of modular concerns as seen in point 5. This code may be customised by the programmer according to application requirements specified through parameterisation. A configured aspect-oriented implementation is generated as output from the ALPH compilation process as shown in point 6. This is then "woven" into the base application by the aspect language weaver as in point 7. The base application may be an existing healthcare application undergoing upgrading to a pervasive environment or a new pervasive healthcare application. In either case, the base application is written in a base GPL language i.e., Java, and has no reference to or consideration for the domain-specific pervasive healthcare functionality. In the development process, developers are aware only of the base application and of the high-level ALPH constructs. This shields developers from the low-level implementation details of the domain-specific pervasive healthcare behaviour and reduces their requirement for domain knowledge. Subsequent to the ALPH compilation and weaving process, a complete, compiled and executable pervasive healthcare application is produced as shown in point 8 of figure 1.1.

An empirical evaluation was carried out to measure the effect of ALPH on application complexity with measurements determined through comparative analysis. Comparative studies took place between standard object-oriented and ALPH implementations of multiple applications. Applications were selected based on their requirement for any subset of the set of pervasive healthcare concerns addressed by ALPH. Five applications were selected including both internal case studies and independent third party codebases:

- DBay is a case study application based on an online auction website scenario. Its requirements were outlined independently by a research laboratory in the Swiss Federal Institute of Technology Lausanne [13] and both OO and ALPH implementations were produced within this work.

- MedHCP is also a case study application based on a scenario from the pervasive healthcare domain. The scenario was conceived by the Centre for Pervasive Healthcare [3] and staff at a collaborating hospital. Both OO and ALPH implementations were produced

as part of this work.

- Rococo [10] is a Bluetooth mobile phone software company based in Dublin, Ireland. They provided an OO mobile phone application codebase for comparative use in this work.
- HL7 Browser [6] is an open-source application for the viewing and manipulation of HL7 data messages. This was used as an OO comparative implementation.
- Healthwatcher [120] is testbed codebase made available through Lancaster University and was used as an OO implementation for comparison with ALPH.

Variations on application complexity were analysed against two dimensions; modularity and abstraction. The evaluation of high-level concepts such as modularity and abstraction is problematical due to the absence of explicit measurements techniques or metrics. The Goal-Question-Metric [43] approach outlines methods to derive quantitative level metrics from high-level conceptual goals. Using the GQM approach, metrics were selected for the measurement of both complexity producers, modularity and abstraction. Modularity can be measured by the variations in its indicators: maintainability, manageability and understandability [201]. Metrics for the measurement of modularity indicators are common AOP code metrics to measure coupling, cohesion, and independence [154] [157] [188]. Indicators of abstraction include conciseness and expressiveness [81] [70]. Metrics for the measurement of variations in abstraction include common code metrics, i.e., size and also required a metric to measure expressiveness.

Results show an overall beneficial effect of ALPH on application complexity. Modularity is increased with benefits evident in many indicators of modularity i.e., coupling, cohesion and independence. However, the use of AOP also produced a negative effect on indicators of modularity by introducing dependencies due to its dependence on syntactical details in the base code. Results show that ALPH has a beneficial effect on abstraction. Metrics show ALPH code to be more concise and more expressive than standard OO application code. The evaluation illustrates that the combination of the use of AOP and DSLs in ALPH

reduces elements of complexity in pervasive healthcare application code and improves the development process.

1.6 Contributions

The approach outlined in this work contributes to knowledge in the domain of pervasive healthcare by addressing the following issues:

- The investigation of pervasive healthcare functionality in applications has identified a set of crosscutting concerns in the pervasive healthcare domain. This set of comprehensive domain concerns reoccur in, and influence multiple parts of, pervasive healthcare applications. Their detailed functional purpose and crosscutting nature is documented to enable the elimination of negative crosscutting consequences i.e., complexity produced by poor modularity resulting in tight coupling, low cohesion and poor module independence.

- Current approaches to pervasive healthcare application development do not address the modularisation of pervasive healthcare functionality. Singular and subsets of concerns have been addressed regarding modularity, but the comprehensive set of concerns required in such applications remains difficult to encapsulate using existing approaches. The proposed approach provides a model for the modularisation of a comprehensive set of pervasive healthcare concerns. The approach provides a modular design and overall architecture for each pervasive healthcare concern. These concerns have been implemented and assembled in a library of modular components. This enables the clean encapsulation of crosscutting pervasive healthcare behaviour, resulting in increased modularity and hence, reduced complexity.

- Currently, the development of pervasive healthcare applications makes use of general purpose languages. These languages offer low-level all-purpose constructs to enable the programming of applications in a broad spectrum of domains. GPLs become problematic when implementing domain-specific functionality. Due to the low-level of abstraction, developers are required to have significant domain knowledge to produce the

required verbose, complex code. This code bears no semantic relevance to the pervasive healthcare domain. This approach establishes a DSL for the pervasive healthcare domain, thus raising the level of programming abstraction for the developer while reducing the requirement for domain expertise. This contributes knowledge to the area of software engineering in pervasive healthcare by realising a method of abstraction for pervasive healthcare application development. The applications developed using the approach are more expressive and semantically intuitive of the domain recognising the benefits that can be gained by the employment of a DSL in pervasive healthcare.

- Currently, it is difficult to evaluate certain characteristics often associated with DSLs. High-level concepts such as abstraction are difficult to assess as few means of quantifiable measurement are currently available. In this work we investigated areas such as empirical law, natural language processing and financial models to identify suitable techniques. Using a natural language technique, along with common code metrics, we establish appropriate metrics for DSL abstraction indicators.

These contributions are summarised in detail in section 7.1

1.7 Thesis Outline

The remainder of this thesis is presented as follows. Chapter 2 presents an overview of the state of the art in the area of pervasive healthcare application development and the related use of AOP and DSLs. Chapter 3 describes the design of the approach, detailing the identification and functionality of pervasive healthcare concerns addressed. Chapter 4 describes the implementation of the modular library of concerns as aspects and the creation of the domain-specific ALPH language. Chapter 5 presents an evaluation of the approach, including a description of the comparative analysis of multiple applications and the metrics applied in empirical experiments. Finally, Chapter 6 summarises the contributions of this thesis and discusses findings of the approach and directions for future work.

Chapter 2

Related Work

ALPH was devised to combat complexity in pervasive healthcare application code and in its development process by applying modularity and abstraction techniques to pervasive healthcare concerns. This chapter explores other programming support for pervasive healthcare application development, modularisation of pervasive healthcare concerns and abstraction in pervasive healthcare. A number of state of the art projects offer support for fractions of pervasive healthcare behaviour. This thesis examines the gap in support for a comprehensive set of concerns required by pervasive healthcare applications, in particular their modularisation and abstraction.

This chapter reviews relevant existing work related to reducing complexity in pervasive healthcare applications. We examine existing research in three major areas:

- Pervasive healthcare application development
- Modularisation of pervasive healthcare concerns
- Abstraction of pervasive healthcare concerns

The three areas above include all works relevant to the approach described in this thesis to reduce complexity in pervasive healthcare applications. These three areas combined in a single approach are required to reduce the complexity produced by difficulties in modularity and abstraction, as shown in figure 2.1. We examine the approach to the three areas in

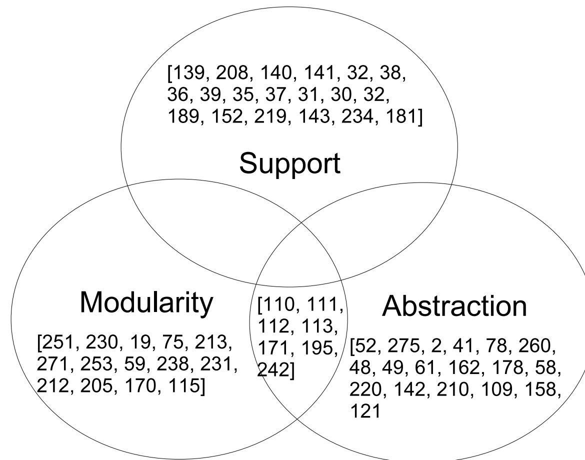

Fig. 2.1: Combination of areas to reduce complexity

existing work and, where appropriate, compare and contrast specific features of the ALPH approach to those systems under review. We conclude by summarising the extent to which each approach addresses these three areas for particular pervasive healthcare behaviour.

2.1 Pervasive Healthcare Application Development

Application development is aided by programming support in the form of APIs, middleware, frameworks, targeted domain languages, etc. Such support is provided with a goal of easing development from the application developer's point of view. In this section, we examine work on programming support for pervasive healthcare applications. We include work addressing the broad area of application development support and focus on the development requirements associated with incorporating HL7 and EHR functionality into healthcare applications.

2.1.1 Centre for Pervasive Healthcare

The majority of work on support for pervasive healthcare application development has taken place in the Centre for Pervasive Healthcare at the University of Aarhus, Denmark [3]. Many areas of research in pervasive healthcare application development have been investigated by the group. Projects have delivered work including requirements engineering in the do-

main, application architecture and design, investigation of computing paradigms in pervasive healthcare application development and the implementation of technologies.

Requirements for the domain have been investigated [139] [208] and the requirement for both context [32] [38] and mobility [36] [39] recognised. Use cases for a pervasive healthcare system have been described [140] [141] and challenges and technologies in the domain have been examined [35]. The requirements identified in these works are included in the domain analysis forming the basis for the ALPH model. While these are based on one particular research centre's requirements engineering, they are a useful, broad and ample set of requirements from the domain.

Designs have been outlined to aid in the development of pervasive healthcare applications. An architecture has been described for the development of a personal medical unit [33]. A pervasive healthcare middleware architecture vision is described in a white paper by Bardram and Christensen [37]. This vision incorporates research in the area with experiences from working with staff in a collaborating hospital [38].

This work has investigated supporting mobility and roaming in healthcare over heterogenous devices using an architecture based on JMS [31]. Activity based computing has also been investigated for the support of pervasive healthcare activities [30] and experiences in the deployment of context-aware technologies and applications in a hospital environment have been documented [32] [38]. The work at the Centre for Pervasive Healthcare contributes to many areas of the pervasive computing domain e.g., requirements, design, experiences and applications, adding valuable knowledge to the domain. However, it does not provide a concrete mechanism, available to developers, to aid in the development of pervasive healthcare applications.

2.1.2 HL7

The HL7 international standards institution promotes the standardisation of electronic healthcare information to facilitate its use in healthcare applications globally [189]. HL7 provides standards for the exchange, integration, sharing and retrieval of electronic health information [189]. The implementation of the HL7 messaging standard improves quality, efficiency and

effectiveness of healthcare delivery and shared medical information. Applications in the area of distributed healthcare require HL7 in systems that exchange and share data with internal and external systems [189].

Three approaches support the incorporation of HL7 standards in pervasive healthcare applications. Microsoft created a software factory for healthcare systems based on HL7 [215]. A model-driven development approach enables developers to configure a HL7 collaboration port application including messages, interactions, roles and flows. While the software factory provides valuable support and a useful tool for HL7, it does not provide support for the incorporation of HL7 functionality into existing applications, as collaboration ports are produced as separate applications. This requires existing applications to be refactored to use this functionality, inhibiting the modularity of domain-specific behaviour. Ko et al [152] have created a service-oriented architecture to ease HL7 healthcare application development. In the architecture, a web service layer sends and receives HL7 messages over SOAP. This approach provides an effective technique for easing the integration process of using HL7 functionality in pervasive healthcare applications. However, the complexity introduced by the functionality is not addressed. It requires significant effort on the developers part to create HL7 messages from a DOM-like (Document Object Model) C# class library and to send them via SOAP to a central Web Service for processing. CORBA and OLE have also been used to develop HL7 components for healthcare applications [219]. While this component based approach provides support in the development of pervasive healthcare applications and the distribution of interoperable healthcare information, it is a dated approach and does not address the complexity introduced by requiring the developer to implement the creation of HL7 objects and to make them known to the object broker or implement the remote access to a remote HL7 object.

2.1.3 Electronic Health Records

An Electronic Health Record (EHR) is defined as "digitally stored healthcare information about an individual's lifetime with the purpose of supporting continuity of care, education and research, and ensuring confidentiality at all times" [134]. The information contained in an

EHR may vary from notes, diagnostic images, laboratory tests, treatments, drugs prescribed, legal permissions to administrative details [92].

Supporting EHRs in pervasive healthcare applications is a difficult task [44]. Vendor-specific software packages are often used and are application-specific, so development in general is not aided. EHR systems encounter complexity on many levels e.g., in their need to be accessible to mobile users, their integration with knowledge bases such as terminology and clinical guidelines and their need for medico-legal support. Despite international guidelines, many hospitals and health facilities are yet to adopt EHRs. This intensifies the requirement for methods of incorporating EHR systems in new and existing healthcare applications.

EHRs are generally used in conjunction with interoperability standards such as HL7 for information and DICOM [183] for images [92]. The standards available for use with EHRs have been outlined to aid their incorporation into healthcare applications [44]. General rules for the architecture of an EHR system have been outlined and can be used in the development of a new EHR system [143] [234]. New EHR systems are difficult to implement due to the volume and heterogeneity of information held within them. The use of open-source software for the provision of EHRs has been identified as an alternative solution to some of the barriers encountered in the attempt to use electronic formats for patient information in pervasive healthcare applications [181]. This open-source software is based on standards and encompasses medico-legal support in its implementation [181]. However, it is a verbose, complicated software that provides little abstraction requiring great developmental effort from application developers.

2.2 Modularisation in Pervasive Healthcare

The use of modularisation as a software engineering approach to reducing complexity in applications was examined in section 1.5.1.1. Separately, it has been noted that "a great deal of literature extols the benefits of modularity, but very little is said about how to achieve it" [200]. To achieve the benefits of modularity, as outlined in section 1.4.1, each particular function, or concern, of a system must be encapsulated into modular components. A concern is a focused attention on some aspect of a system [83]. Each concern relates to one specific

topic that can be functional or non functional e.g., business logic such as authorisation or system based concerns such as distribution.

2.2.0.1 Separation of concerns

Dividing an application so that each module addresses and encapsulates one concern is the fundamental goal of modularisation, also known in software engineering as "the separation of concerns" [83]. This approach reduces the complexity of applications [265] [96]. Separating each concern enables it to be developed in isolation, understood without interference from other concerns and minimises the "impact of change" allowing systems to be maintained without requiring modifications in many other modules [252].

Dijkstra conceived the term, explaining that tackling various concerns simultaneously caused complexity and must be resolved by focusing attention upon and separating particular concerns individually [83]. Parnas had explored a similar notion under the name "modularity" [201]. The focus on separating concerns and advanced modularity techniques continued in subject-oriented programming [124] and multi-dimensional separation of concerns.

2.2.0.2 Crosscutting concerns

When many concerns are incorporated into one application, as is required in applications that need to fulfil a large number of both functional and non-functional requirements, they become difficult to develop [95]. Pervasive healthcare applications are one such group of applications. Many concerns have an effect on multiple parts of an entire system, causing concern code to be found "scattered" and "tangled" with the base functionality of the application [138]. These concerns are known as "crosscutting" [138]. These types of concerns disrupt the modularity desired in object-oriented systems [122] and introduce duplicated code [138]. Crosscutting concerns cannot be modularised using traditional object-oriented techniques, inhibiting the full separation of concerns.

2.2.1 Aspect-Oriented Programming

Following on from the work described in section 2.2.0.1, AOP emerged as a primary technique in the modularisation of crosscutting concerns. It is founded on the principle of separation of concerns [146] and its goal is to achieve a high degree of separation of crosscutting concerns by enabling developers to reason about, and implement them in separate modules [95]. Aspect-oriented applications are implemented using an AOP language such as AspectJ [147]. We use AspectJ terminology to explain the salient features of an aspect-language, namely aspects, join points, pointcuts and advice.

- Aspects are the concrete code realisation of a crosscutting concern as a module. As aspects are implemented in isolation from the rest of the base application, the application does not have any knowledge of or reference to the aspects [103]. Therefore, points of reference must be defined within the aspect to identify where in the base application the crosscutting behaviour should be executed.

- Join points are points in the execution of the base application at which aspect behaviour may occur. A join point is defined as "a location that is affected by a crosscutting concern" [199], "well-defined points in the execution of the program" [148] and "a location in the base program where a set of aspects can intercess their behaviors" [203].

- Pointcuts are predicates over the set of available join points. A pointcut defines a join point or set of join points along with information that might be required by the pointcut e.g., variable values. Each aspect may have collection of pointcuts.

- Advice is the implementation of concern behaviour that will be executed when a join point that matches a description in a pointcut is reached.

To combine the AOP implementation with the base application, the aspect implementation is "woven", or merged, with the base application to produce an executable application. When a pointcut predicate is matched during execution, advice behaviour is executed seamlessly.

2.2.2 AOP in Pervasive Healthcare

The use of AOP to modularise concerns relating to pervasive healthcare has not been examined outside the scope of the ALPH project [189]. However, the pervasive healthcare domain encompasses pervasive computing concerns that have been tackled by AOP techniques in order to achieve increased modularity. We examine related works that have applied AOP to any concern or subset of concerns in the pervasive healthcare domain. The work described does not address abstraction levels and tackles modularisation in isolation.

2.2.3 Context Adaptation

Context-awareness has been addressed in several ways using AOP. Context-aware aspects [251] apply the notion of context-awareness to aspects themselves using the Reflex AOP kernel [250]. Contexts are defined using an annotation framework and a context definition framework, enabling aspects themselves to become context-aware rather than providing context-awareness behaviour. Context-awareness in virtual environments has also been implemented using AOP [230]. In this work, an aspect layer was proposed to support customisation according to context [230]. Personalisation in web servers has been modularised using AOP [19]. The Apache Tomcat server was extended using AspectJ to adapt JSP pages according to contextual information. A set of adaptability aspects have been created using AOP and architectural patterns [75]. The developer provides adaptive behaviour by specifying an interface whose methods will be invoked by the aspects. A context manager triggers these adaptive actions. Displays can be adapted according to contextual information using a technique that encapsulates three facets of adaptation as aspects i.e., display management, content management and display adaptation [213]. The Dynamic Object Composition Language (DynOCoLa) framework [271] offers another method of dynamic customisation of applications based on context information. A healthcare system is used as an example application and role-based access control was developed using the DynOCoLa framework. This object-oriented framework has aspect-oriented language extensions including an "aspect" object and a "typemarked" method where behaviour changes are applied. Context adaptation behaviour defined in the aspects is woven on compilation and executed when appropriate

at runtime. These approaches provide effective modularisation of context adaptation using AOP. Some do not offer generic adaptation akin to that provided in the ALPH model as they are too specific [19] or have a different notion of modularising context [251]. DynOCola [271] and the adaptability aspects [213] offer modularity in adaptation similarly to the ALPH model.

2.2.4 Distribution

Distribution concerns have been modularised using AOP in many attempts to ease distributed application development. Most approaches focus on Java-based distribution solutions. Distributed object allocations [253] for Java have been modularised, aspects to distribute existing applications using RMI created [59] and basic remote access to services using RMI modularised using AOP [238]. Distribution using CORBA has also been separated using AOP [231]. These approaches do not address any behaviour beyond distribution and abstractions are not provided, requiring GPL level implementation from the application developer. The modularisation using AOP applied to distribution in these approaches addresses the complexity created by poorly modularised code. This approach is reused in the ALPH model as distribution is separated in a similar manner apart from functional detail differences.

2.2.5 Persistence

Persistence has been successfully modularised using AOP in several existing works. In work by Rashid and Chitchyan [212], persistence is successfully modularised using AOP producing highly reusable persistence aspects for database access and SQL translation. Persistence is modularised, along with awareness, authentication and multiple views, using AOP as part of work that supports collaborative users in virtual environments [205]. A JDBC-based implementation by Soares et al provides persistence in a similar fashion to ALPH [238]. The modularisation reduced code size significantly, indicating the removal of duplicated crosscutting code. The ALPH model reuses the approach of applying AOP to persistence described in these works.

2.2.6 Quality of Service

Quality of Service (QoS) requirements such as load balancing, efficient resource usage and fault tolerance have been modularised using AOP [231]. The concerns are defined and generated in an aspect-oriented language by a CORBA-LC code generator [231] that uses information from CORBA Interface Repository (IR) and definitions in components' XML files. This approach separates the QoS concern, but requires significant effort from the application developer to design interfaces and establish the set of interfaces and events in each component. This approach considers only QoS functionality and does not support any further pervasive behaviour. The ALPH model includes QoS behaviour in a similar manner using AOP in its modularisation of pervasive healthcare concerns, but abstracts the developer from implementation details.

2.2.7 Mobility

Mobility has been described as an aspect [170] in the AspectM framework. Code mobility, or software roaming, is implemented as a software agent using AspectJ. An actor-based approach using AOP can also be used to develop time-dependent distributed applications [115]. Actors are encapsulated units of data and finite state machines are used to represent their behaviour providing the modular inclusion of software roaming in pervasive applications [115]. The ALPH model addresses the modularisation of mobility in a more detailed manner than existing approaches. Software roaming is provided using a software agent technique, similar to that used in the AspectM framework. However, the ALPH model also addresses the geographic roaming of a user, limited connectivity in a mobile environment, discovery of mobile devices and services and the communication required to support mobility.

2.3 Abstraction in Pervasive Healthcare

As described in section 1.5.1.2, abstraction is commonly used as a technique to ease the development process [81] and reduce complexity [176]. DSLs provide an efficient means of programming while offering application developers the greatest level of abstraction possible

for programming in a particular domain. We examine DSLs and their benefits in detail before examining the application of DSLs to attain abstraction in the domain of pervasive healthcare.

2.3.1 Domain-Specific Languages

DSLs are a valuable approach to facilitating efficient software engineering [118]. A DSL defines an engineering methodology for a particular application domain [256]. 'A DSL is defined as "a programming language that offers, through appropriate notations and abstractions, expressive power focused on and usually restricted to a particular domain" [81].

The use of GPLs for application development in a particular domain provide suboptimal solutions [81]. A DSL is tailored to a problem domain, therefore focusing its syntax and avoiding overly general constructs that are needed to support general-purpose programming [273].

2.3.1.1 DSL Benefits

Adopting a DSL approach in programming for a particular domain offers a number of benefits [81] as described below.

Conciseness

Conciseness is a well documented advantage in the use of DSLs [273] [81] [256]. Programs written using DSLs are more concise and more self-documenting than GPL programs [81]. DSLs are found to be roughly a factor of ten times smaller than equivalent programs written in a GPL [256]. Developers may specify as little as 2% of the code that would be needed to program the equivalent functionality in a conventional programming language [273]. This results in significantly shortened development and testing times [256], and related advantages of comprehensibility, ease of writing, fewer errors and greater productivity [273].

Abstraction

The use of DSLs allow solutions to be expressed at the level of abstraction of the problem domain [81]. The use of abstraction in DSLs to reduce complexity is a common motivation [256]. The high-level constructs provided by the DSL abstract over low-level application details [274]

leading to reduced complexity and shorter development times [256]. When programming at a high-level of abstraction using a DSL, the reduction in domain knowledge required in domain application development is the "single greatest advantage provided by the use of DSLs" [273]. The reduction in the requirement for comprehensive domain knowledge means that the team of domain experts and programmers previously required to develop applications in the target domain can be reduced [179]. As the language incorporates abstractions for domain knowledge, the programmer requires little detailed knowledge of the underlying domain-specific functionality [118]. This opens up application development in the domain to a larger group of software developers, with less domain knowledge and/or programming expertise, compared to using GPLs [179].

Expressiveness

DSLs offer substantial gains in expressiveness over the use of GPLs in their domain of application [179] [81]. They trade generality for expressiveness, offering notations and constructs tailored towards a particular application domain [179]. These notations capture the domain meaning, therefore, the expressiveness of DSL semantics are accepted as being more familiar to the developer [118] [274] [256]. The increase in expressiveness, or "focused expressive power" [81] is a key distinguishing factor in DSL programs, achieving greater meaning to the developer [118]. More meaningful code represents domain knowledge, making it easier to understand and develop.

Reuse

Another documented benefit of DSLs is their enabling of reuse [156] [179] [153] [256]. DSL constructs represent domain knowledge and enable the preservation and reuse of this knowledge [81]. Language grammars, source code, software designs, application generators and domain abstractions can all be reused [179], along with domain expertise [256] and the resulting programs themselves [81]. As DSLs produce components specific to a target domain, they are likely to be suitable for reuse in the domain to a greater extent than GPL components.

Productivity

DSLs improve development efficiency in domain applications as developer productivity is

increased resulting in shorter development times [274]. Developer productivity is increased using DSLs both during and beyond the coding phase of software development [118] [79]. Increased productivity is a well documented benefit of using DSLs [81] as they can achieve an order of magnitude of productivity over GPLs [274]. Using DSLs is more efficient than GPLs for the developer who must solve the domain problem programmatically [273]. Application development is both quicker and easier using a DSL for a particular domain [118].

Software Engineering "Ilities"

Software quality is defined as the degree to which software possesses a desired combination of software engineering characteristics, known as "ilities", such as maintainability, testability, reusability, reliability, comprehensibility[12]. These "ilities' are positively affected by the use of DSLs in domain application development.

Due to the intuitive semantic and syntactical nature of DSLs they are more readable to application developers [256] and resulting DSL programs are easier to understand and to maintain [118]. DSLs increase reliability [81][256], maintainability[261] and portability [128]. They also enable validation and optimisation at the domain level [81] [55]. They can improve testability [81] and increase safety and accessibility [256]. DSLs achieve gains in overall ease of use compared to GPLs [179] and reduce complexity both in the development of applications and in the resulting applications [256] [81].

2.3.1.2 Limitations

DSLs have disadvantages that limit their applicability. These limitations generally emerge from the process of creating a new language i.e., high costs of designing, implementing and maintaining a DSL, difficulties in scoping the DSL and the difficulty in defining appropriate domain constructs [81] [156]. Users of new DSLs also require education on how to use the language that is not neccessary when using familiar GPLs [81].

2.3.2 DSLs in Pervasive Healthcare

The application of DSLs to attain abstraction in the domain of pervasive healthcare has not been addressed by any existing work. As no research exists in this specific area, we

examine projects that have independently applied abstraction using DSLs to any concern that is encompassed within the domain of pervasive healthcare.

As early as 1979, there were recommendations for the development of "high-level programming languages for medical computing" by government agencies and academics [197]. Although the application of DSLs in healthcare was identified as a beneficial and advantageous exercise [197], very little work has been done in the area of DSLs in healthcare. We examine three approaches that apply abstraction to the area of programming for healthcare applications.

2.3.2.1 MUMPS

MUMPS (Massachusetts General Hospital Utility Multi-Programming System) [52], was a domain-specific language developed by Pappalardo and Barnett in the 1960's with the original target application domain of healthcare. However, the concepts abstracted were widely applicable as they addressed the development of database applications. Backend datastore integration was made transparent, abstracting the developer from persistence implementation. MUMPS was used to develop financial, healthcare and telecommunications applications. Its constructs relate solely to operations based on data management e.g., SET, WRITE, DO, NEW, QUIT, and its major competitor is SQL used in conjunction with general purpose programming languages or as part of an enterprise solution. Despite its creation as a healthcare language, it provides no behaviour specific to healthcare. While it is still in use in some financial markets, it has become almost obsolete in its application in new pervasive healthcare applications due to its limited support for pervasive healthcare functionality. In contrast, ALPH provides constructs and functionality to support behaviour specific to the domain of pervasive healthcare. This includes the support for the distribution and storage of healthcare information and the incorporation of mobility and context-awareness behaviour in applications.

2.3.2.2 Reformatting of HL7 Messages

HL7 is the international standard that enables healthcare information interoperability. It outlines a standard for data format but does not provide any language support for its communication or exchange. The HL7 standard has been constantly updated, which has given rise to version compatibility problems. Williams [275] developed a language to generate programs to automatically reformat HL7 messages to a particular version as required by an application. The language takes specifications of what parts of the HL7 message to reformat and outputs a HL7 message in the required format. While this is a useful tool in version compatibility, it assumes the existence of HL7 support in a system and does not address any functionality in pervasive healthcare applications other than version formatting. ALPH supports creation, parsing and segmentation of HL7 messages, in addition to their communication and exchange.

2.3.2.3 Apache Camel

Apache Camel [2] is an open source integration framework based on known enterprise integration patterns. The API provided includes a HL7 component that supports sending and receiving of HL7 messages, along with their formatting. The provision of constructs to support such behaviour enables HL7 functionality in pervasive healthcare applications. Camel, like ALPH, makes use of HAPI (HL7 application programming interface) [4] to support all possible HL7 functionality. ALPH however, provides a DSL rather than an extended API as provided by Camel, achieving the DSL benefits outlined in section 2.3.1.1.

2.3.3 DSLs in Pervasive Computing

DSLs have been applied to many areas within the domain of pervasive computing. ALPH supports mobility and context-awareness concerns particularly to support the deployment of healthcare applications in pervasive environments. We examine existing work that abstract mobility and context-awareness concerns within pervasive computing using DSLs.

2.3.3.1 YABS

YABS [41] provides a DSL for defining entity behaviour and coordination in pervasive computing applications. It bases its composition of components on the stigmergy model from nature. YABS focuses on context-awareness and defines the proximity or area around an entity and behaviours that can be performed by the entity. These are mapped according to specifications so that the entity adapts according to contextual information. This approach is not easily applicable to existing applications as it requires developers to define the sensors and actuators in the base application using Java before defining entity behaviour using YABS. The ALPH model does not require modification to existing base applications to include domain-specific functionality. YABS provides a DSL for context adaptation, but does not address underlying mobility concerns such as distribution. ALPH deals with mobility and healthcare concerns in addition to providing context handing functionality.

2.3.3.2 AmbientTalk

AmbientTalk [78] is a DSL for "ambient-oriented" programming. This includes support for service discovery, distributed communication and composition in mobile ad hoc networks. An underlying actor based, event-driven concurrency model is adopted to support the organisation of mobile nodes in an inherently unreliable mobile environment. An event-driven model includes actors (event loops), events (messages), event notifications (message sends) and event handlers (object methods). Actors are transparently distributed and use asynchronous messaging to communicate messages. Limited connectivity is addressed in two ways; during network disconnection, messages are stored for unavailable entities and remote references are "leased" enabling the identification of persistent failures upon unrenewable leases. AmbientTalk addresses many concerns from the domain of pervasive healthcare including limited connectivity, distributed communication, device and service discovery [260]. However, context-awareness and healthcare specific functionality are not addressed. AmbientTalk is a standalone DSL, meaning that the entire application is implemented using the language. The ALPH model requires the ability to incorporate new behaviour in existing applications, as well as new applications, making the AmbientTalk approach unsuitable.

2.3.3.3 Indus

Indus [49] [48] provides abstractions in a DSL for programming with software agents in pervasive computing. Constructs in Indus enable the definition and coordination of concurrently executing processes (agents) and components. Concerns related to the ALPH approach addressed by Indus are service discovery, distribution, routing, roaming, communication and persistence. While Indus includes a construct called “Context”, this deals with an entity’s application state alone e.g., its policies for transactionality, and does not provide the ability to interact with contextual information from the environment. Applications are built using the Java-based Indus language to develop the entire application. This approach is not applicable to existing languages, requiring full reimplementation using Indus. The ALPH model addresses similar concerns to Indus, but can include domain-specific behaviour in both new and existing applications without the need for refactoring.

2.3.3.4 PLUE

PLUE (Programming Language for Ubiquitous Environments)[161][162] is a DSL based on event-condition-action rules and finite state automata based interactive responses for the dynamic adaptation of pervasive computing applications. Events, conditions and states are outlined using PLUE and transitions from state to state take place when an event satisfying the condition occurs. PLUE acts as a pre-processor to a finite state machine, adding a level of abstraction to the definition of the required states and rules. This approach is markedly different from ALPH. The ALPH model abstracts developers from low-level implementation details by providing pre-implemented domain-specific behaviour via DSL constructs. In PLUE, the developer is required to implement all required behaviour as no underlying pervasive computing functionality is provided by the constructs of the PLUE language. Its implementation also requires the refactoring of base applications to use PLUE to implement the entire application, as opposed to using the ALPH model to introduce domain-specific behaviour where required.

2.3.3.5 Pantaxou

Pantaxou [178] is an event-oriented language that addresses the coordination of networked entities in a pervasive computing environment. Pantaxou provides two services, one for creating an environment description and one for programming coordination services. Entities are adapted based on the states and interactions described using the environment description language. Coordination is addressed by defining which components or services interact with each other using the coordination logic language. Discovery and distribution are concerns addressed by both ALPH and Pantaxou, although Pantaxou uses an event-driven approach. Using Pantaxou, applications are developed using two languages, an environment description language and a coordination logic language. This requires the full refactoring of existing applications to use Pantaxou, requiring the developer to build a networked environment and to develop the application to be deployed in that environment. This approach is not applicable to the requirements of the ALPH model due to the refactoring required in existing applications.

2.3.3.6 DiaSpec

DiaSpec [58] includes a language to define device descriptions and an architecture description modelling language to describe application architectures. It is a continuation of work described in Pantaxou. The same event-based approach is taken, where a taxonomy of devices' implementations describe the types of data gathered from the environment and the actions supported by the devices. The architecture is modelled in terms of a set of contexts that detect relevant devices and trigger appropriate actions. DiaSpec focuses on supporting context-awareness but also supports distribution and device discovery. It provides modelling functionality along with an event-based context language. ALPH addresses the implementation stage of application development and is not concerned with modelling. Pervasive healthcare functionality i.e., mobility, healthcare and context-awareness requires support for a larger set of concerns than addressed by DiaSpec. Also, the same restrictions apply as when using Pantaxou, i.e., applications must be developed from scratch using DiaSpec making its use with existing applications infeasible.

2.3.3.7 Scooby

Scooby [220] provides a service description language for pervasive computing environments. Services are described and composed using high-level constructs. While Scooby offers abstraction and expressivness for detailed service-oriented applications, its applicability is limited to service descriptions and does not offer support for any other pervasive healthcare functionality. It it also limited in its applicability as it is based on the use of Elvin [1], a distributed event routing service. The version of Elvin used is limited and unsuitable for largescale, distributed systems. The ALPH model makes use of scalable technologies in its implementation e.g., RMI.

2.3.3.8 PerIDL

PerIDL [142] is an interface definition language (IDL) that describes services for pervasive computing applications. Services are described by their properties and supported actions, or commands. Event notifications are published by event sources prompting commands to be carried out on the appropriate services. PerIDL offers excellent DSL support for defining services in a similar manner to Pantaxou and DiaSpec. Applications using PerIDL for service discovery extend abstract classes generated by PerIDL. These explicit references to domain-specific code do not support the goal of modularity in the ALPH model.

2.3.3.9 Olympus

Olympus [210] is a programming framework built on the Gaia middleware that allows the specification of entities and operations in pervasive environments. While considered a middleware, Olympus provides a layer of abstraction above most middlewares as a set of classes in an API. It provides high-level programming interfaces to define these components in active spaces. However, the framework remains general purpose and provides abstraction only for the specification of active entities. These entities are then referenced from the developer's base application. This approach does not address the modularity of the domain-specific code, leaving complexity resulting from poor modularity in applications.

[1] http://elvin.org/

2.3.3.10 Context Adaptation

A context-based programming DSL for dynamic adaptations by Fritsch et al is for use in context-aware applications [109]. Context graphs are described using a high-level language and a condition-event-action model is adopted to trigger transitions from one state to another based on contextual events. Adaptation to contextual information is well supported in this work, although no other pervasive healthcare functionality is addressed. It is also applicable only to applications using context graphs, representations of context for finite state machines. Other projects that abstract contextual adaptation include an XML based language that allows adaptation strategies to be defined and system services, such as cache size and underlying communication services, adapted [158]. In addition, a generic adaptation language for component based applications also enables the adaptation of system services [121]. This language adapts the behaviour of existing middlewares to system services and it is not applicable to applications that use unsupported middleware.

2.4 Domain-Specific Aspect Languages

While aspect languages such as AspectJ or AspectC++ [245] enable the modularisation of crosscutting concerns, they are still GPLs and lack support for domain-specific features of the application problem domain [243]. A domain-specific aspect language (DSAL) describes specific crosscutting concerns and provides language constructs tailored to the particular representation of such concerns [214] [233]. This enables the modularisation benefits of AOP to be coupled with those of DSLs which helps to encapsulate the characteristics unique to a particular problem domain [243]. DSALs have addressed many crosscutting concerns including advanced transactions [96], virtual machines [258] and embedded real time systems [243].

We examine existing work on DSALs that address any concern required by pervasive healthcare applications. No single approach supports a comprehensive set of pervasive healthcare concerns in a DSAL, although several languages exist that support a single concern or subset of concerns.

2.4.0.11 AOPAmI

The AOPAmI [111] platform applies AOP in ambient intelligent application development. This approach evolved from the Dynamic Aspect-Oriented Platform (DAOP) platform that supports component-based development. DAOP is concerned with the architecture of component-based applications, and dynamic reconfiguration of these components at runtime. It performs dynamic weaving of components according to architectural definition, making use of aspect-oriented techniques.

As part of the early DAOP work, an ADL was created for architecture definition in the DAOP platform. This XML architecture language, known as DAOP-ADL [206] defines components, their possible implementation classes, inputs and outputs, composition rules and dependencies and aspect evaluation rules. While the language provides a method of defining application information outside the use of a traditional GPL, it does not provide DSL constructs and is targeted at the modelling and design phase of development. DAOP-ADL does not consider the majority of pervasive healthcare concerns.

The DAOP platform and ADL evolved into DAOPAmI [112] to support ambient intelligent environments. Still XML based, the language was extended to define device types as profiles, system aspects such as discovery and strategy rules for strategies to apply when events happen in the environment. This version of the platform supports many pervasive healthcare concerns i.e., discovery, limited connectivity, distributed communication and device context handling. While these are supported, the creators themselves acknowledge the lack of a "complete vocabulary for each concern" in the domain and question the limitations of XML as a means of providing a higher-level language [158].

The work progressed to be known as AOPAmI [111]. The platform, now referred to as a middleware, continues to address the same concerns, describing the weaving of components and aspects at the architectural level [113] using the previously described XML based ADL. Most recently, a feature model is described for a product line architecture building on the existing AOPAmI platform again using the existing XML ADL for architecture descriptions [110].

2.4.0.12 Distributed Definition Language

The Distributed Definition Language (DDL) is a simple high-level DSAL [242]. This language provides descriptions of remote classes and methods using a specified Java distribution technique e.g., RMI or Java sockets. The language compiler, RemoteJ, generates the scaffolding distribution code and manipulates the existing application bytecode directly. This approach offers both modularisation and abstraction in the same way as the ALPH approach, though addressing the single concern of distribution. They differ through functional properties and in the design and scope of the language provided.

2.4.0.13 D Language Framework

The D Language Framework [171] addresses the concern of distribution. Within D, COOL is an aspect language for synchronisation, RIDL is an aspect language for the definition of remote interfaces and JCore is an object-oriented language to express the base functionality of the system. In application development using D, the base application is programmed in JCore, a subset of Java. COOL is then used to outline the coordination of threads i.e., the synchronisation of objects and methods as required. RIDL is used to express remote access strategies. D is implemented as a pre-processor to Java, using an aspect weaver to create the resulting Java application. The D Framework provides both modularisation and abstraction for the concern of distribution though it does not address any other pervasive healthcare concerns. The COOL and RIDL languages provide more distribution behaviour than is available in the current implementation of the ALPH model, i.e., synchronisaion support. Base applications are required to be in a specified using the JCore language meaning that existing base applications may require slight refactoring to conform to JCore requirements.

2.4.0.14 AWED

AWED [195] is a comprehensive DSAL for distribution. Aspect-oriented support for distribution is provided through extensions to the JAsCo [249] framework. Three features can be described using domain-specific constructs; remote pointcuts, distributed advice and distributed aspects. AWED is concerned with the explicit distribution in the aspect language itself,

rather than the application of distribution code using AOP as provided in the ALPH model. Like D and DDL, distribution behaviour is both well modularised and abstracted in AWED but no other pervasive healthcare concerns are addressed.

2.5 Middleware

Middleware is computer software used to glue together or mediate between separate software components or applications. Various middleware models, APIs, frameworks and architectures have emerged to assist in the development of pervasive applications. We include middleware for completeness. No existing middleware supports the development of applications in the pervasive healthcare domain.

Some examples of middleware that support various aspects of pervasive applications include: middleware for static and dynamic adaptation [222] [116] (TAO, ACE, Orbix, OpenORB, MetaSockets, Isis, Horus, , Electra, FRIENDS, EmbeddedJava, ZEN, OpenCorba, DynamicTAO, FelxiNet, Globe, LIME, TSpaces, JavaSpaces, Nexus, Hermes, Bayou, Jini) sensor networks [127] (Cougar, TinyDB, TinyLIME, SINA, MiLAN), service composition [135] (MySIM, PERSE, SeSCo, Broker, SeGSeC, WebDG, SAHARA) , context-awareness [150] [25] [14] (Aura, CARMEN, CARISMA, Cooltown, Gaia, Middlewhere, CORTEX, MobiPADS, SOCAM, CASS, CoBrA, ContextToolkit, Hydrogen, CMF, JCAF, ContextFabric, Context Shadow, HyCon, WildCAT, Solar), service discovery [136] (Jini, JMatos, Salutation, Konark) and components (PCOM). These middleware and frameworks provide support for subsets of pervasive computing concerns. However, they all require references in the base application to their components. The ALPH model provides modular pervasive computing behaviour that requires no modification of the base code. No middleware exists to provide healthcare specific programming support. The ALPH model includes support for the distribution and storage of healthcare information in addition to pervasive computing concerns.

Aspect-oriented middleware supports common services that are non-functional and cross-cutting in nature, such as persistence, distribution, security, and transactions [173]. Many object-oriented middleware were evolved to aspect-oriented versions to support the definition of these services outside component implementation [173].

Some examples of AOP middleware are: AO4BPEL supporting web services, AspectJ2EE supporting persistence, transaction management, security, load balancing and all J2EE services, CAM/DAOP supporting distribution, security, persistence and communication, JAC supporting distribution, persistence, transactions, broadcasting, authentication, access control, consistency and load-balancing, JBoss AOP supporting distribution, caching, communication, transactions and security, Lasagne supporting dynamic context adaptation, PRISMA supporting distribution and communication, Spring AOP supporting transactions, persistence and distribution, WSML/JAsCo supporting web services. AOP middleware provides the separation of domain-specific behaviour in addition to support for distributed applications. No AOP middleware addresses healthcare specific concerns. The ALPH model could have used components from existing AOP middleware, but would have had to combine several frameworks, and extend them significantly to provide support for the modularisation of pervasive healthcare concerns. The provision of DSL constructs on top of these middleware may have also been a more complicated process than building the model to the targeted domain from scratch.

While middleware assists in the development of pervasive applications, there is no specific focus on the reduction of complexity as in the work described in this thesis. In fact, middleware itself is becoming so progressively complex that it threatens to undermine its goal of simplifying the construction of distributed systems [69]. The huge volume of middleware standards and technologies also contribute to this complexity [280]. While a level of abstraction can be achieved using an API or framework, it cannot achieve the expressiveness, conciseness and reduction in domain knowledge required by developers as can be achieved using a DSL. The ALPH model combines AOP, the provision of pervasive healthcare functionality and DSLs to achieve the best separation of concerns, a high-level of abstraction, and a reduced developmental effort.

2.6 Summary

This chapter has presented related work in pervasive healthcare application development. In particular, the chapter has examined state of the art projects in programming support

for pervasive healthcare application development and support for the modularisation and abstraction of concerns within the domain.

Table 2.1 summaries the approaches and projects examined. The extent to which these approaches support pervasive healthcare concerns is illustrated. HL7 healthcare information formatting is supported by a number of projects. However, each of these approaches addresses HL7 in isolation and does not address the underlying mobility concerns required for the distribution of the applications exchanging the information. There is little programmatic support for the inclusion of EHRs, inhibiting the organisation of healthcare information. State of the art context-awareness frameworks support the adaptation of applications in response to changes in contextual information from the pervasive environment in which pervasive healthcare applications are deployed. In most approaches, context-awareness is addressed in isolation and they provide limited, or no, support for mobility concerns. Mobility concerns are supported by projects focusing on the distribution of applications, coordination of entities and communication between entities. Concerns including service discovery, quality of service, limited connectivity and software roaming are addressed by various projects with many supporting a collection of mobility concerns. Again, these projects focus on only one aspect of pervasive healthcare applications. The use of AOP in examined approaches has modularised a number of pervasive healthcare concerns. The same can be said for the use of DSLs enabling the abstraction of many concerns. However, existing approaches generally address particular concerns in isolation, with only four approaches addressing three or more concerns.

In summary, there are many approaches to support the incorporation of pervasive healthcare concerns in applications. However, while each of these projects provide some level of support for pervasive healthcare application development, the support for a comprehensive set of concerns is limited and no approach fully supports all requirements. From this we conclude that no single approach offers full support for a set of pervasive healthcare concerns. These projects therefore cannot, without significant extension, be used to support modularity and abstraction in pervasive healthcare application development.

The next chapter describes the design of a novel approach to the development of pervasive

healthcare applications. This approach includes support for a comprehensive set of concerns from the domain. In Chapter 3, the methodology used in the creation of the ALPH model is explained. The rest of the chapter focuses on the domain analysis performed to identify concerns for inclusion in the ALPH model.

	HL7	EHR	Context	Persistence	Distribution	Software roaming	Network roaming	Service discovery	Device discovery	Connectivity	Location	QoS	Communication
Microsoft Software Factory	•												
SOA HL7	•							•					
CORBA HL7	•				•								
Context-aware Aspects			•△										
Virtual Environments			•△										
Personalisation as an Aspect			•△										
Adaptability Aspects			•△										
Context-aware Displays			•△										
DynOCoLa			•△										
Distributed Object Allocation					•△								
Distribute existing applications					•△								
CORBA AOP					•△								
Persistence as an Aspect				•△									
Persistence and Distribution				•△	•△								
AspectM						•△							
Time-dependent Applications					•△	•△							
CORBA-LC AOP												•△	
MUMPS				•□									
Reformatting HL7 Messages	•□												
Apache Camel	•□												
YABS			•□		•□								
AmbientTalk					•□			•□	•□	•□			•□
Indus				•□	•□	•□		•□					•□
PLUE			•□										
Pantaxou					•□			•□					
DiaSpec			•□		•□				•□				
Scooby								•□					
Olympus													
AOPAmI								•△	•△	•△			•△
DDL					•△□								
D Language Framework					•△□								

• = Supported, △ = Modularised, □ = Abstracted

Table 2.1: Summary of state of the art approaches to pervasive healthcare concerns

Chapter 3

Domain Analysis

The analysis of the state of the art approaches to pervasive healthcare application development in the previous chapter shows that they are limited in their support for modularising pervasive healthcare concerns and in providing high-level abstractions to application developers in the pervasive healthcare domain. The hypothesis of this thesis is that modularisation and abstraction are required for crosscutting concerns in pervasive healthcare applications to reduce the complexity of the development process and of application code.

This chapter describes the design of ALPH, a novel approach to the development of pervasive healthcare applications. The ALPH model was devised using the Cleaveland methodology [68] for its design, implementation and use. This chapter describes the phases in the methodology used in the creation of the ALPH model. The chapter then focuses on the analysis phase of the model, describing the steps carried out in domain analysis.

3.1 Overview of Model

The ALPH model provides support for pervasive healthcare application development with the goal of reducing complexity. The model provides support to developers in three ways; with means to include pre-written pervasive healthcare functionality, with modularisation of pervasive healthcare concerns and with a means to abstract over domain implementation details. Combined, these techniques address the complexity that is manifested in current

pervasive healthcare application development, as outlined in section 1.4 i.e., difficulties with modularity and inappropriate levels of abstraction. The ALPH model supports application development in a modular and abstract way by using AOP and DSLs. This approach requires the implementation of a library of crosscutting concerns and formation of a DSL that makes this functionality available to the developer in an intuitive manner. The development of such a model requires a methodical development process. The following section describes the methodology adopted in the formation of the ALPH model.

3.2 Methodology

The ALPH model's goal is to ease application development in the target domain of pervasive healthcare by providing modular implementations and programming abstractions for domain-specific functionality. We follow the established design methodology based on work by Cleaveland [68] in the development of the ALPH model. This model defines the steps typically involved in the development of a domain-specific language [81] [45]. The methodology involves the following steps:

1. Recognise domain: identify the problem domain of interest.
2. Define domain boundaries: gather all the relevant knowledge in this domain.
3. Define an underlying model: cluster this knowledge in a handful of semantic notions and operations on them.
4. Construct a library that implements the semantic notions.
5. Design a DSL that concisely describes applications in the domain.
6. Design and implement a compiler that translates DSL programs to a sequence of library calls.
7. Write DSL programs for all desired applications and compile them.

The steps in the methodology can be broadly considered as analysis (steps 1-3), design and implementation (steps 4-6) and use (step 7) [81]. This methodology systematically structures

the development of a model, from early domain analysis to later implementation stages. This chapter examines the analysis phase, with Chapters 4, 5 and 6 covering the remaining phases in the methodology.

3.2.1 ALPH Design

Adopting the described development methodology, ALPH follows the stages of analysis and design, implementation and use. Figure 3.1 shows the correlation of steps in the methodology to phases of development and also to the chapters in this thesis that describe the details of each step.

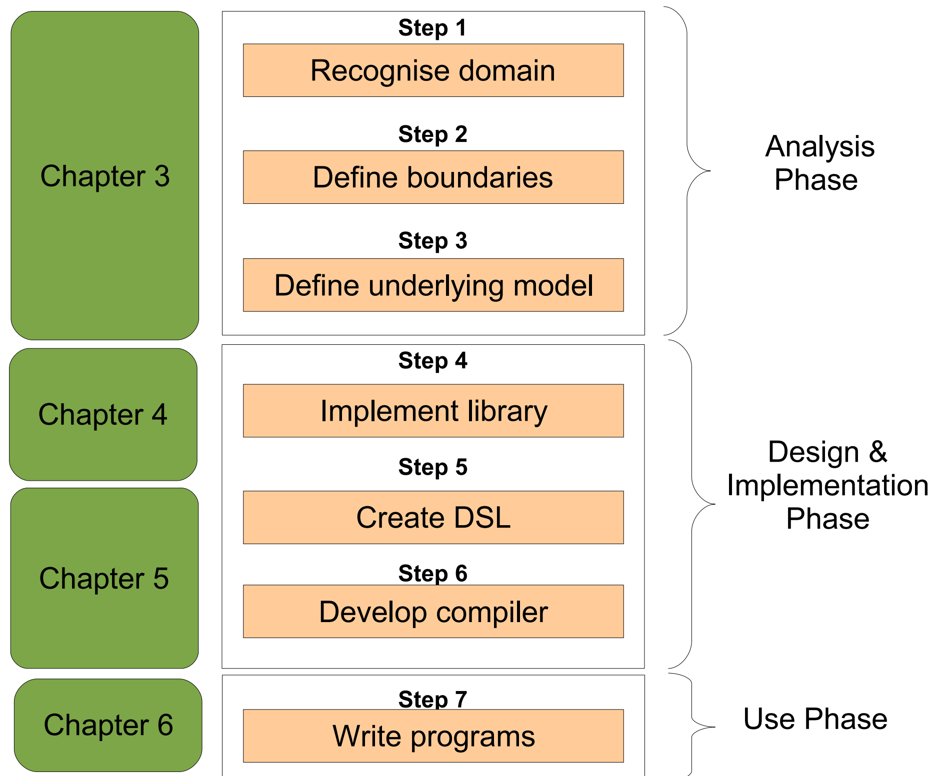

Fig. 3.1: Development Methodology Steps, Phases and Chapters

In the analysis phase, three steps are outlined. In the development of ALPH, these three steps were required to assemble domain knowledge that form the basis of the ALPH model. The steps, as they apply specifically to the ALPH model are:

- Domain analysis of pervasive healthcare application functionality;
- Further analysis to identify the set of crosscutting concerns in pervasive healthcare applications;
- Formation of a comprehensive set of pervasive healthcare concerns.

Each step in the analysis stage of the ALPH model requires significant investigation in the domain of pervasive healthcare and contributes sequentially to the development of the model proposed in this work. The first step involves domain analysis to identify application requirements and common domain functionality (section 3.3). The second involves the investigation of the nature of the functionality from phase 1, to identify crosscutting concerns (section 3.4). The third step involves the collation of identified crosscutting concerns into a set of pervasive healthcare concerns (section 3.5). These concerns are described in detail later in this chapter (section 3.6).

3.3 Domain Analysis

Analysis of a domain provides fundamental knowledge and understanding of domain functionality. Domain analysis was performed on four sources of domain knowledge [179] in the domain of pervasive healthcare, as shown in figure 3.2. Requirements analysis and elicitation works in the domain were examined, a literature review of pervasive healthcare applications was performed, pervasive healthcare case studies were studied and available codebases were examined. We describe the process of domain analysis in each group in the following sections.

Pervasive healthcare concerns identified in domain analysis are documented throughout this chapter. The terms used to label concerns, as in tables 3.1, 3.2, 3.4, are representative of the concerns identified. Variations in wording was observed and concerns were allocated to an appropriate concern category if it fell under the heading of an existing concern to maintain consistency, as illustrated by a selection of examples in figure 3.3.

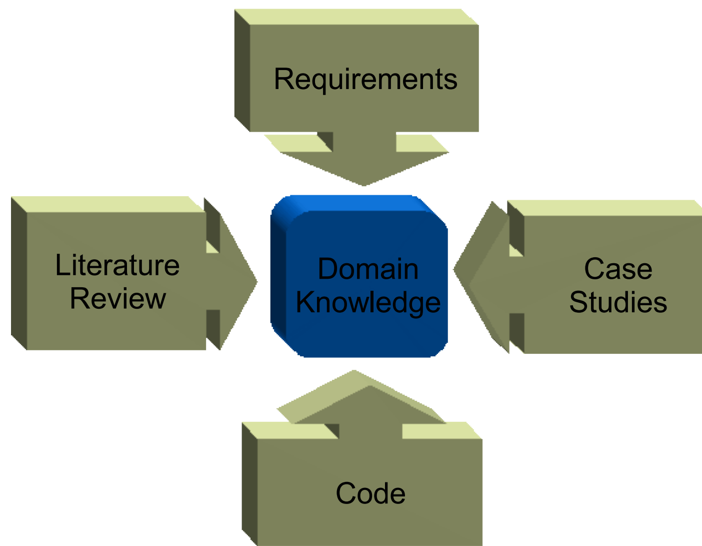

Fig. 3.2: Domain Analysis

LANGUAGE USED	*CONCERN CATEGORY*
Electronic patient records	EHR
On the move	Mobility
Reconfiguration	Context Adaptation
Mobile devices	Mobility
Devices dynamically joining and leaving	Device Discovery
Device communication	Communication
Context-aware devices	Context Adaptation
Receiving and producing messages	Communication \ HL7
Data storage	Persistence
GPS	Location
Patient records	EHR
Nomadic devices	Mobility
Data access and management	Persistence
Show different locations	Location
Medicine schema	EHR
Problem of immobility	Mobility
Devices over wireless network	Communication / Distribution
React upon certain changes of context	Context Adaptation

Fig. 3.3: Concern Terms

3.3.1 Requirements for Pervasive Healthcare Applications

The goal of the first step in designing the ALPH model is to identify the common functionality within the domain of pervasive healthcare. In general, requirements outline the functionality required by applications [240]. Requirements should describe both the core application logic,

the technical constraints and the underlying infrastructure required for application deployment in its environment. We examine requirements specifications in the domain of pervasive healthcare to identify functionality that reoccurs frequently in the domain.

	HL7	EHR	Context-aware Adaptation	Persistence	Distribution	Software Roaming	Network Roaming	Service Discovery	Device Discovery	Limited connectivity	Location	QoS	Communication	Ad hoc networking	Mobility	Security	Fault tolerance	Transactions	Performance	Availability	Reasoning
Req. For Mobile Middleware	•	•	•	•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	
Pervasive Healthcare Req. Engineering		•	•		•			•	•	•	•		•		•	•					•
Software Req. For Pervasive Healthcare			•			•	•	•	•				•		•						
Literature Review- Pervasive Healthcare	•	•	•	•	•			•	•		•		•		•	•			•	•	•

• = Concern required

Table 3.1: Pervasive healthcare requirements in domain analysis

Several requirements analysis projects have taken place in the domain of pervasive healthcare. For example, extensive work has been undertaken by the Centre for Pervasive Healthcare and various other projects have also contributed to the area [279] [198].

The Centre for Pervasive Healthcare has produced two sets of requirements in the domain. The first set of requirements is described following requirements elicitation from applications in the area of pervasive healthcare [208]. Requirements identified include distribution, quality of service, security, discovery, ad-hoc networking, adaptation to contextual information and location along with other mobility based concerns. The concerns identified in this work can be seen in row 1 of table 3.1. The second set of requirements emerged from a formal requirements engineering project for a pervasive healthcare system [139]. This project collaborated with a local hospital and elicited requirements from real world scenarios and academic future scenarios devised in collaboration with medical personnel. Requirements are described using executable use cases [140], Petri Nets [141], informal descriptions [37] and are represented in row 2 of table 3.1.

Support for mobile, heterogeneous devices, discovery mechanisms, managing context information, supporting interaction and supporting migration are examples of infrastructural software requirements for pervasive healthcare systems [279]. We depict these in row 3 of table 3.1.

Orwat et al outline a literature survey investigating 67 different pervasive healthcare systems [198]. The goal of the survey was to find application requirements by categorising applications based on purpose, deployment setting, user groups and system features. The requirements identified are represented in row 4 of table 3.1.

3.3.2 Application Literature Review

During domain analysis, a literature review of pervasive healthcare applications was conducted to identify reoccurring pervasive healthcare behaviour. Applications are either patient or health care professional based i.e., used in the patient's home or within a healthcare facility environment [192] and both were investigated to gain a comprehensive view of functionality provided by pervasive healthcare applications. Literature describing pervasive healthcare systems and their specifications were examined and the concerns within the applications were identified informally i.e., by reading and analysing application descriptions and requirements. A taxonomy of pervasive healthcare applications [192] was also examined in the literature review. For space reasons, all application literature examined cannot be detailed so a representative selection of applications that show the set of concerns identified are outlined in table 3.2.

3.3.3 Pervasive Healthcare Case Studies

Scenarios are an important method of eliciting and specifying software behaviour [163]. The information contained in a scenario represents a specification that models domain knowledge [248] and is an evolving description of situations in the environment [163]. Case studies that describe applications based on scenarios can be used to investigate domain requirements [248] [269]. In our domain analysis, we included two case studies that depict scenarios in pervasive environments. The first case study is based on an optimal application solution for

	Context-aware Adaptation	Distribution	Service Discovery	Device Discovery	Location	QoS	Communication	Mobility	Security	Reasoning	Proximity	Time
AwareHome	•				•					•		
Multimodel For Older Users	•	•			•		•			•		
Autominder	•	•					•	•		•		•
Orientation Aid For Amnesiacs	•	•			•		•	•		•		•
Flexible Technologies And Smart Clothing	•				•					•		
Mobility In Healthcare Application	•	•	•	•	•	•	•	•	•	•	•	

• = Concern Required

Table 3.2: Pervasive healthcare literature analysis

	HL7	EHR	Context-aware Adaptation	Persistence	Distribution	Software Roaming	Network Roaming	Service Discovery	Device Discovery	Limited connectivity	Location	QoS	Communication	Ad hoc networking	Mobility	Security	Fault tolerance	Transactions	Performance	Availability
Auction Scenario			•		•	•	•	•	•		•	•	•		•					
Hospital Scenario	•		•	•	•			•	•		•		•	•	•	•	•	•	•	•

• = Concern required

Table 3.3: Pervasive healthcare case study analysis

a future scenario in pervasive healthcare by Raatikainen et al [208]. Clinicians were heavily engaged in the process of envisioning, designing and testing proposals for future support in healthcare applications for pervasive computing technology [3]. The scenario requires a pervasive computing infrastructure along with healthcare specific functionality. Concerns were identified from the scenario description and analysis and are shown in figure 3.3. The second case study examined outlines a pervasive web-based application that requires support

for mobility and context-awareness. The case study application is based on an online auction scenario. This case study considers the various mobility issues that can affect the average distributed application scenario [173]. Its requirements were outlined by a Swiss research laboratory [13] and it was extended to execute on mobile devices in a pervasive environment. The case study scenario was analysed and concerns relating to pervasive healthcare were identified as illustrated in table 3.3.

3.3.4 Pervasive Healthcare Codebases

Requirements can be extracted by "mining" domain knowledge from code [179]. The proven linkage between source code and domain knowledge [167] [166] makes code from applications in a target domain a source of domain knowledge [179]. Domain knowledge and domain concepts are embedded in source code e.g., as the names of variables, methods and classes [278]. We requested access to pervasive healthcare application codebases from many sources. However, a significant problem in this endeavour is the fact that pervasive healthcare is a "nascent" or emerging domain [193]. Few operational systems exist, and where they do, they are generally proprietary and code is unavailable. One operational hospital scheduling and awareness system in Denmark, known as the iHopsital [1], started as an academic project in the Centre for Pervasive Healthcare but the project was taken over and the system was made commercial, making the code unavailable.

Nonetheless, we obtained codebases for several academic applications from Trinity College and two third party applications. Two of the applications explicitly addressed healthcare behaviour while the rest contributed from a pervasive perspective. As pervasive healthcare applies pervasive technology in the healthcare domain, pervasive applications enable the identification of pervasive computing concerns that are required by pervasive healthcare applications. The third party applications were sourced from Lancaster University [120] and from a Dublin based software company that is a leader in Bluetooth technology [10]. Trinity College applications were sourced from unrelated projects within the research group where the ALPH model originated. The first *Dynamic Healthcare Scheduling* application is a per-

[1] http://www.ihospital.dk/

	HL7	EHR	Context-aware Adaptation	Persistence	Distribution	Software Roaming	Network Roaming	Service Discovery	Device Discovery	Limited Connectivity	Location	QoS	Ad hoc Networking	Communication
Riddlehunt					●●	●●		●●	●●	●●	●●	●	●	●●
Quazoom					●●	●●	●●	●●	●	●●	●●	●	●●	●●
TCI					●●	●				●●	●●	●		●●
Healthwatcher				●●	●●									●●
Rococo								●●	●●					●
Dynamic Healthcare Scheduling			●●		●●	●		●●	●●	●	●●		●●	●●
Oisin			●●		●●	●		●●	●●	●	●●		●	●●

●●= Essential Concern, ● = Ancillary Concern

Table 3.4: Pervasive Healthcare codebase analysis

vasive healthcare application that facilitates dynamic scheduling for healthcare professionals [88]. The application, deployed on PDAs, uses Hermes framework [86] components to support mobile context-aware activity scheduling for healthcare professionals. The application is aware of the physical and social situation in which they are deployed, detecting and reasoning about sensor information from the environment to appropriately reschedule the "to-do list" of medical workers. *Riddlehunt* is a mobile treasure hunt game, where players compete at answering puzzles left at different geographical locations. *Riddlehunt* is built on a generic framework for mobile, context-aware applications [86] and is based on the concepts of trails, i.e., collections of activities with contextual information and a visiting order between them. The application uses the location of the user to determine which riddles the user is able to solve. The application is run on PDA devices which make use of external GPS location sensors. The application is dynamically reconfigured according to the user's mobile context. *Oisin* is another context-aware mobile application that is also based on the notion of trails [87]. It involves collaborative work in a mobile, context-aware environment and the reconfiguration of the application depending on contextual information [87]. The *Quazoom* game

uses the FLARE platform [149] which enables the development of mobile, location aware applications. The application is a multiplayer game run on various mobile devices including laptops, PDAs and wearable computers. The actual position of the user in the environment is mapped into a virtual environment, which is overlaid onto the real world. Network partitions due to limited connectivity must be handled frequently. The *Traffic Congestion Indicator* (TCI) project [84] developed a traffic congestion level indicator which uses real-time data to provide traffic information for a predefined area. The information is based on the location of the mobile user.

Table 3.4 depicts the concerns identified in the code examined. Note that only two applications were targeted specifically at the pervasive healthcare domain. The remainder are pervasive computing applications that indicate concerns required in a pervasive environment. Boxes checked with two bullets represent concerns which are essential for the application to function correctly, whereas concerns checked with one bullet can be regarded as ancillary functions for the application. This is discussed further in section 3.5.

3.4 Crosscutting concerns

In the domain analysis step, applications, case studies and articles were examined to extract and identify common pervasive healthcare concerns in the domain. This generated a set of reoccurring domain-specific concerns. Further examination of applications took place to assess the crosscutting nature of the domain-specific functionality.

The examination of the codebases enabled us to observe the degree to which each concern affected the application as a whole and to verify the classification of concerns as crosscutting. A classification technique, by Eaddy, Aho and Murphy [91], based on concern identification heuristics was used to identify crosscutting concerns in application codebases. This approach is more accurate and easier to apply than other approaches including aspect mining and static analysis techniques [91]. This systematic methodology defines guidelines for manually identifying concerns and their associated code fragments through analysis of sourcecode.

A concern is scattered if it is connected to multiple elements (i.e., more than one), and tangled if both it and at least one other concern are related to the same target element [101].

Fig. 3.4: Codebase with Highlighted Crosscutting Concerns

Elements considered in OO implementations include files, classes, fields, methods, statements, and statement blocks (for-loops, if-then-else blocks, etc.). The structure of crosscutting concerns differ depending on the concern implementation. Structures of crosscutting concerns manifesting themselves over a system range from a concern that touches only a few points in

a system to a concern that is partially modularised by one or more classes but is also spread across a number of others [89].

To illustrate the crosscutting effect visible on pervasive healthcare applications, figure 3.4 illustrates an application containing crosscutting concerns. This is an application supporting wireless communication using mobile devices [10]. We do not discuss the code in detail, as it is both verbose and low-level. A high-level view of the application as a whole is sufficient to demonstrate the effect of crosscutting concerns in pervasive healthcare applications. Figure 3.4 provides a graphical illustration of the crosscutting nature of concerns within the example application code. Concerns are highlighted in various colours e.g., discovery in blue, distribution in yellow, limited connectivity in pink, distributed communication in red and device context adaptation in green.

3.5 Concern Selection

The goal of the domain analysis was to find a set of pervasive healthcare concerns that would be considered for the inclusion in the ALPH model. The concerns that emerged include functionality that ranged in applicability from one application to many applications. To scope the ALPH model, concerns identified in domain analysis were considered for inclusion in the ALPH model based on the following criteria.

- The primary heuristic used to determine the inclusion of a concern is based on a term's frequency of use in the domain, relative to the total number of unique occurences.
- Concerns selected for inclusion must have been identified as crosscutting using concern identification heuristics. These crosscutting concerns contribute to complexity in pervasive healthcare applications and can be addressed by modularity in the ALPH model.

Concerns were eliminated based on the number of applications requiring their functionality and on their crosscutting nature. This eliminates outliers that occurred infrequently and are not considered common domain functionality. Healthcare specific concerns were weighted

in the calculation of frequency. This was necessary due to the lack of access to pervasive healthcare applications resulting in the prevalence of pervasive computing applications and codebases used in the identification of concerns. Healthcare specific concerns are also intrinsically linked to the communication and persistence of information in pervasive healthcare applications. Following the examination of domain analysis outputs, the most frequently occurring concerns (>7 occurances (weighting of +100% to healthcare concerns)) were selected for inclusion in the ALPH model. The most frequently occurring crosscutting concerns form a conclusive set of pervasive healthcare concerns include 8 mobility based concerns, 2 context based concerns and 3 healthcare based concerns. The set of pervasive healthcare concerns identified are described in the following section.

3.6 Pervasive Healthcare Concerns

The analysis phase in DSL design methodology requires input of documents or code from which domain knowledge can be obtained [179]. The reoccurrence of functionality and the repetition of domain-specific tasks throughout these inputs results in the identification of domain-specific concerns. The output of domain analysis varies, but is some representation of the domain knowledge obtained [179]. The variance in forms of knowledge extend from high-level UML diagrams through design techniques to domain implementations [179]. Output from domain analysis in the ALPH model is at the implementation end of the output spectrum in the form of "a domain implementation consisting of a set of domain-specific reusable components" [179].

This section describes the set of pervasive healthcare crosscutting concerns selected in the domain analysis phase. Each concern relates to a particular functional or non-functional requirement of pervasive healthcare applications. Each concern has been identified as common and reoccurring across domain applications and has also been recognised as crosscutting. These concerns form the basis for the pervasive healthcare functionality provided by the ALPH model. We group these concerns in three categories relating to: mobility, context and healthcare. The following sections describe the set of pervasive healthcare concerns under these headings.

3.6.1 Mobility

Developments in wireless technology facilitate the move from static physical locations to mobile environments, enabling the paradigm of "mobile computing" [114]. Mobile computing allows users to move freely through a geographical space while connected to resources and other users through wireless links [114].

Mobility is a profound characteristic of hospitals and healthcare facilities [36]. Medical work is highly mobile [31] and subsequently healthcare professionals are a highly mobile populace, so support for mobility in pervasive healthcare applications is essential [104]. Working in a hospital environment requires moving between different locations of work [155]. Healthcare professionals may cover distances up to 15 km during a shift in a hospital environment [36]. This movement between patients, workplaces and locations, is a normal mode of work in such environments but little or no technology supports this type of mobility in healthcare applications [46].

Mobile access to healthcare information is key to improving medical work [46]. Increased mobility also improves patient care, improves work systems [46] and allows healthcare professionals to spend more time with patients and less time gathering information [174].

Concerns relating to mobility featured heavily in the analysis of the pervasive healthcare domain. These concerns are required to support healthcare professionals as they work while on the move [36]. The following sections describe the selected concerns that relate to mobility, namely; distribution, communication, roaming, device discovery, service discovery, limited connectivity, location and quality of service. A high-level description is given in this chapter, as further description, code based designs and implementation details are described in Chapter 4.

3.6.1.1 Distribution

Distribution addresses the geographical and technical dispersion of nodes and the underlying architecture used to support the transfer of data between nodes [187]. Distribution is a central characteristics of pervasive healthcare environments [216] as applications deployed in these environments are distributed both geographically and logically [186]. ALPH facilitates

distribution using Java distribution techniques e.g., RMI, Sockets. This provides underlying distribution for applications using the ALPH model.

3.6.1.2 Communication

Distributed applications require the use of network communication services to communicate with other distributed nodes [227]. Distributed communication implementations, such as over Java sockets or web services, provide reliable, efficient and flexible means of communication [29]. Information communicated over these channels is subject to formatting according to the applications transmitting and receiving the information e.g., HL7 for healthcare information.

3.6.1.3 Network Roaming

An important aspect of mobility handling is client roaming. When a user moves around a geographic region, it is likely that different networks provide the best service in different locations, and applications should adapt accordingly. We refer to the physical roaming of a client as Network Roaming. Network roaming functionality in mobile systems identifies and handles roaming events frequently during execution. When handling network change, consideration should be given to changes in network features such as latency, bandwidth, and any security provided by the network.

3.6.1.4 Software Roaming

The ability to move code across the nodes of a network is known as software roaming, or code mobility [114]. Software roaming can be seen as changing the bindings between code fragments and the location where they execute [114]. In pervasive healthcare applications, certain tasks may require behaviour to occur at another location e.g., the acquisition of information in a remote database. The use of mobile software agent systems is a technique in achieving software roaming. The mobile software-agent paradigm allows data or state to relocate from one machine to another [267]. A mobile software agent is designed to migrate from one machine to another to carry out certain operations [267].

3.6.1.5 Service Discovery

Pervasive environments are dynamic, observing services entering and leaving the environment in various locations, constantly changing the view of resources within the system. Applications that use services in a mobile environment require the ability to dynamically discover and acquire access to necessary services [160]. ALPH includes service discovery functionality to facilitate the location of available services. Services advertise their existence on the network and clients request a particular service and obtain access information for the most suitable service in their current context [188]. The requested service is accessed via a service discovery protocol. When the requested service is accessed, its availability is confirmed and the application can proceed to use the service.

3.6.1.6 Device Discovery

As the user moves in the environment, new devices may come into communication range and enable new interaction possibilities. It is often useful for an application to be notified when new devices enter the environment e.g., the discovery of colleagues work nodes. These notifications inform the application about a remote device's status and interaction capabilities. Currently, all wireless network standards support some kind of device discovery [217]. Whenever the appearance or disappearance of a device from the environment is detected, the application should react by applying an appropriate strategy, e.g., connection or disconnection.

3.6.1.7 Limited Connectivity

In pervasive environments, network connectivity is no longer an expensive add-on, rather it is a basic feature of any computing facility [114]. Users of mobile devices experience periods of direct connectivity, intermittent connectivity, and disconnectivity and can have varying amounts of bandwidth available to them as well as different networks latencies depending on how they connect to the distributed application servers [20]. Limited connectivity hinders users of portable devices from accessing distributed information systems, whenever they need to [24]. Disconnections occur in pervasive healthcare environments for several reasons e.g.,

due to path loss, poor network coverage, failed network handovers, severe network congestion and client hardware failures [263]. Disconnections must be detected in a timely fashion and contingency plans are required to adequately handle abrupt disconnections e.g., facilities for rollbacks to ensure consistency in system state [188].

3.6.1.8 Location

In pervasive environments, healthcare applications often need to dynamically obtain information that is relevant to their current location. Applications can adapt their behaviour based on the current location of the device e.g., prescriptions may only be written within the boundaries of the building. Pervasive applications use location systems to acquire location information [211]. Each location system has its own accuracy and representation of location data e.g., GPS.

3.6.1.9 Quality of Service

Quality of Service (QoS) assurances are significantly diminished when associated with a mobile wireless environment [61]. The various choices of networks available provide fundamentally different degrees of QoS, ranging from CAN networks with strong timing and reliability behaviour to weaker guarantees in wireless ad hoc networks which are prone to high latency, low bandwidth, and have a high probability of being congested and suffering collisions. To handle the varying degrees of QoS assurances bound to particular networks, applications need to be able to associate a QoS rating to each network and adjust its communications strategy accordingly. Applications require different degrees of QoS under different configurations and environmental conditions, and multiple QoS properties must be combined with and/or traded off against each other to achieve the optimal application outcome [227].

3.6.2 Context

Adaptation in response to contextual information from a pervasive healthcare environment featured heavily in the applications, case studies, papers and codebases examined in domain analysis. Context-awareness allows applications to dynamically and appropriately adapt to

their changing environment. Many types of context were identified and applications reacted and adapted differently based on the context type. Schmidt et. al. describes a hierarchically-defined context space used here to categorise context types [229]. We divided the overall context space into eight sub-categories: device, location, user, social, environmental, system, temporal, and application-specific context [190]. The ALPH model provides explicit functionality to adapt to context types that featured most frequently ; device context and location context. We include descriptions of the other six context types as these were investigated further and prototyped in the development of the ALPH model [190].

3.6.2.1 Device Context

Device context encompasses information that describes the device on which the application is executed. Device information includes screen size, colour depth, processing power and storage capacity [229]. The acquisition of device context allows applications to adapt according to both the hardware and software characteristics of a particular computing device.

3.6.2.2 Location Context

Location is central to context-awareness [16] as geographical location enables the identification of users in an environment and can infer higher-level context information e.g., user is present, user is in a particular room. While Schmidt et al [229] argue that location is only one of many context types, it remains the most widely-used element of context in context-aware applications [22] [40] and appeared most frequently as context in the domain analysis of pervasive healthcare described earlier in this chapter. Adaptation to location information is useful when using mobile devices as location information can be used to infer higher-level contextual information e.g., proximity to other entities [190]. Application adaptation takes place in response to low-level, sensor location information based on the device's current location. This information can be acquired from various forms of location data providers e.g., GPS, Cell IDs.

3.6.2.3 Other Context Types

The ALPH model does not provides explicit functionality in its current implementation for adaptation to the following context types although the were prototyped and used in the development of the ALPH model [190]. These context types occurred infrequently and so were not selected as "common" context types for inclusion in the ALPH model.

User Context

User context represents information relating to the user of the application in question. This includes all knowledge pertaining to the user that is known to an application i.e., the physiological context, emotional state, current activity and the user's schedule [190].

Social Context

Social context is defined as any information that is relevant to the characterisation of a situation that influences the interactions of one user with other users and that describes the relationships of the user to other people [266]. Again, social context enables the deduction of higher-level context states e.g., a "free for lunch" context may be inferred from contextual information relating to the location and availability of the user.

Environmental Context

Environmental context is defined as relating to the physical world in which the application runs, e.g., temperature, noise and lighting conditions [16]. This type of contextual information is acquired by sensors in the physical environment of the application.

Temporal Context

Temporal context encompasses all data related to time, e.g., current time, day, month or year [16]. Temporal context can be used to adapt applications according to schedules, plans, calendars and reminders.

Application Context

Application context represents information directly related to the core functionality of an application [190]. This information is dependent on the business rules and functionality of any particular application and so only limited support can be provided generically.

System Context

System context represents context related to the technical infrastructure in which an appli-

cation executes, e.g., available computation resources, communication capabilities, and information pertaining to the system components and their configuration [229]. Often, system context relates to performance requirements, hardware availability and network conditions [190].

3.6.3 Healthcare

When applications are deployed in a healthcare environment, additional healthcare specific concerns are encountered. In the domain analysis phase of the ALPH model a number of healthcare specific concerns common to multiple pervasive healthcare applications were identified namely; the use of healthcare standards, EHRs and persistence. The use of varying hospital and departmental information systems result in various formats of data, inhibiting interoperablity between systems and institutions. A number of standards have emerged to address this problem and conformance to standards must be implemented in order to enable interoperability of both in-house heterogeneous systems and inter-establishment systems. The HL7 standard is a key concern in the domain to enable the exchange of standardised electronic healthcare information. There are also design standards that should be considered such as EHRs. EHRs are full patient records consisting of data from various hospital and healthcare professional systems. Storage and persistence related functionality is also very prevalent in healthcare systems affecting multiple modules. We describe the healthcare specific concerns in the set of pervasive healthcare crosscutting concerns within the ALPH model below.

3.6.3.1 HL7

Healthcare information is central to pervasive healthcare applications, as discussed in 2.1.2. Access to healthcare information is required at the point of care to facilitate decision making [104]. To distribute healthcare information it must be captured in a standard format. This standardisation defines layouts that enable the transfer and use of healthcare information in heterogeneous healthcare information systems. The formatting of information makes it available for distribution in pervasive healthcare applications. The international HL7 standards institution promotes and enforces the standardisation of electronic healthcare information

to facilitate its communication and management [186]. In domain analysis, many pervasive healthcare applications had a requirement for the distributed communication of information, with no explicit reference to HL7. However, applications that communicate patient data should conform to the international HL7 communication protocols and wrap all healthcare information in the appropriate HL7 messaging format [225] [196].

3.6.3.2 EHR

Comprehensive EHRs are patient records consisting of files and components from various hospital and healthcare professional systems. Large pervasive healthcare systems aim to incorporate a form of EHR to structure patient data [143]. However, EHRs have proven extremely difficult to develop [44]. Huge duplication exists even in situations where EHRs have been implemented for logging and paper trails [187] functionality. The release of open source solutions provides an alternative to expensive vendor EHR systems. New EHR ventures, such as Google Health and Microsoft's Health-Vault, highlight the requirement for modularisation, enabling EHR systems to be pluggable components [187]. EHR systems currently being implemented in hospitals run on desktop PCs, and do not have any support for mobility [31]. Taking the degree of mobility within hospitals into account, "there is a substantial motivation for both EHR vendors as well as the hospital administration to develop solutions for mobile access to the EHR systems" [31].

3.6.3.3 Persistence

Healthcare systems require a large amount of data-centric functionality. Electronic healthcare information must be persisted and details are retrieved from datastores frequently. A previous programming language targeted at the healthcare domain was predominantly persistence based [52]. In pervasive healthcare applications, remote connections to datastores are required throughout the application to persist healthcare data, often in conjunction with an EHR system.

3.7 Summary

This chapter described the analysis phase of the ALPH model proposed in this thesis. The overall methodology used for the development of the ALPH model was outlined. The design steps of the methodology correspond to the analysis phase described in this chapter.

Domain analysis provides fundamental knowledge and understanding of domain functionality and identifies the common functionality within the domain of pervasive healthcare. The analysis carried out in the development of ALPH included the analysis of requirements in the domain, analysis of literature in the domain, studying domain case studies and examining codebases. Common concerns were identified and analysed to identify crosscutting concerns. The output of the analysis phase is the set of crosscutting pervasive healthcare concerns that forms the basis of the ALPH model, shown in table 3.5. This resulting set of pervasive healthcare concerns that are both reoccurring and crosscutting are inlcuded in the ALPH model and will be made available through the ALPH DSL.

Distribution
Communication
Network Roaming
Software Roaming
Service Discovery
Device Discovery
Limited Connectivity
Quality of Service
Device Adaptation
Location
EHR
HL7
Persistence

Table 3.5: Concern Summary

The next chapter describes the design and implementation phase of the concerns in the

ALPH model. The implementation details of the library of crosscutting pervasive healthcare concerns in the ALPH model are described, including the set of techniques and designs used in the implementation of each concern. Each concern's behaviour and modular design is described with details of the classes and aspects involved.

Chapter 4

Pervasive Healthcare Aspect Library

Previous chapters have described complexity factors in pervasive healthcare application development and identified a set of concerns in the domain that crosscut applications. These crosscutting concerns reduce modularity and hence increase complexity. This chapter presents a modular implementation of a library of crosscutting pervasive healthcare concerns. The set of concerns, as listed in section 3.6, were identified during a domain analysis in the area of pervasive healthcare application development. The modular implementations in this chapter provide the underlying behaviour available using the ALPH model.

The chapter begins with a high-level overview of the library as an element of the ALPH model. The languages and tools used to design, develop and describe the implementations are detailed. This is followed by a description of the modular implementation of each concern within the library. The set of aspects and classes involved in each concern are presented. Design details and implementation descriptions are supplemented with sequence diagrams.

4.1 Library Overview

The ALPH model's goal of reducing complexity in pervasive healthcare applications requires us to address difficulties with modularisation and inappropriate levels of abstraction. Within

the ALPH model, behaviour common to pervasive healthcare systems is made available to the developer, aiding application development in the domain. To address complexity in the applications, this behaviour must be both modularised and abstracted. Following the identification of crosscutting concerns through domain analysis, the modularisation of crosscutting pervasive healthcare concerns is the next step in reducing complexity in pervasive healthcare applications.

Poor modularity is a source of complexity in both application code and in application development, as discussed in section 1.4.1. Poor modularisation increases complexity by reducing code quality [56] and negatively impacts on manageability, maintainability and understandability [201] [129]. It is caused by the crosscutting nature of behaviour in applications. The previous chapter investigated the behaviour in pervasive healthcare applications and identified a set of crosscutting concerns. These concerns affect applications in multiple areas and concern-specific code is intertwined with base application logic. Crosscutting concerns require advanced modularisation techniques as object-oriented techniques lack the capacity to cleanly encapsulate their code.

The ALPH model employs AOP to modularise crosscutting concerns. Aspect-oriented separation of crosscutting pervasive healthcare concerns maximises the independence of both the concerns themselves and the base application logic modules [200]. This "separation of concerns", discussed in section 2.2.0.1, reduces application complexity while increasing application modularity [265] [96].

As discussed in section 3.6, the first step of DSL design methodology is to produce a model or representation of the domain knowledge through domain analysis [179]. In the ALPH model, domain analysis resulted in the identification of a set of reoccurring crosscutting concerns in pervasive healthcare applications. This knowledge is the basis for the library of elements in the ALPH model, i.e., "a domain implementation consisting of a set of domain-specific reusable components" [179]. This chapter presents the implementation details of the modular design and development of the crosscutting concerns that make up the library of concerns in the ALPH model, as shown in figure 4.1. Each concern addresses a common, crosscutting requirement in pervasive healthcare applications and is modularised using AOP.

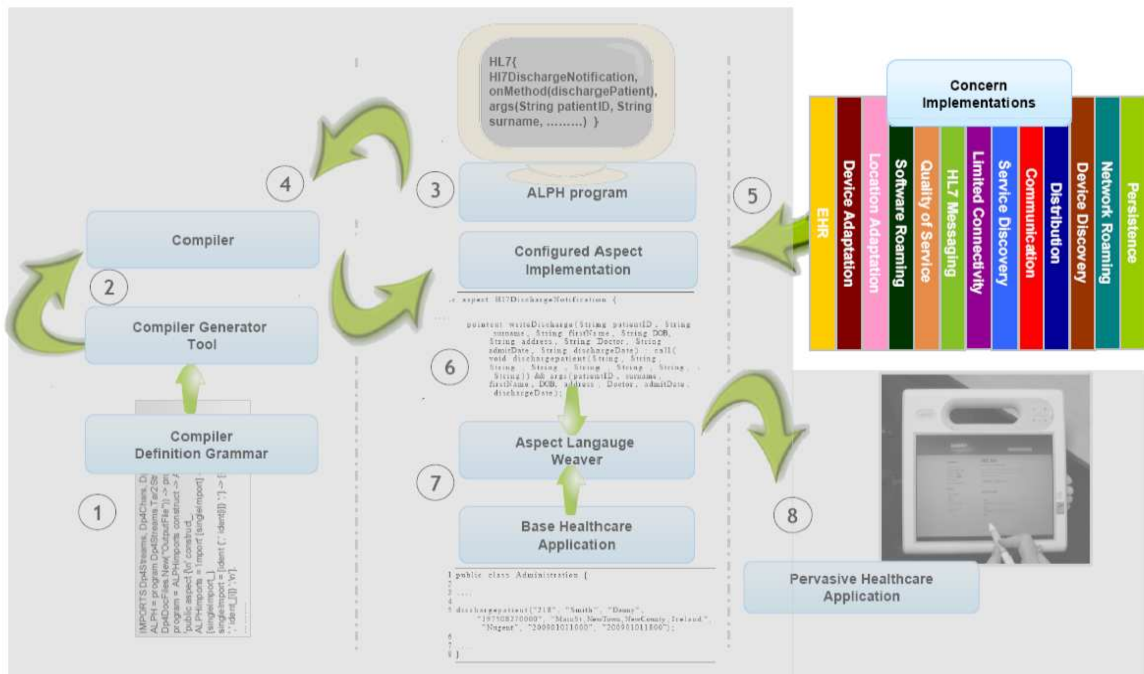

Fig. 4.1: Library in the ALPH Model

4.2 Language and Tools

In this chapter, we capture pervasive healthcare concerns at the design and implementation level using an aspect-oriented UML-based approach, Theme/UML [67] and the implementation level languages AspectJ [147] and Java. There are two benefits of this approach. First, it allows us to communicate the behavioural impact of each concern on an application. Second, it illustrates a modular design to separate concerns from the rest of the base application, achieving better modularity in this complex domain as a result. In this section, we describe both Theme/UML and AspectJ.

4.2.1 Theme/UML

Theme/UML [67] extends standard UML to support modularisation and composition specification of concerns that cut across a system. In this chapter, mostly standard UML is employed, with two new notations. First, the designer of a concern that has a behavioural

impact on the base application must be able to abstractly refer to those places in the application where the additional behaviour is to occur. This is done using templates. In Theme/UML, each concern is modularised into a package called a theme, and the templates appear in the top right-hand corner of the theme. Each template represents points in the base application that can be referred to within the theme. The second notation relates to how the templates are used in standard sequence diagrams. The template triggers augment behaviour as defined by the concern, with the sequence diagram indicating where the concern's behaviour should occur relative to the execution of the base behaviour. The notation "do" prepended to the template name on an operation call in the sequence diagram indicates where the base behaviour should occur.

4.2.2 AspectJ

AspectJ is a general purpose aspect-oriented extension to the Java language [172]. The salient features of AspectJ, as outlined in section 2.2.1, are aspects, join points, pointcuts and advice. Aspects are modules that encapsulate crosscutting concerns as code. These aspects define crosscutting functionality and where it should be applied in the base application by means of join points, pointcuts and advice. Join points are points in an application's execution where aspect behaviour can occur e.g., method calls, constructors, field references. Pointcuts are predicates that define a join point, or set of join points, where aspect behaviour should be applied when a pointcut is matched during execution. The behaviour implementation is defined in advice along with a keyword to indicate if the code should execute before, after, or around the matched pointcut.

Applications requiring crosscutting concerns modularised using AspectJ are compositions of both objects and aspects. Core application logic is implemented using Java and crosscutting concerns are encapsulated in aspects. These components are combined by the AspectJ weaver, ensuring that crosscutting behaviour is carried out during base application execution where required. The ALPH model proposed in this thesis uses AspectJ to develop modularised implementations of crosscutting pervasive healthcare concerns. Therefore, for the purpose of prototypical demonstration, the model is for use with base pervasive healthcare applications

programmed in Java. The use of AspectJ, and subsequent use of base applications written in Java, is an implementation decision for the current implementation of the ALPH model. Concerns could be re-implemented using an alternative aspect language to support other base languages e.g., AspectC++ [245].

4.3 Concern Implementations

The following sections contain detailed descriptions of each concern in the set of crosscutting pervasive healthcare concerns identified in Chapter 3.

4.3.1 Distribution

Distribution enables the geographical and technical dispersion of nodes and provides the underlying architecture used to support the transfer of data between nodes [187]. ALPH provides distribution functionality using Java RMI [85]. RMI is based on a client-server model and provides mechanisms, for both client and server side applications, to facilitate the availability of remote objects residing in different JVMs. RMI requires the following actions to be taken:

- Remote interfaces must define methods available remotely
- Remote objects must implement at least one remote interface
- Remote objects must be exported
- Remote objects must be bound to the RMIRegistry by the Server
- Remote references must be located in the RMIRegistry by the client using JNDI
- Clients can then make calls to methods on remote objects

Given the amount of code relating to these actions, encapsulating distribution using RMI as an aspect is a complicated process. Each action must be carried out transparently, i.e., the base application must remain oblivious to the RMI distribution. The design implemented by

the ALPH model is inspired by the aspect refactoring method by Ceccato and Tonella [59]. The components within the distribution model are illustrated in figure 4.2.

To maintain separation between the base application and the distribution aspect, remote interfaces for objects being distributed are generated by the ALPH model. The remote interface generated extends `java.rmi.Remote`, which enables the inclusion of RMI basic functionality. The methods of the object being distributed, e.g., `ServerImpl`, are copied into a remote interface, called `IRemoteServer`. Within this interface, each method is defined and suffixed with "_remote". Each method signature also throws the required `RemoteException`. The distribution aspect, `InterfaceImplAspect`, defines that the original base class implements two interfaces by declaring them as parents. The first is the generated remote interface, `IRemoteServer`. Declaring that the base class implements `IRemoteServer` requires the introduction of `IRemoteServer`'s methods into the base class. `InterfaceImplAspect` introduces these methods, "_remoteMethodName" etc. with no implementation, as the aspect will intercept their execution and redirect the call to the original base class method. The second interface that the base class is defined to implement is the `IRemoteClass` interface. This interface does not require the base class to implement any methods and is used to distinguish local classes from remote classes in execution.

Figure 4.3 illustrates the sequence of events on the creation of each base class which is to be distributed. The `ConstructorInvocationAspect` aspect intercepts construction by recognising each class implementing the `IRemoteClass` interface. The code executed instead of the original constructor delegates the object creation to the factory, `ObjectFactory`. This `ObjectFactory` is implemented to create remote object implementations. Its `createObject` method obtains the internal details of the base class using reflection. The new remote object created is exported using `UnicastRemoteObject` and bound in the `RMIRegistry`, making it available remotely. When the object has been created on the remote host, an object of `Remote` type is returned. The client class that launched the call to the constructor is expecting an object of the base class type e.g., `ServerImpl`. To address this, we introduce a facade `ServerImpl` object on the client in which we wrap the remote object. To do this, the `InterfaceImplAspect` declares a new field for keeping a reference to the remote object

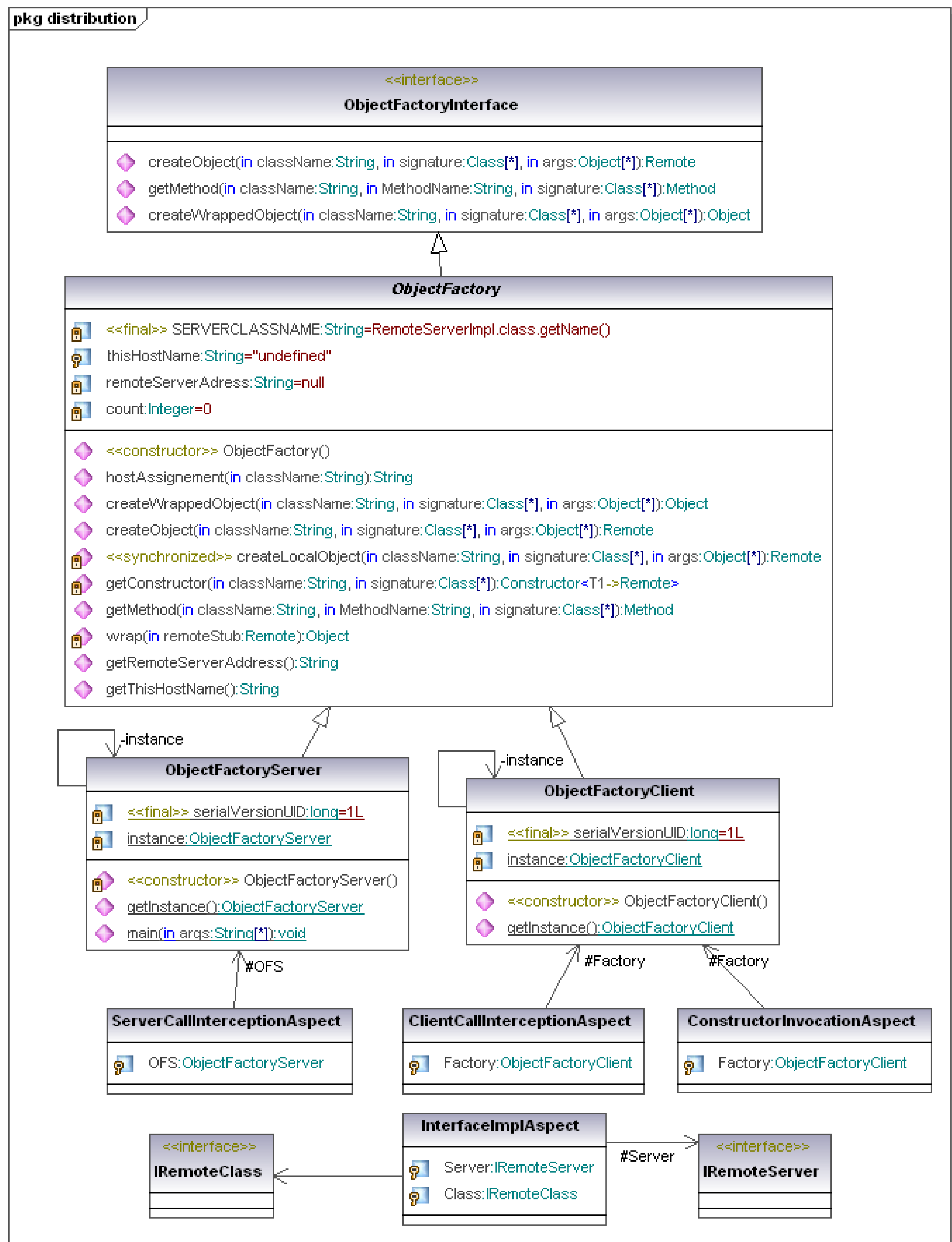

Fig. 4.2: Distribution Module Class Diagram

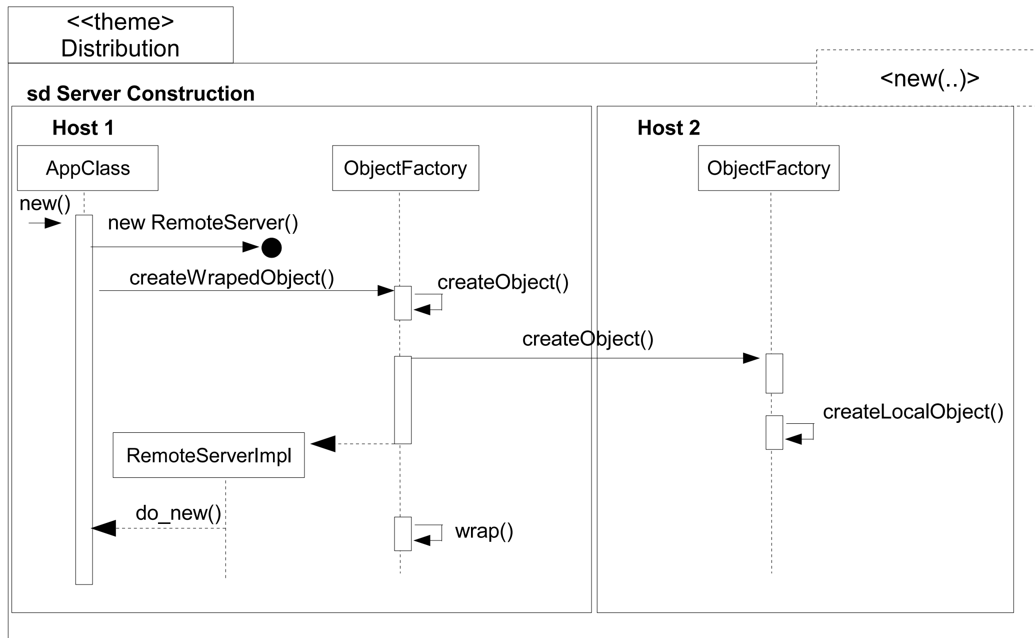

Fig. 4.3: Distribution on Server Construction

and a new constructor that assigns the remote reference. From the client's point of view, it communicates with a local instance of `ServerImpl` but as we will see in the next section, each call will be transferred to the real server on the remote host. The process of object construction in the distribution module of ALPH is illustrated in figure 4.3.

RMI is based on remote method invocations. When a method invocation occurs on an object that is being distributed by the ALPH model, the `ClientCallInterceptionAspect` aspect intercepts the invocation, as shown in figure 4.4. The aspect uses the reference of the real server which is contained in the field introduced to hold its reference. The invocation is replaced by a remote one, by concatenating the "_remote" suffix to call the method of the `IRemoteServer` interface and handled as a remote method invocation by RMI. This step is almost recreated backwards at the invocation of the remote method on the remote host. The `ClientCallInterceptionAspect` aspect on that host intercepts the remote method call. The call is then redirected to the original base class method implementation and the required behaviour is achieved. The `ServerCallInterceptionAspect` aspect intercepts all the call to the Remote Server methods and redirect their execution to their corresponding local method

on the local Server object.

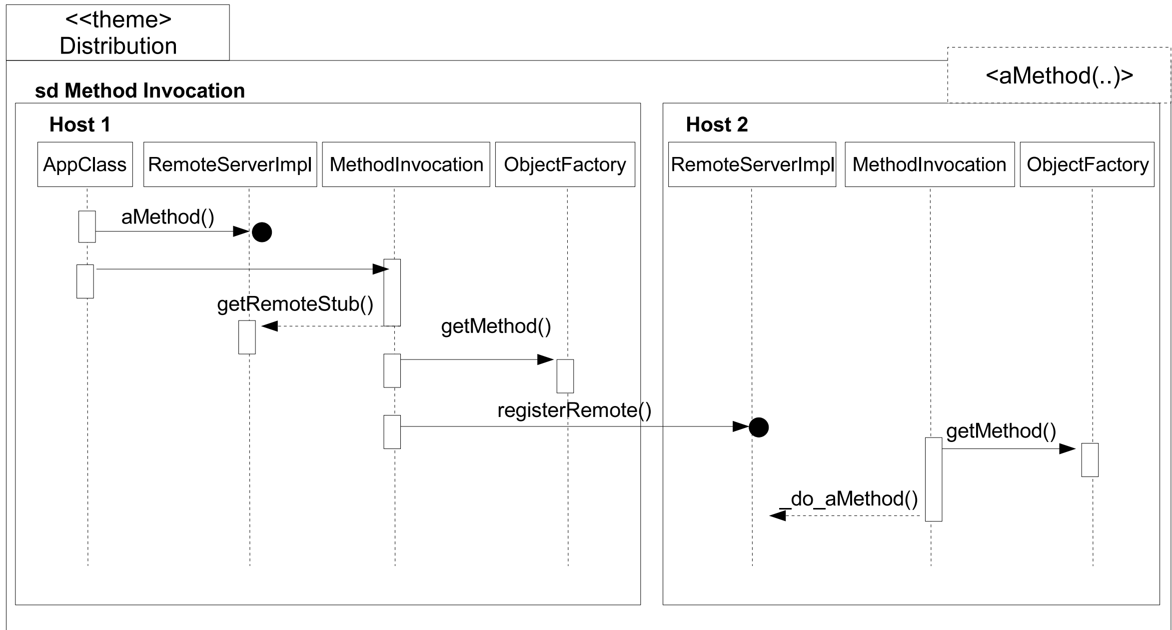

Fig. 4.4: Distribution on Method Invocation

The implementation provides clear modularisation of distribution from base applications. The approach is based on a design previously used to modularise distribution using AOP [59], i.e., the use of an object factory and interceptions on method calls and constructor calls. This approach maintains the obliviousness of base applications to distribution functionality, increasing modularity and thus reducing complexity. The ALPH model provides RMI distribution in its current implementation to examine the reduction in complexity achieved using this approach. The model could be extended to provide a choice of distribution technologies.

4.3.2 Network Roaming

Network Roaming and Quality of Service concerns share a fundamental concept i.e., applications may need to switch network at some point in an application's execution. Quality of Service builds on the functionality provided by the Network Roaming module in the ALPH model. In this section, we focus on Network Roaming while the following section outlines how the Quality of Service module builds on its network switching behaviour.

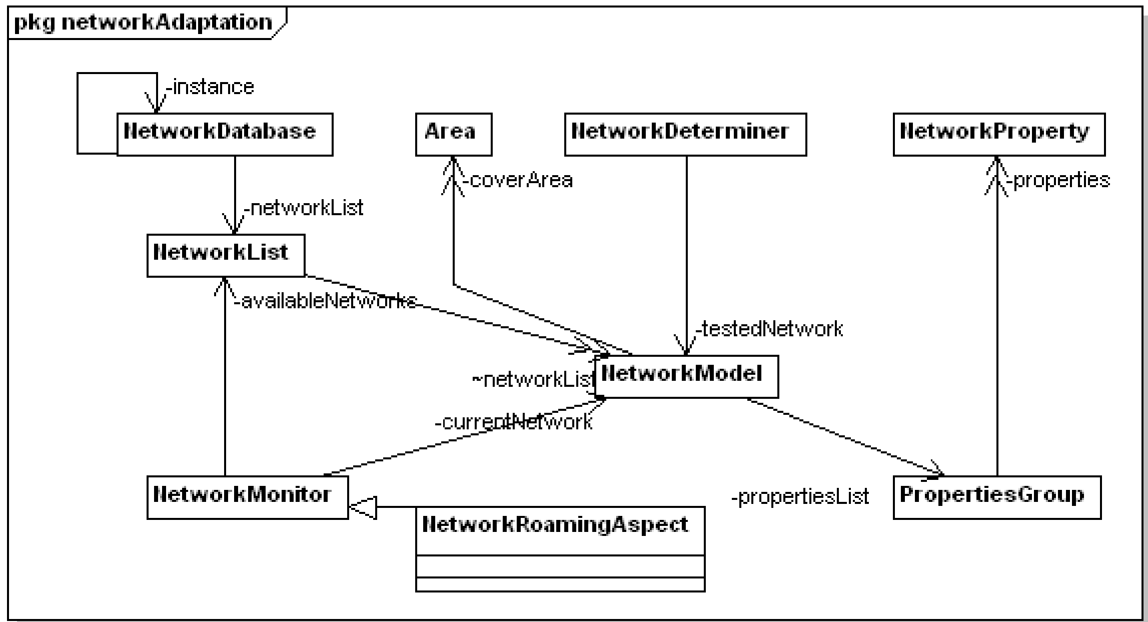

Fig. 4.5: Network Roaming Class Diagram

Because of the unstable nature of wireless networks mobile devices can easily lose their current network connection. However, most mobile devices support a variety of different networks, like IEEE 802.11 and Bluetooth. This module addresses the situation where the network connection the device is currently using is lost or deteriorates, so the device can change the network connection depending on which networks are currently available to the device.

The Network Roaming module consists of three main entities along with helper classes, as illustrated in figure 4.5. The `NetworkModel` class represents a particular type of network and is responsible for acquiring an accurate list of features relating to that network. The `NetworkMonitor` class is responsible for managing the networks and searching for new networks. The `NetworkDatabase` is populated from an XML file of properties of available networks. The `NetworkRoaming` aspect encapsulates crosscutting code which is woven into an application and is triggered whenever a network activity is required.

The Network Roaming module in the ALPH model provides a facility for selecting the

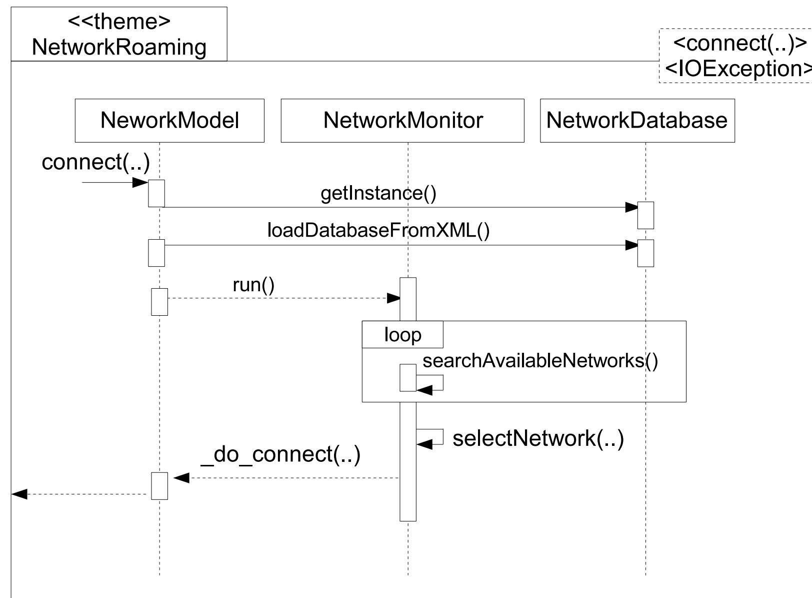

Fig. 4.6: Network Roaming Module

network that the application uses and ensures that an appropriate network is available to the application. The aspect intercepts connections to a network and executes network evaluation to select the most appropriate network, as shown in figure 4.6. The aspect must also intercept any exceptions caused by network failures which in turn invokes the re-evaluation of the characteristics of each known network and any new networks that may be found. The application then selects the best available network over which network communications can proceed before returning control to the method that threw the exception.

The Network Roaming module modularises the described functionality so it can be added to new and existing base applications without the need for refactoring. No work exists modularising network roaming behaviour and so the ALPH implementation is an original design. The current implementation addresses only the outlined functionality. A limitation of the design is the requirement for network properties to be provided by the user in XML format, as shown in listing 4.1. The automation of this process would require an extension of the current ALPH implementation.

```
1 <networkList>
2     <network networkName="wifi1">
```

```
3         <propertiesList>
4                 <property name="NetworkType" value = "1" type ="String"/>
5                 <property name="Latency" value="10" type = "Int"/>
6                 <property name="Congestion" value="5" type="float"/>
7                 <property name="bandwidth" value="1000" type="float"/>
8                 <property name="encryption" value="1" type="boolean"/>
9         </propertiesList>
10        <area xStart = "4" yStart="0" length="2" width="10"/>
11    </network>
12 </networkList>
13 .....
```

Listing 4.1: NetworkPropertyList.xml

4.3.3 Quality of Service

Quality of Service (QoS) controls resources to provide applications or tasks with different priorities [99]. QoS assures certain guarantees regarding network properties to attain a certain level of performance e.g., bandwidth. The ALPH model provides functionality to meet network based QoS assurances. Meeting these assurances requires moving a user to another network when the current network cannot fulfill the required assurances. This function utilises the Network Roaming functionality in the ALPH model, described in the previous section 4.3.2, to find an appropriate network to fulfill QoS assurances.

The QoS aspect uses the Network Roaming module functionality from the library of concerns to facilitate the selection of a network that best suits the network requirements of the application method which is crosscut. Components of the Quality of Service module, shown in figure 4.7, interact with the Network Roaming module, as illustrated in figure 4.8. The `NetworkMonitor` is responsible for providing the list of available networks in the user area. `NetworkDeterminers` are responsible for selecting valid networks from the list provided by the `NetworkMonitor`, i.e., networks that satisfy the quality of service assurances. The `AssuranceReqDB` stores all the assurances defined by the user in a database. This database is populated from configurations in an XML file. Different `Assurance`s can be defined for each task or method call.

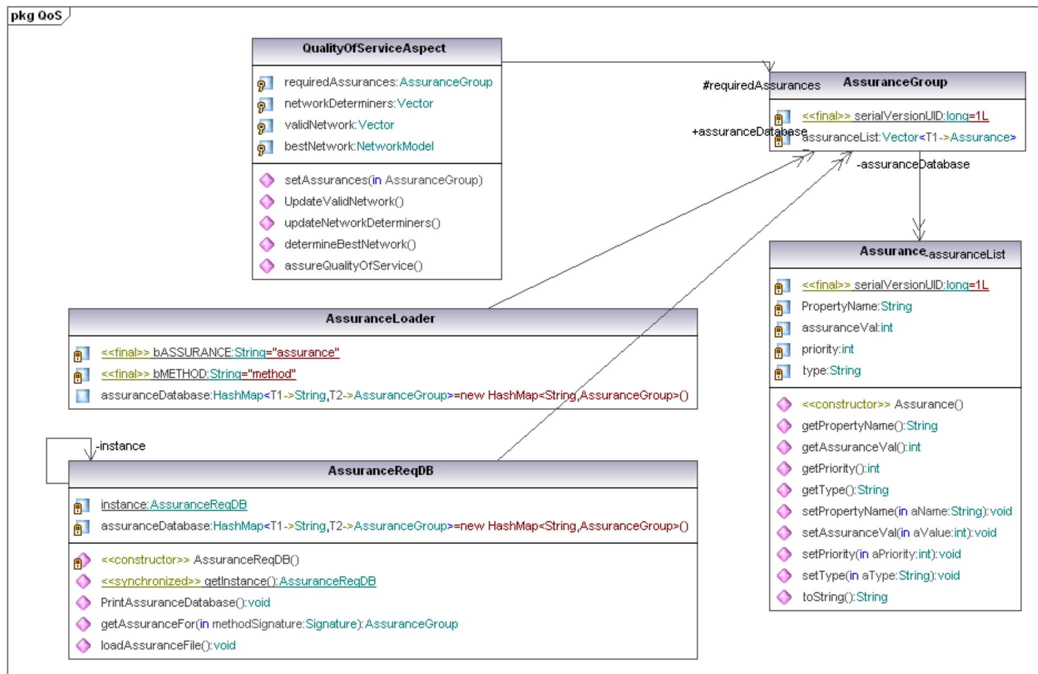

Fig. 4.7: Quality of Service Class Diagram

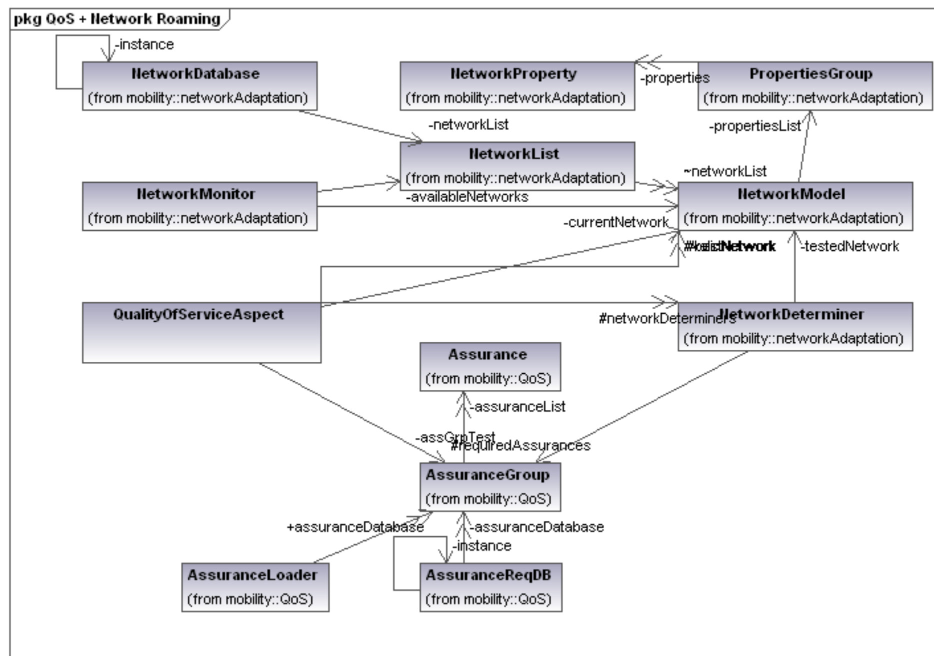

Fig. 4.8: Quality of Service & Network Roaming

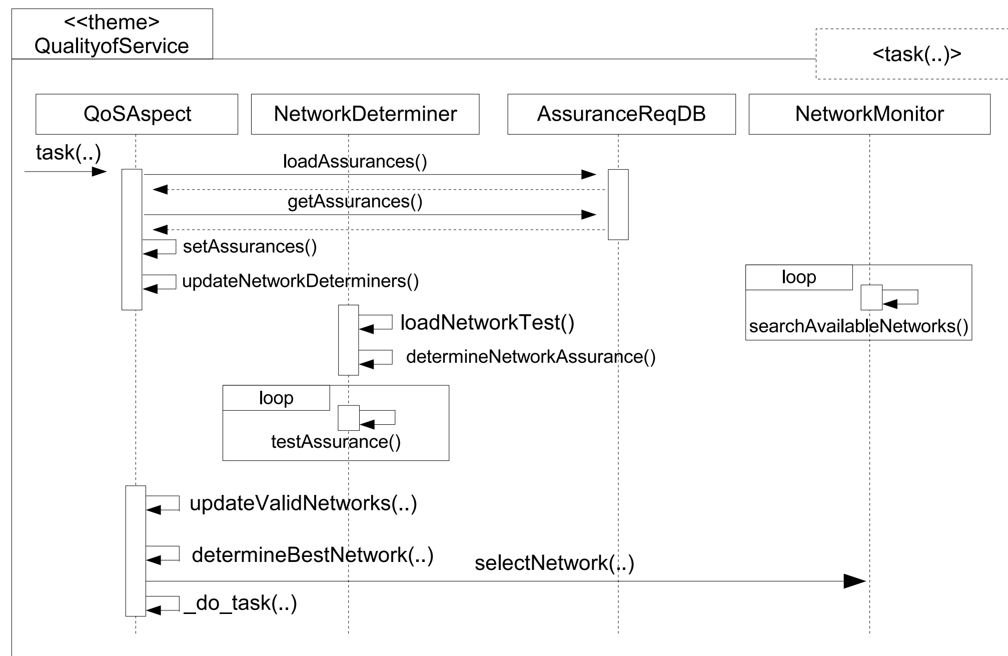

Fig. 4.9: Quality of Service Module

The `NetworkMonitor` continuously updates a list of available networks in the user's area. Simultaneously, the `QualityOfService` aspect checks in the assurance database to acertain if the method needs specific assurances and manages the `NetworkDeterminer`s accordingly. As illustrated in figure 4.9, the `QualityOfService` aspect is triggered on a method call. Acquisition of data regarding available networks in the environment is addressed by the `NetworkDatabase` singleton class where information about networks is loaded from an XML configuration file. Network configuration details include network type, bandwidth and encryption, as illustrated in previously in network romaing, depicted in listing 4.1.

The QoS module, implemented on an original design, separates QoS functionality from base applications using AOP. Unlike existing modularisations of QoS [231], the ALPH implementation does not require the application developer to design interfaces, classes or events for each component. Behaviour is supplied in the aspects and classes in the module. The current implementation of ALPH does, however, require the developer to explicitly provide assurances in an external XML file as shown in listing 4.2.

```
1 <assuranceList>
```

```
2   <method methodName="userPage">
3     <assurance name="bandwidth" value="30" priority="1" type="min"/>
4     <assurance name="congestion" value="20" priority="2" type="max"/>
5     <assurance name="encryption" value="1" priority="3" type="equ"/>
6   </method>
7 </assuranceList>
```

Listing 4.2: AssurancesList.xml

4.3.4 Software Roaming

Software roaming facilitates the migration of code by allowing data or state to relocate from one machine to another [267]. A mobile agent system can be used as an approach to achieve software roaming, as outlined in section 3.6.1.4. Within the ALPH model, software roaming is supported using a mobile agent approach. Mobile agents migrate code to carry out tasks on remote machines. The code is encapsulated within an agent. Client and server applications both require mechanisms for sending and receiving agents. These mechanisms include the serialisation of mobile agent contents, and deserialisation on the receiving host. The agent should also be returned to the original sender after its remote task is complete.

The ALPH model implements a software agent framework that represents the core components of a mobile agent architecture. This provides the basic functionality required to serialise mobile software agents and to transmit them across the network to a remote host. Figure 4.10 shows the components in the ALPH model software roaming module. The `AgentManager` class is responsible for managing the sending and receiving of `MobileAgent` objects. Each host has an `AgentManager`, enabling the application to be run on many interacting hosts or devices. The `AgentManager` implements a remote interface, similarly to the distribution module described in section 4.3.1 and provides functionality to `receiveAgent` and `migrateAgent`. This class is registered with a directory service so that it is available to remote hosts. The two aspects `ClientAgentManagerAspect` and `ServerAgentManagerAspect` are responsible for adding a reference to the `AgentManager` in the client and the server classes. The `MobileAgent` is an abstract class that represent a mobile agent. This class is made the parent of services or tasks that require software roaming. It provides functionality to start and sus-

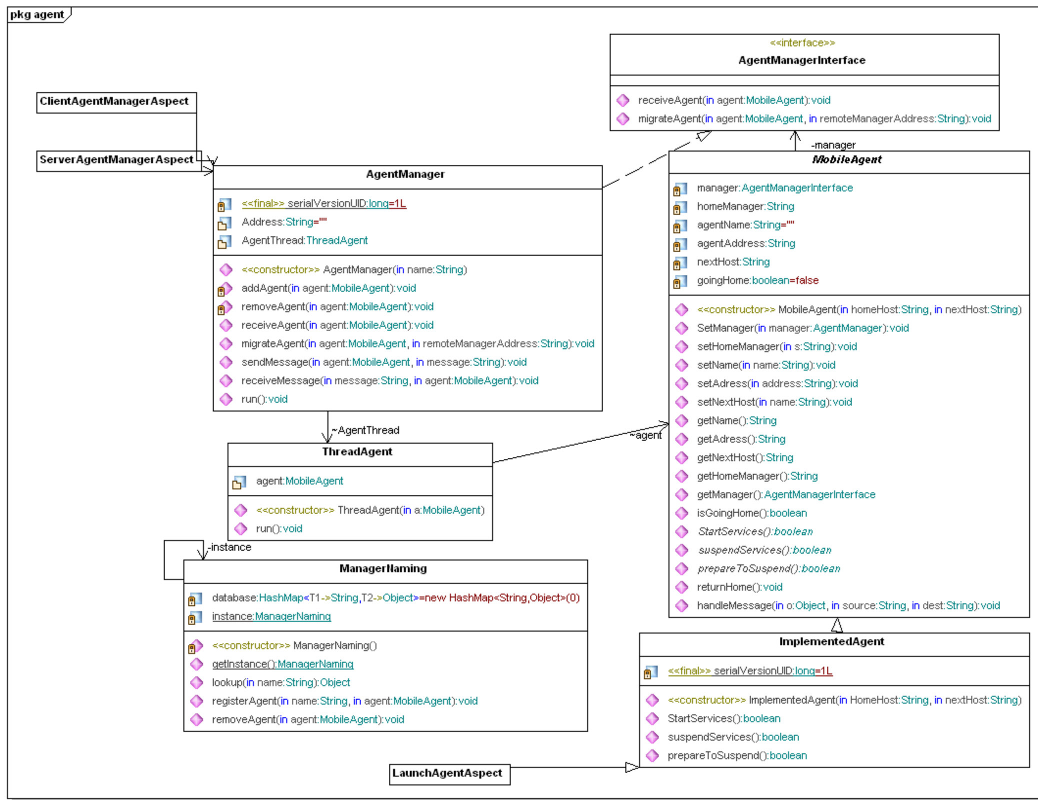

Fig. 4.10: Software Roaming Class Diagram

pend services, as well as enabling the agent to migrate and return to its original host. The `LaunchAgentAspect` creates a new implementation of a mobile agent, `ImplementedAgent`. The aspect intercepts points in the base application's execution defined by the developer and the agent migrates to the remote `AgentManager`, carries out functionality and returns back to the original host.

The theme illustrated in figure 4.11 shows the allocation of an `AgentManager` on agent startup and its registration with a naming or directory service. The migration of a `MobileAgent` during execution is also shown. The software roaming functionality provided by the ALPH model maintains an oblivious base application, encapsulating roaming behaviour in the AOP library. It supports the fundamental behaviour required to migrate tasks using mobile agents. The approach is similar to that of the AspectM framework [170], with functional differences in implementation details.

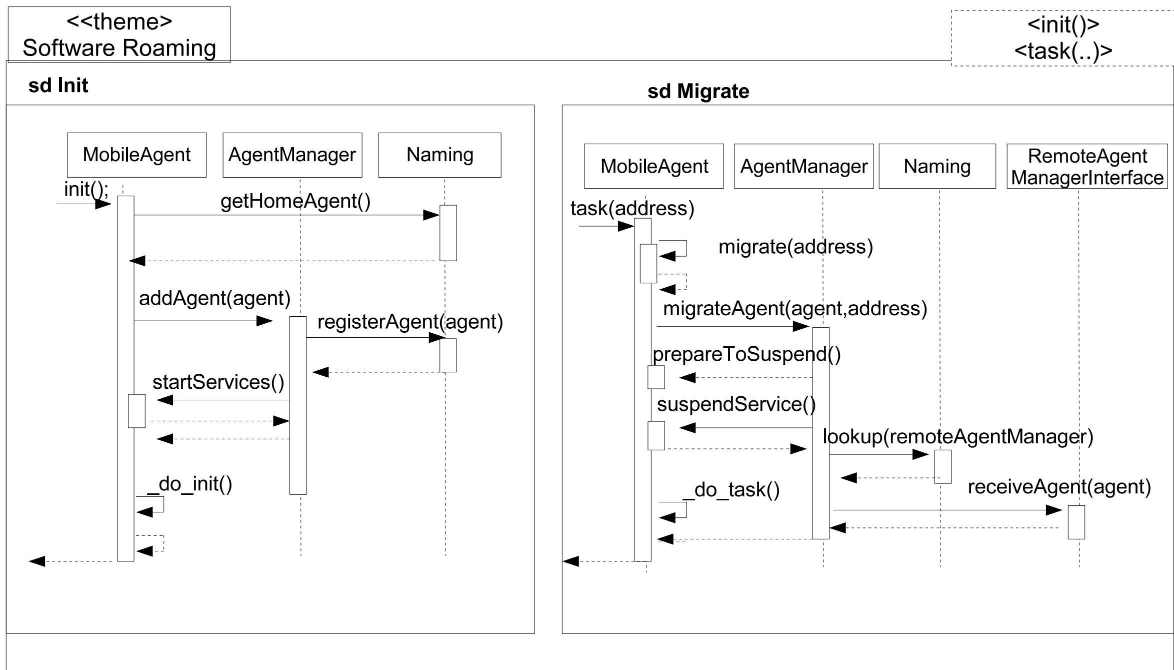

Fig. 4.11: Software Roaming Module

4.3.5 Service Discovery

Applications in mobile environments require the ability to to dynamically discover and acquire access to necessary services [217]. The service discovery module in the ALPH model provides means to discover and acquire access to services by weaving the required methods at appropriate places in the source code [188]. The module provides mechanisms to find suitable services based on the requirements of the application.

A directory service stores information about services and enables lookups to locate required services. Many directory services are available [272] [21] depending on the resources available to the application. While these directory services were investigated, in the current prototypical ALPH model, a more general, lightweight approach was substituted supplying sufficient directory functionality.

The service discovery module consists of an aspect and several auxiliary classes, as shown in figure 4.12. The `DirectoryService` class uses a relational database to retain service information. The `DirectoryService` stores description and location information about available

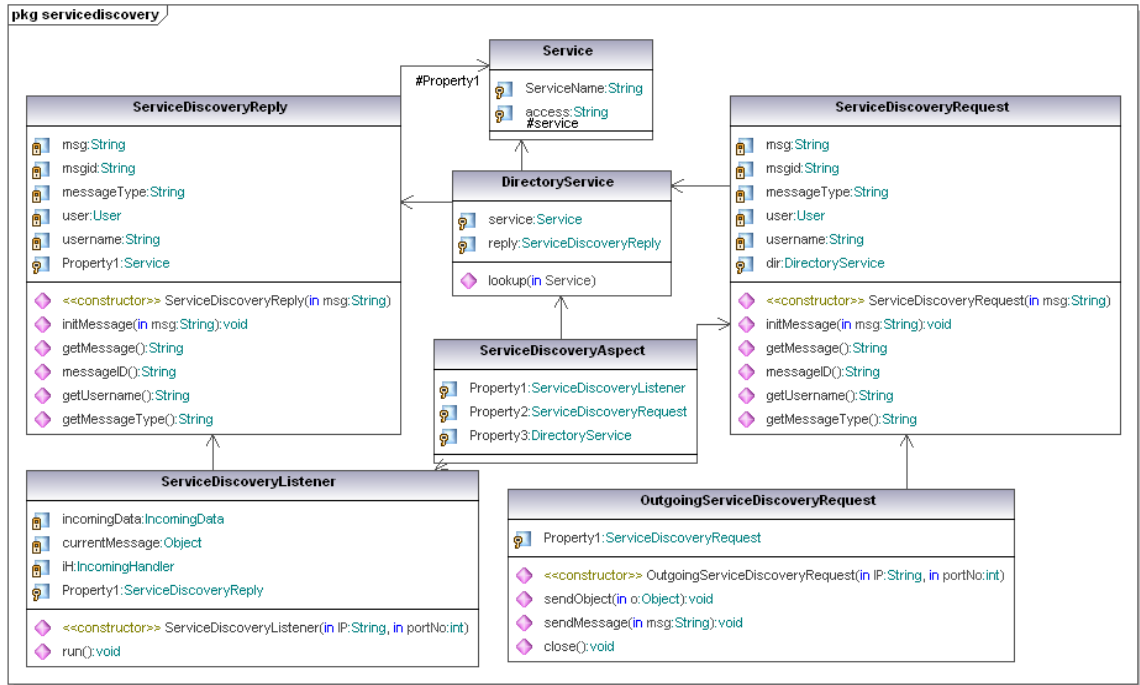

Fig. 4.12: Service Discovery Class Diagram

active services, as well as providing interfaces to allow services themselves to maintain their description entries in the directory service. The `Service` abstract class defines functionality to maintain the state of a service. Services to be advertised are made a subclass of `Service`. ALPH statements define intercept points in the application at which the `ServiceDiscov-eryAspect` injects required service discovery behaviour, as shown in figure 4.13. It initiates a `ServiceDiscoveryRequest` and sends it to the`DirectoryService`. A `ServiceDiscov-eryReply` is sent back to the aspect containing details about the available service. If the service is suitable for use a connection is accepted on behalf of the client. The `ServiceDis-coveryAspect` also enables the updating of service descriptions in the `DirectoryService` when required by base applications.

The ALPH model provides basic service discovery behaviour based on an original design. In future versions, the model could be extended to offer support for various service discovery protocols and more sophisticated directory services. The current implementation modularises the basic discovery functionality, removing crosscutting code from base applications.

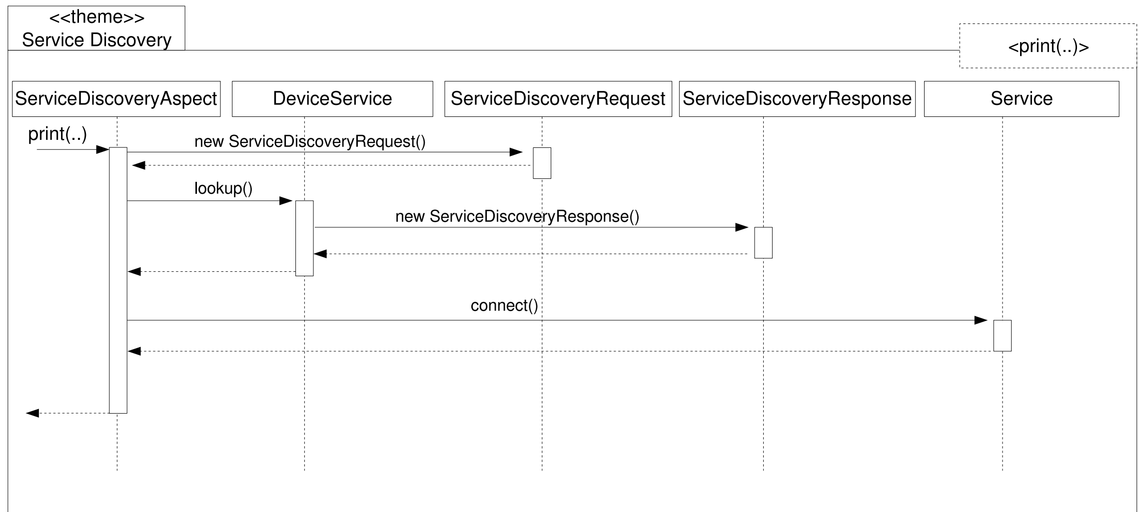

Fig. 4.13: Service Discovery Module

4.3.6 Communication

Applications in mobile, distributed environments require the use of network communication services to communicate with other distributed nodes [227]. Communication mechanisms are essential in pervasive healthcare systems to send and receive data from peers over a network. Underlying network communication implementation details are encapsulated by the ALPH model in the Communication module. Java sockets are used as the underlying transport mechanism. Sockets provide the lowest level network transport available with higher level technologies building on their simple transport facility e.g., RMI and JMS.

A collection of classes and aspects work together to provide methods to transfer data, as well as handlers to manage data transferred, as shown in figure 4.14. The `CommunicationAspect` is responsible for intercepting points in the base application where distributed communication is required, as shown in figure 4.15. The `CommunicationAspect` creates Sockets on both sides of a communication channel, `ClientSocket` and `ServerSocket` with associated `Connection` classes. The `Data` class represents message send or receive entities. Two subclasses, `OutgoingData` and `IncomingData` represent data at either end of the communication channel.

The `Receiver` class is a Thread that runs on each node to listen for incoming data. On

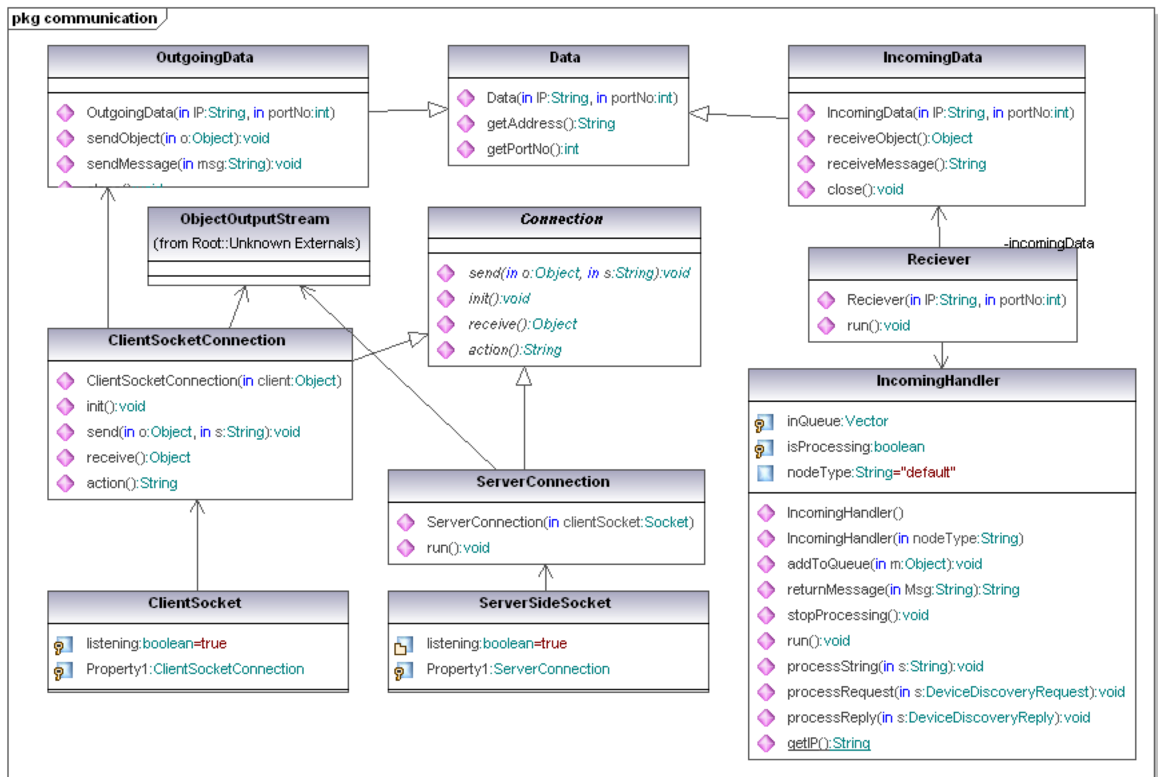

Fig. 4.14: Communication Class Diagram

the setup of this listener, an instance of the `IncomingData` class is created which creates a connection to another node via a socket. An `IncomingHandler` is also created on the setup of a `Receiver` to handle message processing. When data is transferred using the communication module, it is received by the `Receiver` on that node. This data is then passed to the `IncomingHandler` where a vector holds a queue of messages. Its thread constantly processes messages in the queue on the receiver side.

The ALPH model modularises communication functionality outside the base application. The current implementation offers support for socket based communication only. Higher level messaging technologies such as JMS and other communication protocols such as SOAP could also be used for communication functionality. These are not supported in the current ALPH implementation and would require further extension of the model.

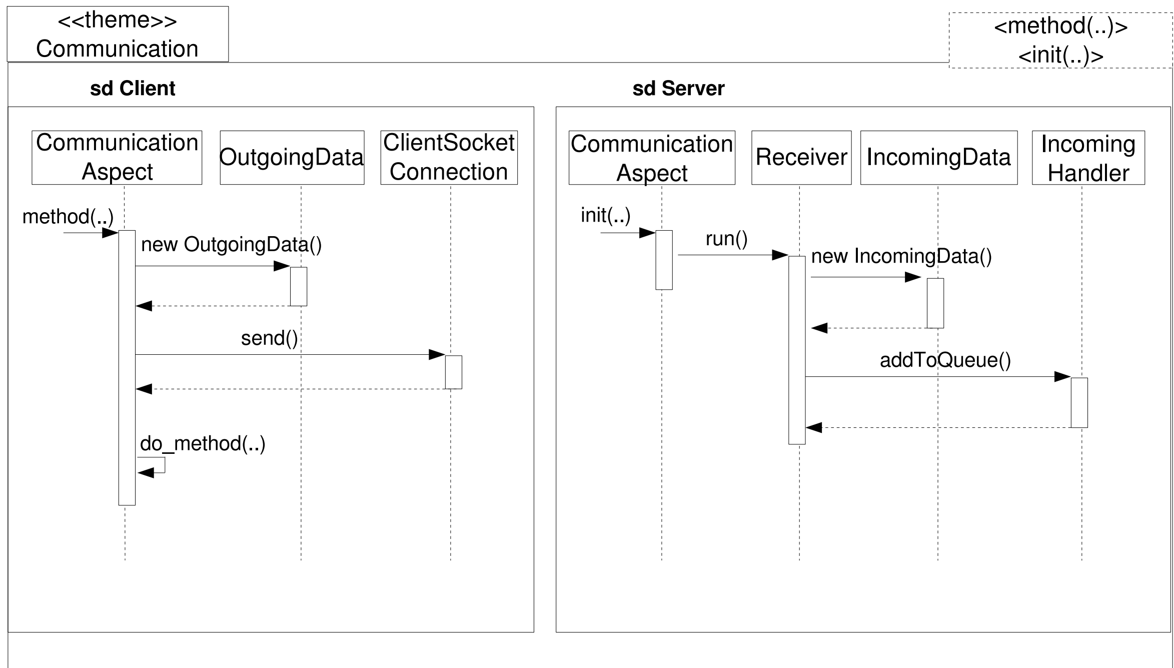

Fig. 4.15: Communication Module

4.3.7 Device Discovery

Devices frequently enter and leave mobile environments. Applications require notifications when new devices enter the environment. The Device Discovery module in the ALPH model adapts the application when new nodes, or devices, become active. The module is based on a multicasting discovery mechanism. The components that make up the device discovery ALPH module are illustrated in figure 4.16.

The `DeviceDiscoveryAspect` is responsible for intercepting the base application at points where new devices become available. A `DeviceDiscoveryListener` is set up for the device, and a `DeviceDiscoveryRequest` is multicasted to a predefined multicast address, as shown in figure 4.17. The device discovery module uses communication functionality from the communication module in the ALPH model to transfer the request to nodes across the network. When a node's `DeviceDiscoveryListener` receives a `DeviceDiscoveryRequest`, the node ensures it is not a self sent request. Requests are handled and a `DeviceDiscoveryReply` is returned to the discovered node containing location information about other nodes. The ALPH

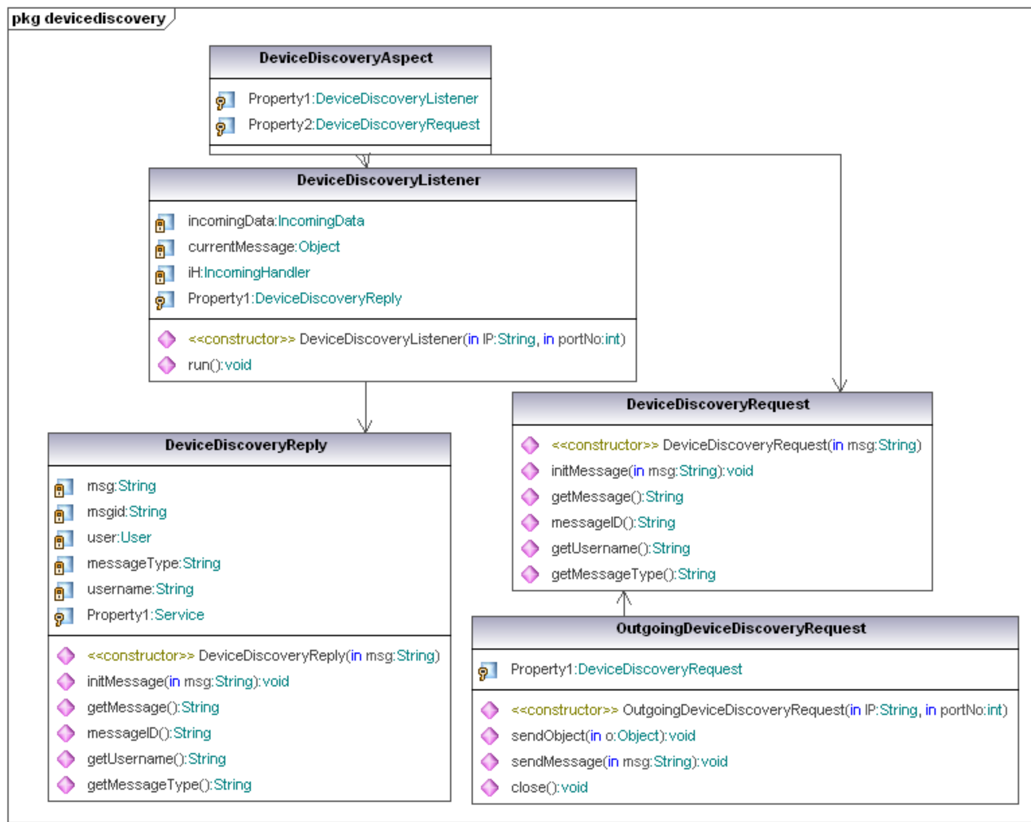

Fig. 4.16: Device Discovery Class Diagram

model also supports the discovery of devices using Bluetooth. The described architecture is used but with Bluetooth protocols over the Bluetooth stack.

ALPH provides modular device discovery functionality using AOP in an original design. The current implementation is based on the Bluetooth device discovery protocol and a multicasting implementation, similar to that of the Simple Service Discovery Protocol (SSDP). Other discovery protocols e.g., Cisco Discovery Protocol (CDP), Universal Plug and Play (UPnP), Link Layer Discovery Protocol (LLDP) could be used but would require extensions to the current ALPH model.

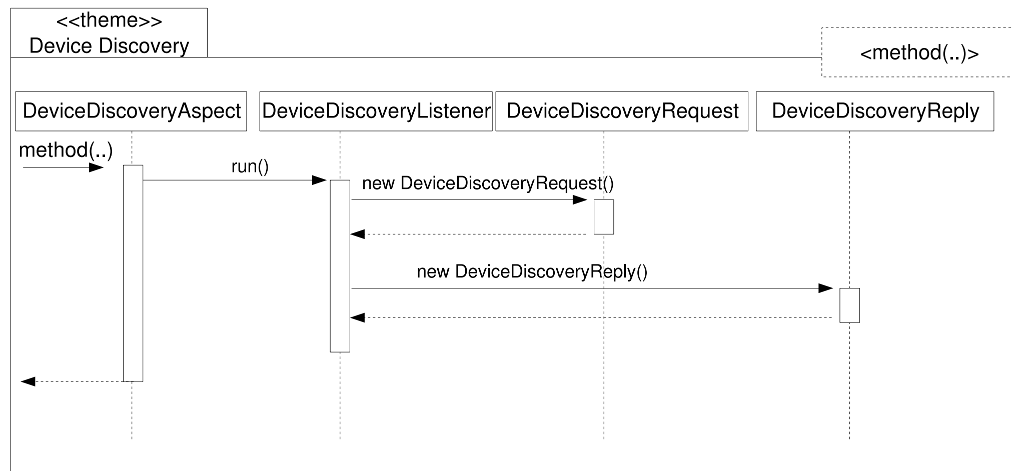

Fig. 4.17: Device Discovery Module

4.3.8 Limited Connectivity

Due to the unstable nature of wireless mobile networks, connectivity is often interrupted by disconnections. ALPH provides two mechanisms for handling limited connectivity, provided as contingency plans. The first is a logging mechanism that maintains application state which can be used to ensure consistency when recovering from a disconnection. The second provides the use of the Network Roaming module to find an available network in the event of a disconnection.

The components in the Limited Connectivity module are shown in figure 4.18. The `ContingencyPlan` class defines methods common to all contingency plans. Concrete subclasses provide specific actions to be taken when handling disconnections. The `LimitedConnectivityAspect` invokes a contingency plan whenever a disconnection is identified, as shown in figure 4.19. The Network Roaming module is utilised in the identification of network disconnections. The `LimitedConnectivityAspect` uses the `LoggingContingencyPlan` class to log application state and activity. This maintains an external log that can be used by contingency plans to ensure application state consistency. This log is managed by the `LogRepository`.

The contingencies are fully modularised in the limited connectivity module in ALPH and can be used with any Java base application. The ALPH model implementation is limited to logging and network contingency plans for limited connectivity events. More concrete con-

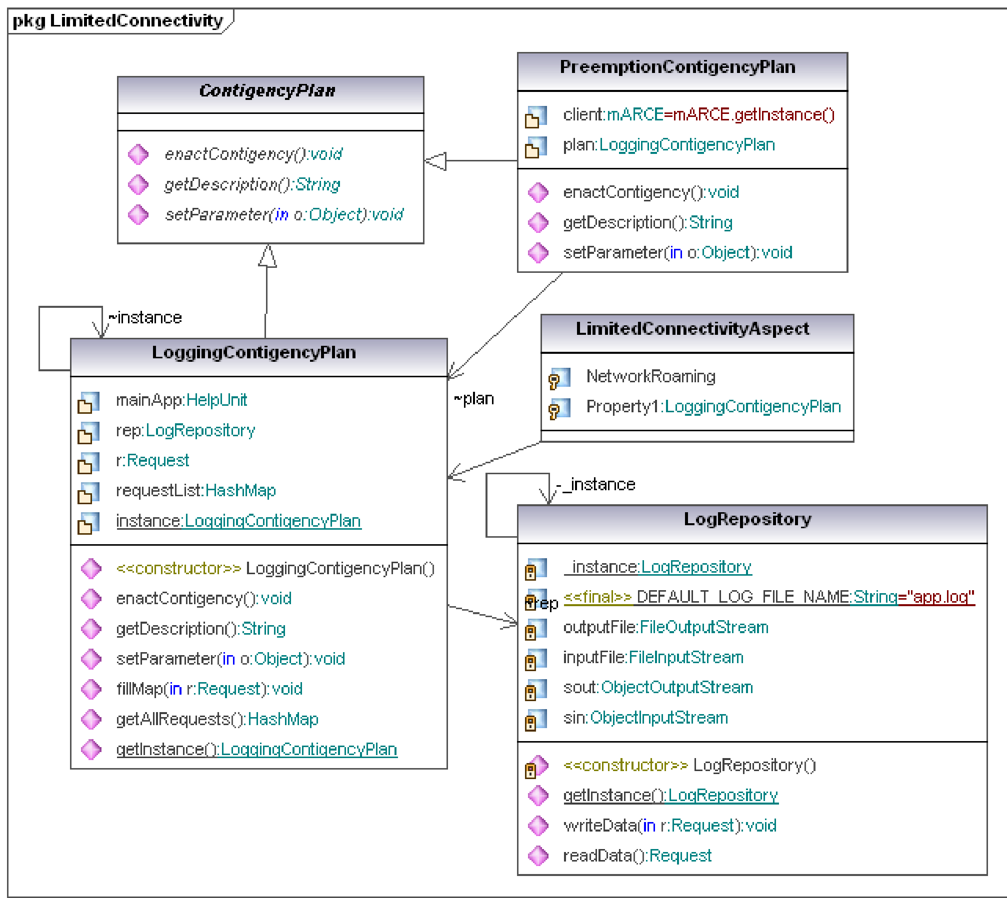

Fig. 4.18: Limited Connectivity Class Diagram

tingency plans could be implemented as extensions to the current implementation, including plans for transactional rollbacks. Transactions are not supported in the current ALPH model. If transactions are necessary, transactions code is explicitly required.

4.3.9 Device Context

Device context information includes device-specific properties such as screen size and storage capacity [229]. This information can be used to adapt applications appropriately to the current device. The Device Context module in the ALPH model encapsulates all concerns dealing with the adaptation of content with respect to the current capabilities of a device e.g., screen size, colour depth, processing power, storage capacity, and bandwidth [229]. This

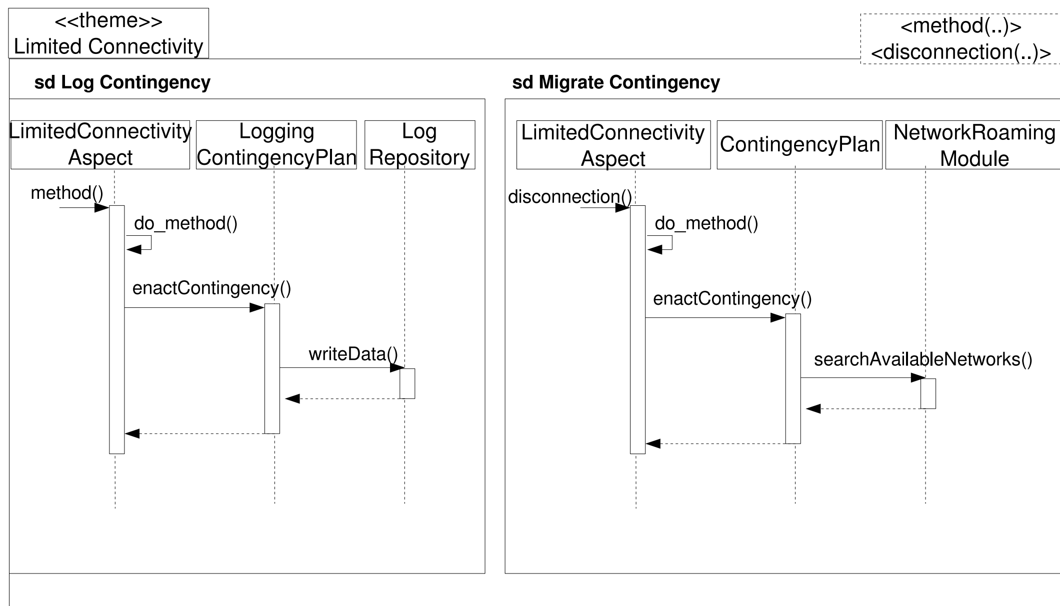

Fig. 4.19: Limited Connectivity Module

information is used to adapt the application to accommodate device capabilities. The overall architecture of the module is depicted in figure 4.20.

The adaptation of content to the current capabilities of the device is encapsulated within the `DeviceAdaptationAspect` aspect and a collection of auxiliary classes. The `DeviceAdaptationAspect` intercepts calls to methods that generate or display content, as shown in figure 4.21. It then obtains the current capabilities of the device from the `DeviceMonitor`, which maintains the state and the capabilities of the device currently in use. The device currently in use is represented by the `Device` class. After having obtained device capabilities, the `DeviceAdaptationAspect` reasons about the contextual information to compute the optimal course of action. The next section describes the reasoning mechanism.

4.3.9.1 Reasoning

The adaptation is realised using artificial intelligence techniques, specifically rule-based reasoning, whereby a set of rules can reason about the device's current characteristics and adapt the application's content prior to sending it to the device. Jess is a Java based rule engine

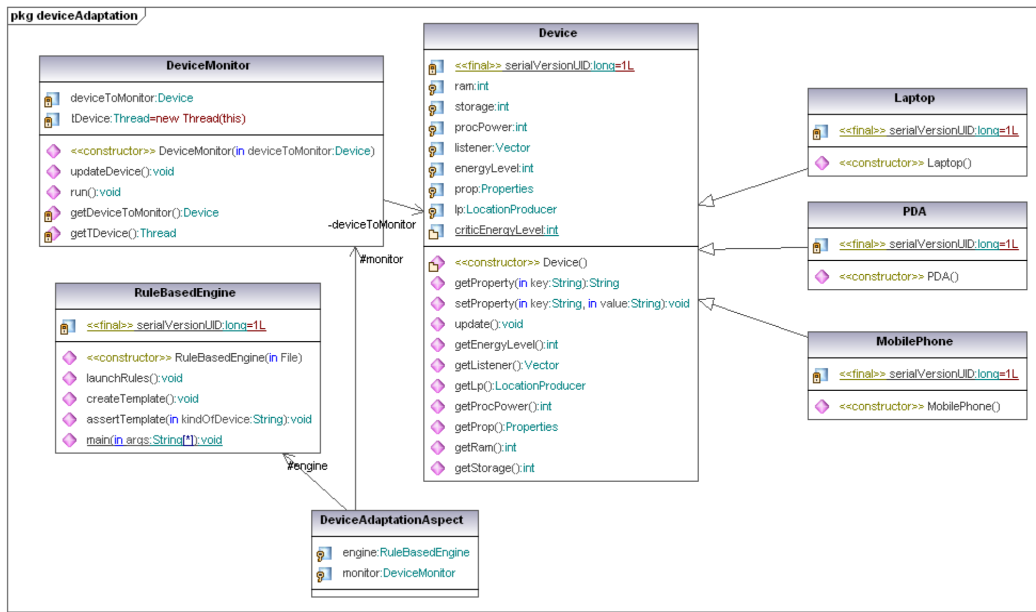

Fig. 4.20: Device Context Class Diagram

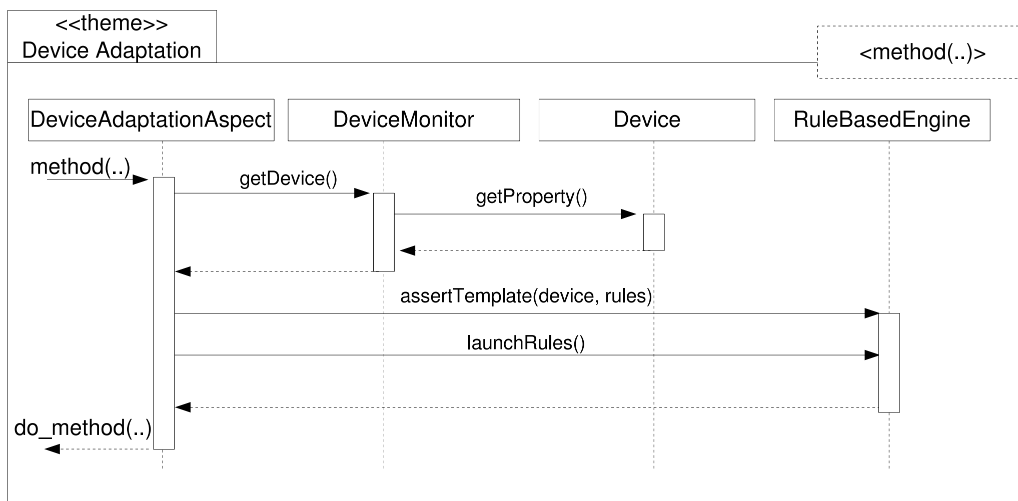

Fig. 4.21: Device Context Module

[1] that uses an enhanced version of the Rete algorithm to process rules [105]. Templates are created to represent a condition or context based on a particular set of rules or facts. The developer must provide a file that contains definitions of Jess templates to adapt the appli-

[1] Jess - http://www.jessrules.com/

cation according to their requirements. A template file is shown in listing 4.3. The rules are evaluated at defined points of execution, and the template is changed if another context is more suitable. The `RuleBasedEngine` class interacts with the Jess framework and sets the templates and rules. The `DeviceAdaptationAspect` uses the `RuleBasedEngine` class to reason about device information and to relate the current context of the device to the rule-based reasoning system.

```
1 ;(deftemplate device (slot name) (slot screenSize))
2 ;(reset)
3 ;(defrule screen-size
4 ;       "Adapt to screen"
5 ;        (device {screenSize < 200})
6 ;        =>
7 ;       (printout t "Adapt to small screen." crlf))
8 ;(assert (device (screenSize 200)))
9 ;(run)
10 ;
```

Listing 4.3: DeviceScreenRules.clp

Device adaptation functionality is modularised into aspects and classes in the library of the ALPH model. The implementation is not based on any related work, but provides similar functionality to existing adaptability aspects [213]. The current ALPH model uses the Jess rule-based engine, and is therefore limited to adaptations in response to rules defined by the developer. Any further behaviour, including set rules for particular devices, would require the extension of the current prototypical ALPH model.

4.3.10 Location

Pervasive applications often use location information to adapt application behaviour to the current context. Location was identified in chapter 3 both independently and encompassed in context-aware adaptation. This ALPH location module provides functionality to obtain location information and provides means to translate between different representations of a location. The module consists of five classes and two aspects.

`LocationProducer` is an abstract class that represents hardware devices to encapsulate

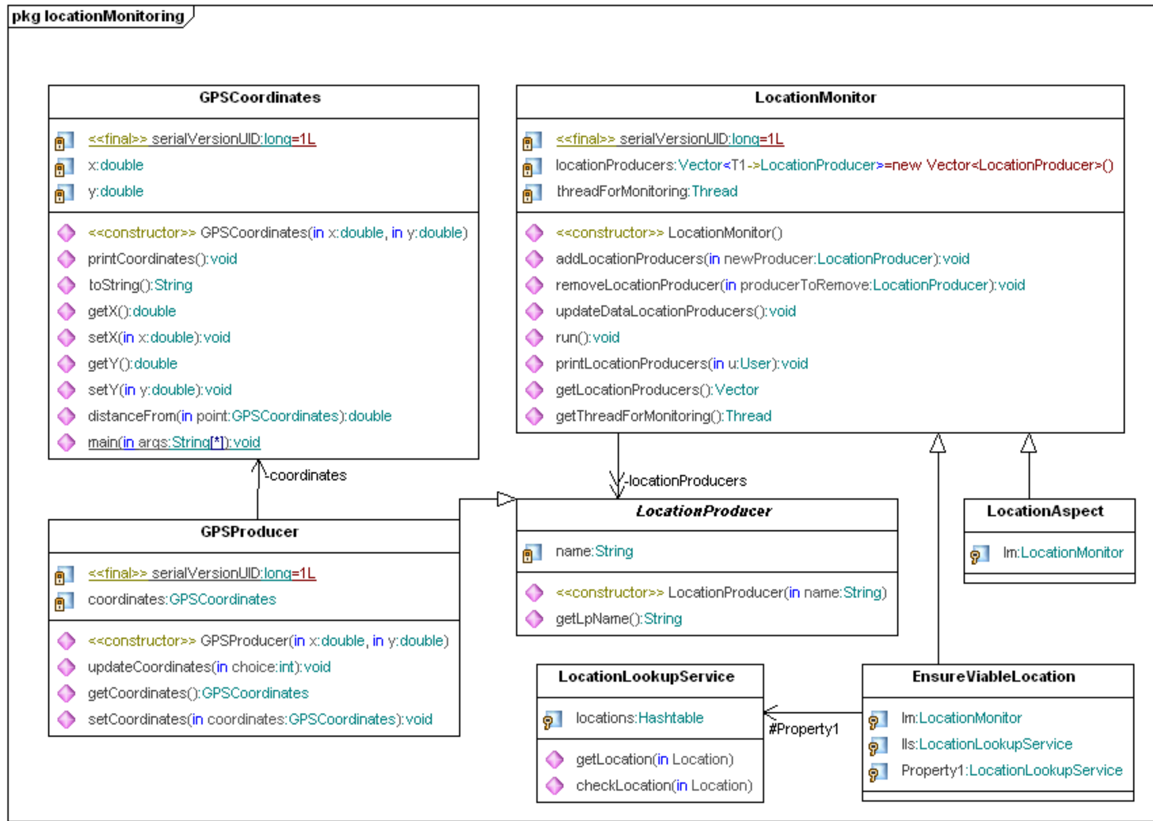

Fig. 4.22: Location Class Diagram

the location data provided by the device. Each type of location device class extends a common `LocationProducer` class. The ALPH model provides an implementation for GPS location information in the `GPSProducer` and `GPSCoordinates` classes. The `LocationMonitor` module is responsible for monitoring all `LocationProducers` and for ensuring that the location information provided is up-to-date. The `LocationAspect` aspect provides behaviour that is executed anywhere within an application where a location is required. It obtains the current location from the `LocationMonitor` class, as shown in figure 4.23.

An additional aspect, `EnsureViableLocation`, is available to carry out checks on current location to ensure that the user is in a valid location for the task to proceed. The `LocationLookupService` class provides methods for translating between the various different formats that location information can be encoded in. This allows the aspect to reason about location

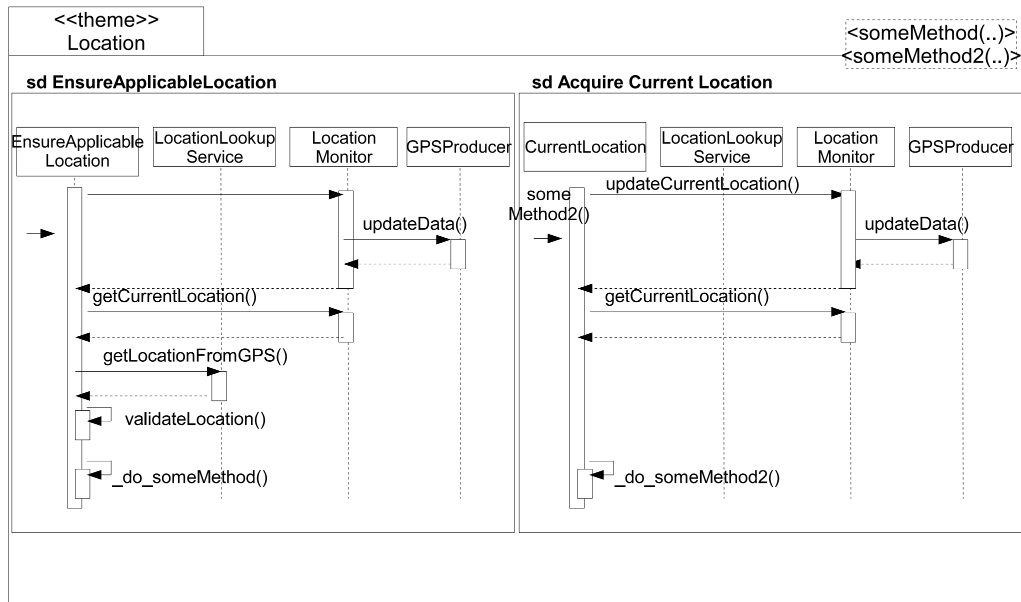

Fig. 4.23: Location Module

data from any form of device, for example the location producers may provide only GPS coordinates whereas the location aspect may be required to filter data based on its origins being within a particular named boundary e.g., facility or area. Explicit locations may also be defined and passed to the application in an external file as shown in listing 4.4. The `EnsureViableLocation` aspect obtains the current location from the `LocationMonitor` class. It translates the location data into a form that can be easily reasoned about, and the current location is queried using the `LocationLookupService` before deciding whether or not it is safe to continue with the execution.

```
1 TCD, 53.342383, −6.250495, 53.34506 −6.259185
2 IRE, 55.337268, −5.516968, 51.442367, −10.230103
```

Listing 4.4: safeLocations.txt

The ALPH model encapsulates location behaviour in a modular way, removing any requirement for references to location monitoring in base applications. The current implementation is limited to GPS location information. The model would require extension to support other location sources, e.g, RFID, mobile cell location information.

4.3.11 HL7

Healthcare information requires standardisation to enable its use in the various systems of healthcare facilities. These standards facilitate the transfer and sharing of healthcare information, enhancing interoperability. The ALPH model includes support for the incorporation of HL7 standards to facilitate this interoperability in healthcare applications. HL7 support in the ALPH model is documented as the modularisation of HL7 healthcare information using AOP [189].

HL7 version 2.4 includes over 70 message types. Many sections of the messages are constant, but each message has particular elements relating to its purpose e.g., an OMD_-O03 message representing a Dietary Order message needs to receive specific information, the most important being the diet order itself. To comprehensively support all message types an existing API can be used in conjunction with AOP. HL7 application programming interface (HAPI) [4] is an open-source, object-oriented HL7 parser for Java and can be used to manage all forms of HL7 version 2.x messages.

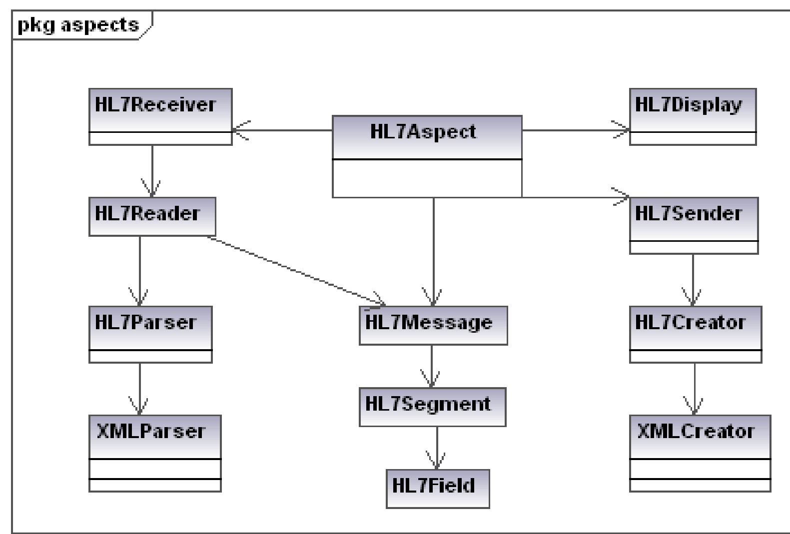

Fig. 4.24: HL7 Class Diagram

The ALPH model HL7 module consists of various helper classes and an aspect to coordinate their activity, as shown in figure 4.24. The `HL7Aspect` intercepts points in the

base application where information should be formatted or where already formatted information requires parsing. The `HL7Aspect` contains calls to the various helper classes to carry out the required HL7 functionality e.g., to create or parse a `HL7Message`, shown in figure 4.25. Healthlink [5] have cooperated in the implementation of the ALPH model. Healthlink is an electronic communications project funded by the Irish Government's Health Service Executive. The project provides a healthcare communications network that links general practitioners, hospitals and healthcare related agencies. Healthlink facilitates data exchange using HL7 version 2.4 in XML format. In collaboration with ALPH, Healthlink have provided templates for a selection of the HL7 messages currently used within the Irish healthcare system. We used these templates to create XML DOM handler classes to create and parse these messages. The ALPH model HL7 module is designed to make use of these handler classes and to introduce the required behaviour into the base application at the appropriate points in execution.

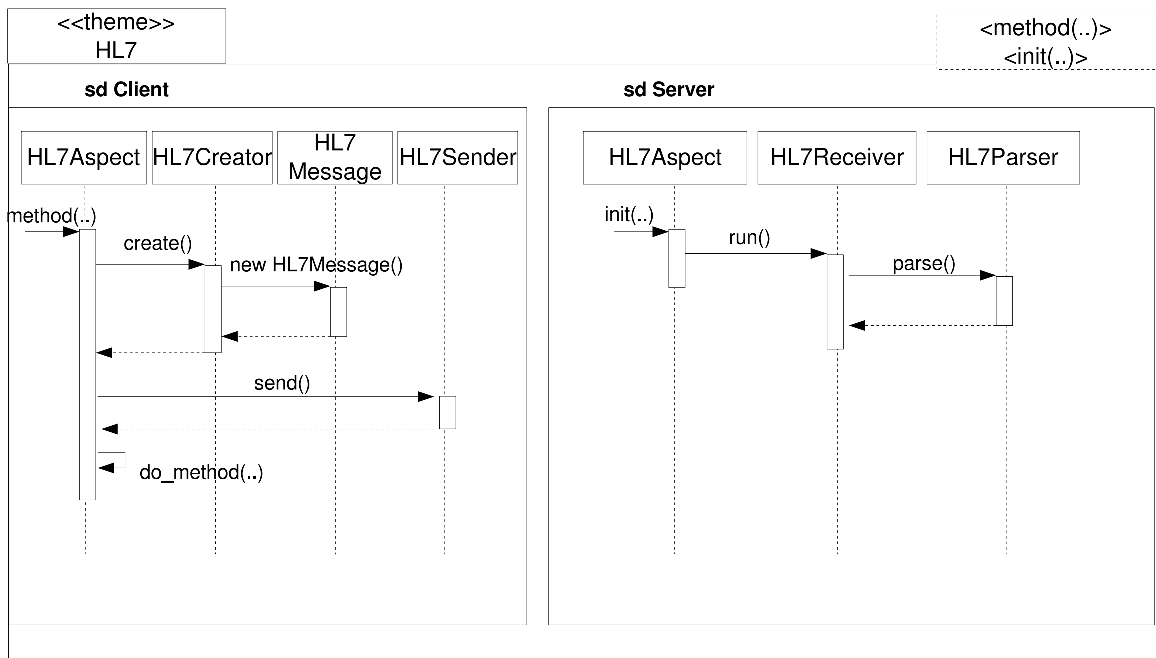

Fig. 4.25: HL7 Module

The ALPH model provides basic create, parse, display, send and receive HL7 functionality

for messages in version 2.x, based on the co-operation of the Healthlink project. It does not support the use of HL7 version 3.0 messages or protocols. In order to fully support HL7 v3 the ALPH model would need to be extended using the HL7 Java SIG Project API [2].

4.3.12 EHR

Electronic records of medical data are complex entities made up of various possible heterogeneous components. A patient record, or EHR, might consist of lab results, prescriptions, personal data and administration information. Standardisation is therefore key in the implementation of an EHR system. Similar to the use of the HL7 standard for the transfer of healthcare information in the ALPH model, an EHR system based on standards is also employed.

Building EHR systems is notoriously difficult [44]. In the migration to EHR systems from paper based systems it has been discovered that over 2000 paper forms are used in the treatment of patients in hospitals [246]. Given the enormity of possible record types, the ALPH model uses an established standard EHR system, openEHR [9]. The openEHR foundation is a non-profit organisation that develops open-source specifications for EHRs in healthcare applications. The openEHR architecture is based on fifteen years of research in numerous projects and standards from around the world. The organisation has worked with the European Committee for Standardization (CEN) to standardise the format of data in EHR systems and ISO/TR 20514.

openEhr applies a two-level modelling approach. The first level is a reference information model that defines generic structures and data types for representing information within an EHR. These structures are illustrated in figure 4.26. Each patient has their own `EHR` with a unique EHR id. An `EHR` encapsulates a set of `Composition` objects where each `Composition` represents information about a particular change to the `EHR` e.g. admission to a hospital or a set of test results. Each `Composition` contains one or many `Entry` objects, each of which represents a single medical statement e.g. a particular medication or result from a specific test. openEhr defines five generic types of `Entry- Administration`, `Observation`, `Evalu-`

[2]HL7 JavaSig http://www.hl7.org/Special/committees/java/index.cfm

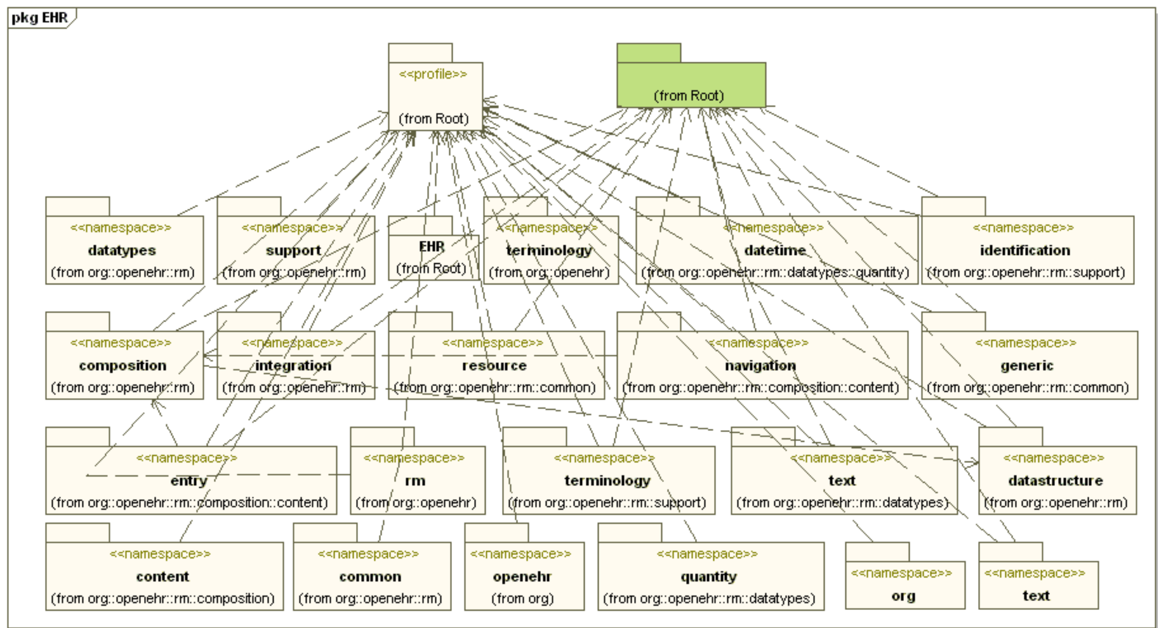

Fig. 4.26: EHR Package Diagram

ation, `Instruction` and `Action`. The second level within the openEhr approach formally defines the structure of clinical content in an archetype model. `Archetypes` are descriptions of valid `Entries` and `Compositions`. For example a blood pressure `Archetype` defines all the information a clinician might want to report about a blood pressure measurement (an observation `Entry`). By formally defining valid structures openEhr enables the sharing of `EHRs` across healthcare systems. In the archetype model `Archetypes` are defined using an Archetype Definition Language (ADL) and edited using an ADL editor, as illustrated in figure 4.27. Each `Archetype` defines constraints on the reference model for a particular `Composition` or `Entry`. For example an `Archetype` might define a set of `Entries` that must be included for a hospital discharge summary.

ALPH uses the openEhr Java reference implementation in its `EHR` module. It provides `Archetypes` for a limited selection of common healthcare information operations based on previous work with the Irish Healthcare Executive Healthlink project [5]. An `EHRAspect` intercepts the base application at specified points of execution, as shown in figure 4.28. The developer specifies the `Archetypes` used to validate the `Entry` and `Composition` to be added

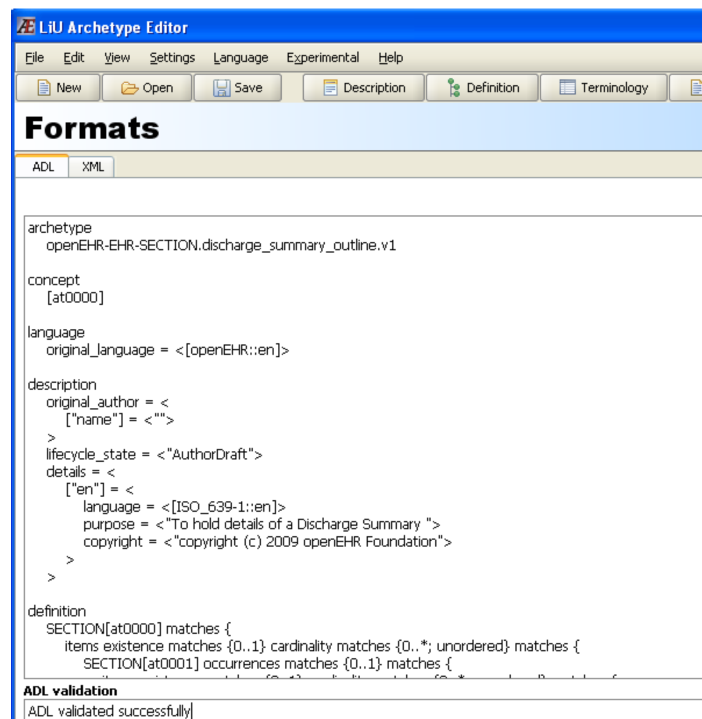

Fig. 4.27: Archetype Definition Language

to the `EHR`. Variable values from the base application are used to instantiate an `Entry`, which is encapsulated in a `Composition`. These structures are validated against the `Archetypes` as they are created. The `Composition` is then added to the `EHR`.

4.3.13 Persistence

Healthcare systems replacing and supporting paper based systems handle large amounts of data. This data must be persisted and accessed frequently in pervasive healthcare applications. The ALPH model provides persistence implementation using the JDBC API. The modular persistence code is encapsulated in a set of aspects and helper classes, as shown in figure 4.29. An abstract aspect defines persistence behaviour in a reusable module. Concrete implementations use application-specific details required for persistence related functionality. We support an object-relational model, where objects are persisted automatically on creation and modification. Persistence requires general tasks to carry out basic persistence operations.

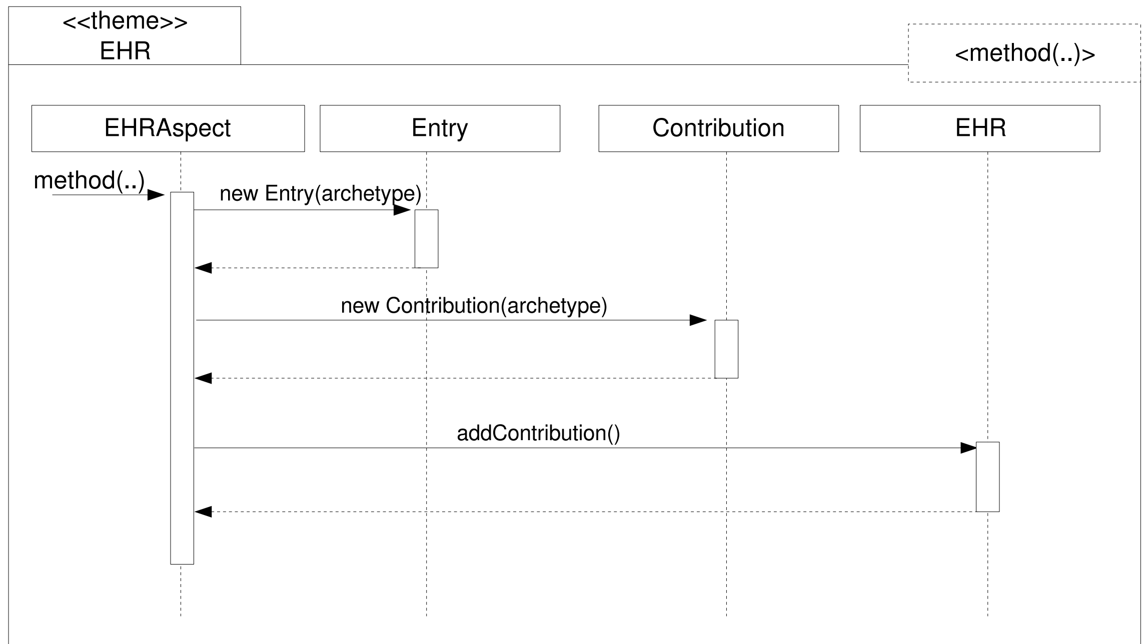

Fig. 4.28: EHR Module

The ALPH model provides functionality for these operations, namely: connection handling and datastore interaction handling i.e., inserting, updating, selecting and deleting database content.

To acquire a connection to a database, the database name and location must be provided. In addition, the driver required to create the connection must be specified. This behaviour should be executed on the initialisation of an application. An `initConnection` pointcut defines the appropriate join point on application startup to trigger the initialisation of the database connection, as shown in figure 4.30. Object-relational based persistence requires that the datastore be updated on the creation or manipulation of each object in application memory. Classes which are to be persisted must be defined explicitly to enable the persistence module to identify them. The `PersistenceManager` aspect compares the class of the object being created to identify if it is to be persisted. If so, the datastore is checked to ensure a table exists to hold its data and creates one if required with a column for each attribute of the class, along with a column to represent the object id. An object id value is introduced by the aspect into each instance persisted to enable its retrieval. The object is persisted by

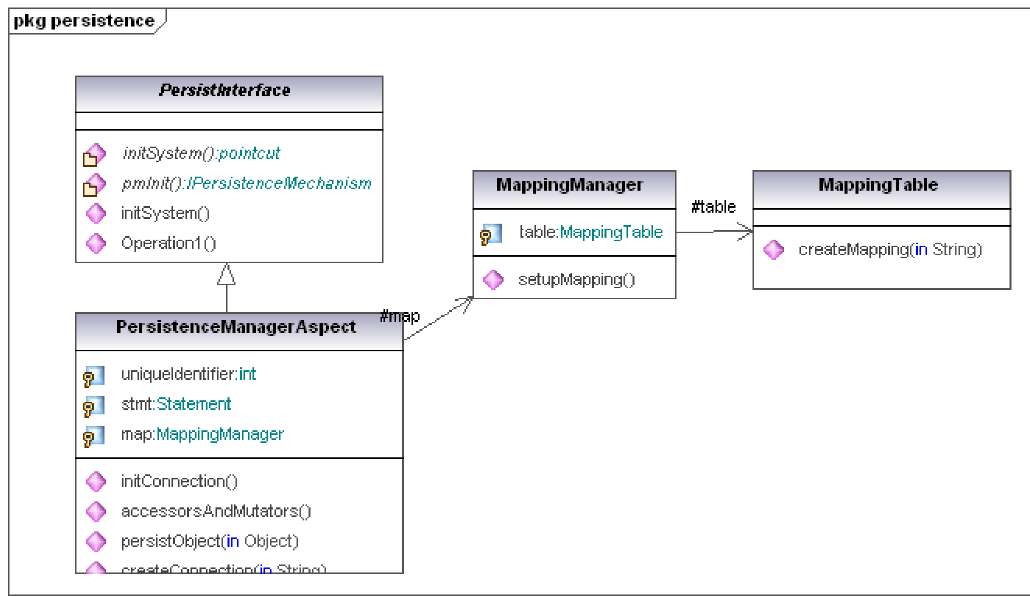

Fig. 4.29: Persistence Class Diagram

using a JDBC SQL insert statement. Further updates, deletes and retrievals are carried out as the `PersistenceManager` aspect intercepts variable mutators.

To address the mismatch between the object model of the object-oriented programming language and the database relational model mappings are required [93]. The ALPH model provides limited prototypical persistence support for object references using object-relational mapping. Mappings are maintained in a table and used to ensure all reachable references are updated or deleted as appropriate. This approach to mapping has been used in previous AOP persistence projects [212] [131]. ALPH reuses this approach, as it provides an architectural solution to mapping in a modular manner, retaining the obliviousness of base applications.

4.4 Summary

This chapter has detailed the modular design and implementation of a library of crosscutting pervasive healthcare concerns in the ALPH model. Modularisation is achieved by using AOP in the implementation of concern functionality, encapsulating concern behaviour in modules

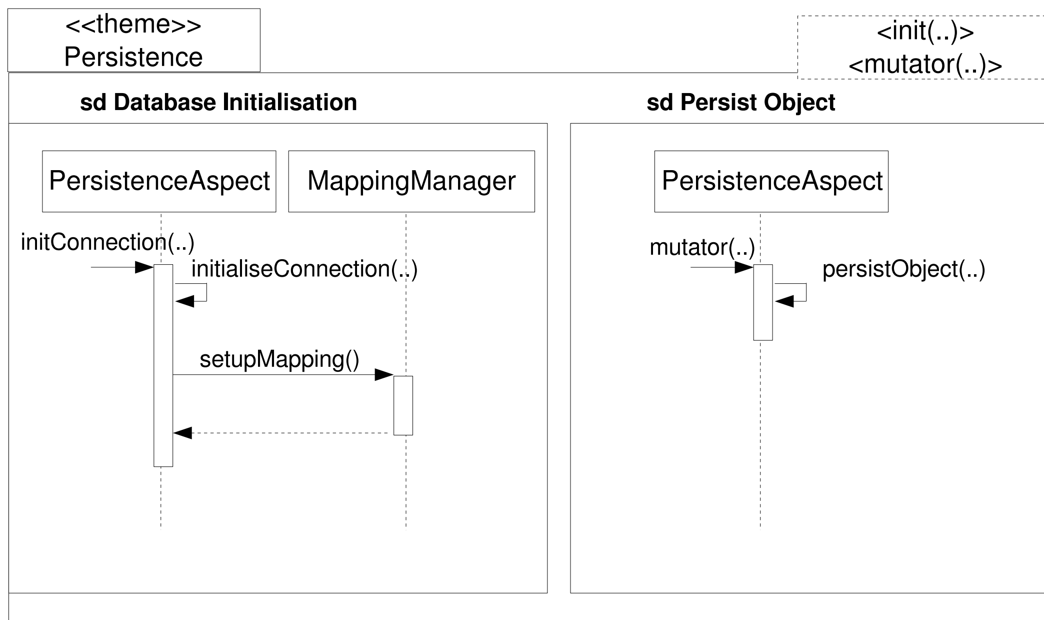

Fig. 4.30: Persistence Module

of aspects and corresponding auxiliary classes. This approach increases the separation of crosscutting concern code from the base applciation logic, in the attempt to reduce complexity in pervasive healthcare applications.

Chapter 5 introduces the DSL in the ALPH model. The ALPH language provides high-level domain-specific constructs that enable the developer to use functionality from the library of concerns. The constructs abstract the developer from the implementation level details discussed in this chapter, reducing the requirement for domain knowledge. An overview of DSL development is outlined, followed by a detailed description of the ALPH language, its compiler and the language processing technique. The domain-specific constructs provided in the ALPH language are also presented.

Chapter 5

ALPH Language

This chapter describes steps 5 and 6 of the methodology followed in the development of ALPH, described in section 3.2. These steps are the creation of a DSL and its associated translator and are depicted in figure 5.1 as part of the ALPH model. An overview of the ALPH language is given before describing the general process of implementing DSLs. The ALPH language compiler and its implementation are described including details on the components developed and language processing that takes place. The language constructs available in ALPH are also outlined.

5.1 Overview

The pervasive healthcare concerns identified in Chapter 3 represent domain-specific functionality that is common and crosscutting in pervasive healthcare applications. Chapter 4 addresses the modularisation of these concerns by compiling a set of modular implementations into a library of pervasive healthcare concern functionality. The goal for design and implementation of these concerns using AOP is to increase modularisation in pervasive healthcare applications, thus reducing complexity. The second approach to reducing complexity in the ALPH model is to provide an appropriate level of abstraction for reasoning about and programming applications in the pervasive healthcare domain.

The ALPH language provides high-level domain-specific constructs that enable the devel-

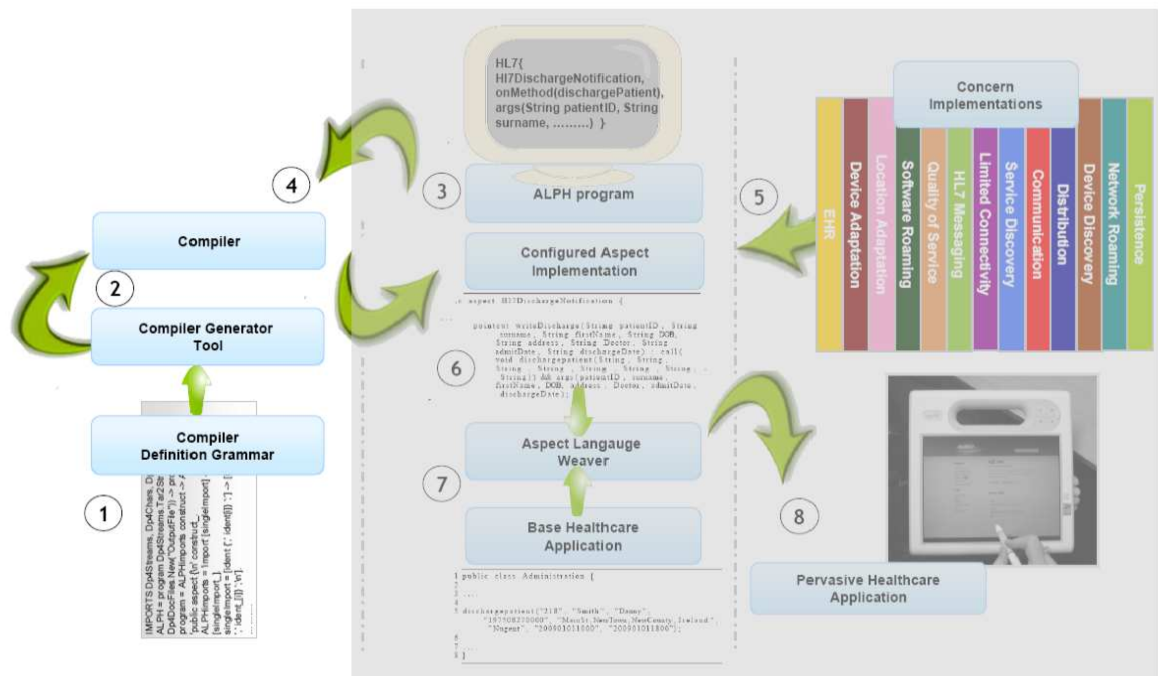

Fig. 5.1: ALPH Language in the ALPH Model

oper to use functionality from the library of concerns described in Chapter 4 in their pervasive healthcare applications. Developers use the ALPH language constructs to define what concerns they want to include in their applications and they provide information to assist the customisation of the aspects generated to introduce pervasive healthcare behaviour into their applications. Using these high-level constructs reduces the requirement for application developers to have extensive knowledge about the concerns they are using. The constructs capture domain concerns and provide abstractions built on the library of pre-implemented concerns, removing the need for developers to implement concerns from scratch. The ALPH language abstracts the developer from the low-level implementation details concealed within the library of concerns. The high-level constructs are also expressive, enabling developers to use semantically intuitive language in the development of domain-specific behaviour.

5.2 DSL Implementation

DSLs can be categorised into two basic categories; ones that build on existing languages and ones that are entirely new languages [179]. Building on an existing language has the advantages of using well known language formats and existing compilers. DSLs can be embedded in GPLs by defining new types and operations in the existing language and by providing a library of domain-specific functionality. This embedding requires the extension of the compiler or interpreter of a GPL to work with new domain-specific constructs. Appropriate domain-specific constructs are usually beyond the limited user-definable notation offered by GPLs, and DSL constructs and abstractions cannot always be mapped in a straightforward manner to methods or objects that can be put in a library [179].

The alternative is to create a new language, which offers more flexibility than when extending an existing GPL [179]. New languages require a means of transforming source code in the DSL language to a target executable language. Language processors, interpreters and compilers perform these transformations. Providing an interpreter or compiler for a DSL allows a custom transformation tailored towards applications in the target domain [81]. An interpreter is a computer program that directly performs instructions written in a programming language. A compiler is also a computer program that reads the DSL source code and translates it to executable code. DSL compilers are also referred to as application generators.

While building a new DSL from scratch has advantages, there is increased difficulty and development effort associated with creating a new programming language [179]. Many tools exist to aid in the development of a DSL. Most of these tools support the description of language transformers and are known as compiler compilers, parser generators, or compiler generators. These tools create parsers, interpreters, or compilers from definitions created by language developers. Some examples of well known tools of this kind include ANTLR [1], Javacc [7], LISA [180], Lex [164] and Yacc [164].

In the ALPH model, a new language is implemented as a declarative DSL for pervasive healthcare. The ALPH language is a preprocessor to an aspect language using an aspect-oriented library to produce target code making it a Domain-Specific Aspect Language (DSAL). The Javacc (Java Compiler Compiler) parser generator tool is used in the

development of the ALPH compiler, as described in section 5.3.2.

5.3 ALPHc Compiler

Compilers translate one language into another. The input program is known as the source language and the output is known as the target language. The ALPH language compiler, ALPHc, translates ALPH programs into AspectJ code. The translation can be described using T-Diagrams [18].

5.3.1 T-Diagrams

Fig. 5.2: T Diagram Concept

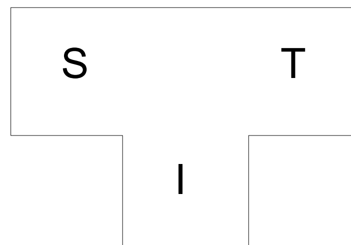

Fig. 5.3: T Diagram

T-Diagrams consist of a set of graphical notations used for describing the transformations provided by language processors i.e., generators, translators and compilers. They are useful in the explanation of the transformation for the source language to the target language. The notation takes the form of a "T" shape with the source language, target language and implementation language forming the shape in the diagram, as shown in figures 5.2 and 5.3.

Figure 5.4 illustrates the ALPH transformation. ALPH programs are the source language

that are translated to AspectJ using ALPHc. The ALPHc compiler was itself transformed using Javacc into target Java representation. These Java objects are then used to transform the ALPH source code into the target language AspectJ. The last step in the compilation process is the weaving of the AspectJ representation into base Java. While this is not strictly a compilation process, we include it as a transformation that takes place as part of language processing within the ALPH model. The purpose of ALPHc is to interpret ALPH programs and to generate intermediate objects used by the ALPH model to include domain-specific pervasive healthcare functionality in base applications.

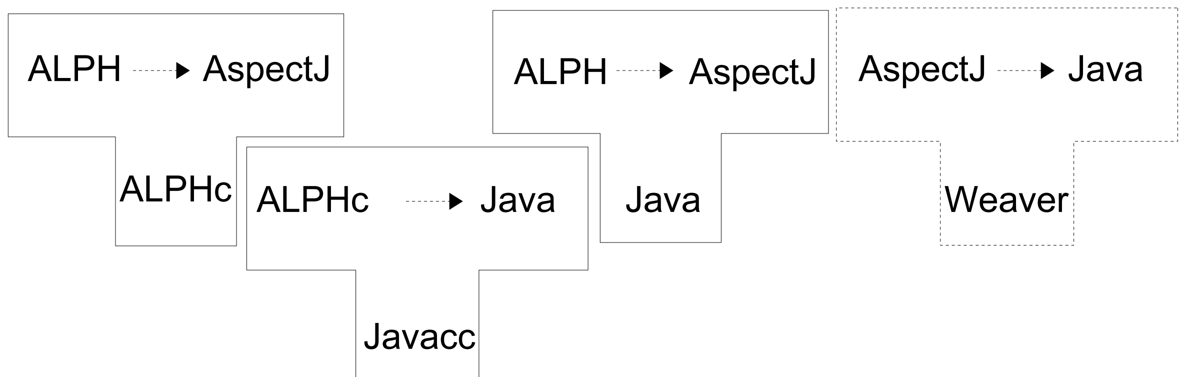

Fig. 5.4: ALPH T Diagram

5.3.2 Compiler Generation

Javacc (Java Compiler Compiler), as mentioned in section 5.2 is a parser generator tool. It helps language developers to create parsers and lexical analysers for programing languages. In particular, it was used in the development of the ALPH compiler, ALPHc. As with all compiler generator tools, the language in question must be defined using a formal language, e.g., BNF [151]. Javacc takes an EBNF, Extended Backus Naur Form [276], specification and uses this to generate the appropriate parser. An EBNF definition describes the production rules for the ALPHc compiler. The grammar definition of the ALPH language from the ALPH model is included in Appendix B.

5.4 Language Processing

A pervasive healthcare programmer will construct an ALPH program using the domain-specific constructs available. This program defines what behaviour is to be included in the base application along with other information required for its inclusion e.g., where in the base application it is appropriate to insert the functionality. The first phase loads and parses the ALPH program and completes lexical and syntactical analysis of the file. The parser produces an abstract syntax tree of the file and passes it to the semantic analyser. The semantic analyser checks the file for errors. The last phase generates an intermediate object representation of the program. This consists of a series of objects that represent aspects for the inclusion of the pervasive healthcare concerns described in Chapter 4, the parameters used to configure the aspects to specific application requirements and the mappings from constructs to target language code.

5.4.1 Components

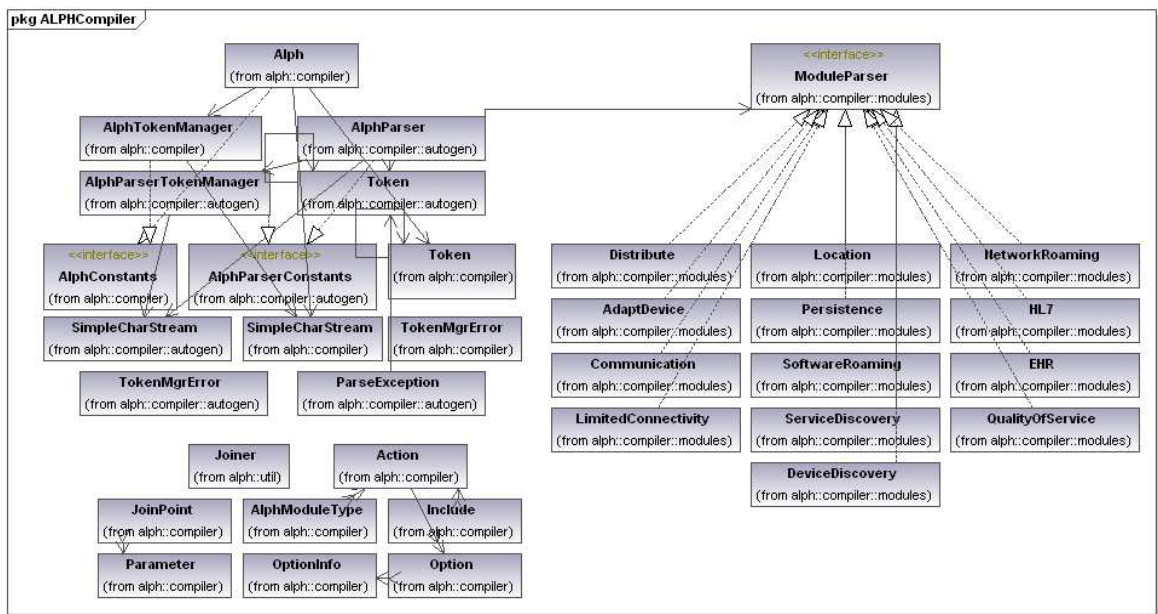

Fig. 5.5: Compiler Class Diagram

Figure 5.5 shows the main components of the ALPH language processor. The entry point to the compiler is the `Alph` class. This class reads the ALPH program from a file using standard input reading and passes it to the `Alph.start()` method. The `start()` method is automatically generated by Javacc from the definition file AlphParser.jj. The contents of the definition file is discussed in section 5.4.3 and included in Appendix B. It parses the file presented on standard input into a List of `Include` objects and returns that list. The `Include` objects are a form of intermediate representation for the ALPH "include" constructs. The mapping between these objects and the code is explained in the next section on parsing. The basic goal of the design of these objects is to allow the parsed ALPH code to be queried and reformatted easily. Each ALPH construct, e.g., HL7, Location, is modelled as a separate class that implements the `ConstructBuilder` interface. This interface defines a way to query the `ConstructBuilder's` name (`getConstructName() : String`) and a way to parse an `Include` statement into Java code (`buildConstruct(Include incl) : String`). Each parsed `Include` statement is iterated through and the appropriate ALPH construct is selected by comparing the `Include` statement's `constructName` with the `constructName` of the available `ConstructBuilder's`, e.g., HL7, Location. If a match is found, the `Include` is passed to the corresponding `buildConstruct` method for that `ConstructBuilder`. Each `ConstructBuilder` expands the `Include` statement into Java code. In the current implementation, a template is filled out inserting the appropriate parameter data from the application-specific ALPH file into the generated aspect template. Figure 5.6 illustrates the sequence of events described for two example constructs, HL7 and Location.

5.4.2 Parsing Overview

The ALPHc parser, `AlphParser`, performs lexical and syntactical analysis on ALPH programs, enabling the compiler to recognise domain-specific constructs and to build up a representation of the source language. This section describes the mappings between the ALPH program defined by the application developer to the intermediate representation objects in the ALPHc compiler. Lexical analysis is carried out by the `AlphParserTokenManager`. It builds up a collection of `Tokens` from the input source ALPH program. Each `Token` repre-

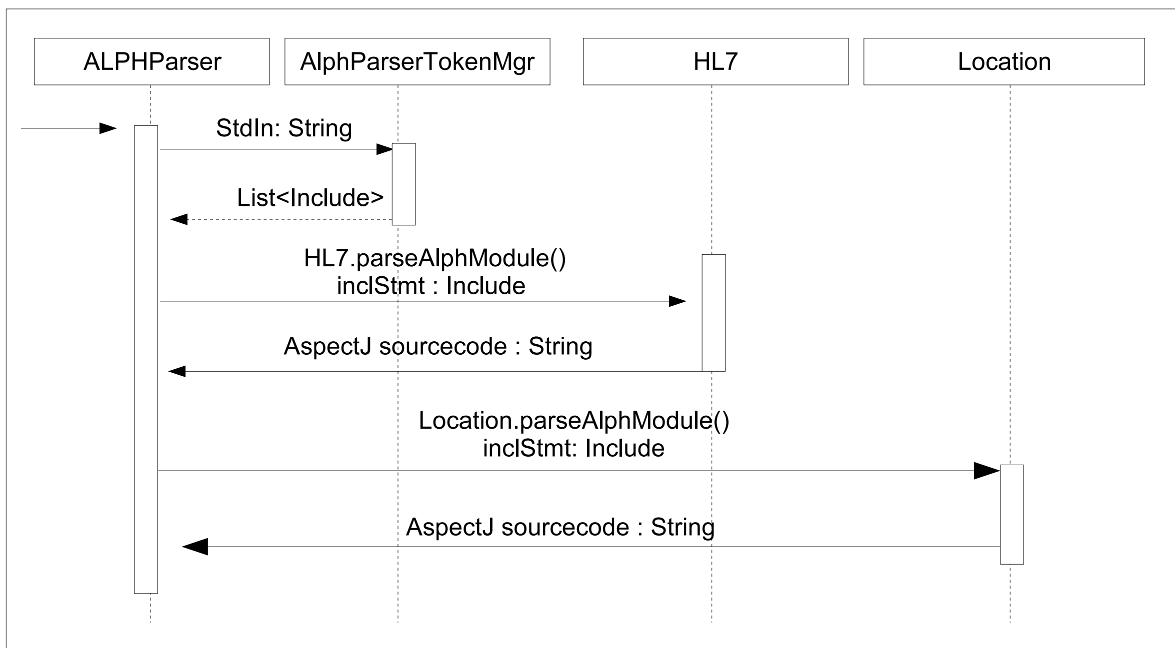

Fig. 5.6: Compiler Sequence Diagram

sents a token kind, each of which must have been defined in the original language grammar. This process validates input e.g., keywords and their format.

```
{-Include------------------------------------------------------------------------}
    {-Action-----------------------------------------------------------------}
                         {-Option-----} {-Option-------------------}

include <moduleTypeName> (<actionName>) @ [<optionInfo>] [<optionInfo>, <optionInfo>]
```

Fig. 5.7: Compiler Mapping

The parser, `AlphParser`, maps statements formed from the domain-specific constructs available in ALPH to `Include` objects. The statements are particular to each concern, or module. The mapping of syntax to object is illustrated in figure 5.7. The “include” statement maps to the `Include` object. Within the `Include` object is an `Action` and a number of `Options`, mapping to the statement in figure 5.7. Syntactical analysis is performed by the `AlphParser`. It parses the list of tokens resulting from the lexical analysis, building a parse

tree on iteration. The `AlphParserConstants` interface associates token classes with symbolic names e.g., "include" or "semicolon". Boilerplate Javacc files `SimpleCharStream`, `Token`, `TokenMgrError` and `ParseException` help the parsing process by providing representation for the stream on input characters, single input tokens and the exceptions thrown when input doesn't conform the the parser's grammar.

`Include`, `Action`, `Option` and `OptionInfo` objects represent the definition in the ALPH program. `OptionInfo` is an abstract class that can currently be one of three subtypes:

- A String (e.g., ADT_A01)
- A Parameter (e.g., User Application.currentUser)
- A JoinPoint (e.g., void App.setCurrentUser (User u))

Classes also exist to represent these subtypes i.e., `JoinPoint`, `Parameter`, `OptionInfoS-tring`.

5.4.3 Definition

The `AlphParser` is defined using EBNF. The steps, or rules, defined outline the valid syntax and semantics of the ALPH language. The definition file is annotated with snippets of Java code to inform the parser of the required output, as shown in Appendix B. The ALPH definition instructs Javacc to create the components described in section 5.4.1 and to create the parsing behaviour described in section 5.4.2. Each rule outlines valid constructs and a valid format for their use. `Includes` is a list of `Include` statements. `Include` is the token "include" followed by a construct name and a list of actions, ended by a semicolon. `Actions` is a comma delimited list of `Action` statements. `Action` is an optional `ActionName` surrounded by parenthesis, the token "@", and a list of `Options`. `Options` is a list of space separated `Option` statements. `Option` is enclosed by braces and consists of one or more `OptionInfo` statements. `OptionInfo` is one of either a single identifier, a `Parameter` or a `JoinPoint`.

5.5 ALPH Language Constructs

The pervasive healthcare concerns identified in Chapter 3 were implemented and moduarised as described in Chapter 4. The implementations described in section 4.3 provide modular components that perform the behaviour required by each module addressed in the library of pervasive healthcare concerns. The ALPH language constructs abstract application developers from the implementation in those components and provide

5.5.1 Terminology

The ALPH language terminology is based on the library of concerns described in section 4.3. Collections of vocabularies in the healthcare informatics domain were also examined to investigate domain terminology. SNOMED CT (Systematized Nomenclature of Medicine-Clinical Terms) [244] and ICF (International Classification of Functioning, Disability and Health) [11] are the most widely used international vocabulary standards for medical applications. To achieve full standardisation, the standards would have to be considered in the entire application placing constraints on the base application implementation. As this is not suitable for the ALPH model, standards are integrated within the limits of maintaining base application obliviousness. Both EHR and HL7 concerns in the ALPH model include standard formats of healthcare information. The EHR in the ALPH model is built on SNOMED CT and is interoperable with HL7 v2.

5.5.2 Constructs

Each concern from the library of modular pervasive healthcare concerns described in Chapter 4 has an associated construct. The constructs form the core of an ALPH program, with a set of "include" statements defining the inclusion of any one particular concern. The statements follow the pattern shown in figure 5.7. That general format is customised for each concern according to the parameters and actions required. The statements outline what concerns to include, passing in application-specific information to enable the ALPHc compiler to generate an application-specific aspect implementation that will incorporate the

required pervasive healthcare behaviour from the library of concerns in the base application. The following sections describe the statements and constructs as they are provided in the ALPH language, and the resulting generated application-specific aspect implementation. The resulting generated output is included in Appendix A.

5.5.2.1 Distribute

The use of the `distribute` construct generates three aspects; `ConstructorInvocationAspect`, `ClientCallInterceptionAspect` and `ServerCallInterceptionAspect`. These aspects are customised using the Objects passed as parameters to the distribute construct. Currently RMI is the only supported technology.

The general format of the `distribute` construct statement is:

- `include distribute (Technology) @ [objectsToDistribute];`

A concrete example of a statement including distribution of a particular object using RMI in an application is as follows:

- `include distribute (RMI) @ [ServerImpl];`

5.5.2.2 AdaptDevice

The `adaptDevice` construct includes adaptation to device context information in applications. A `DeviceAdaptationAspect` is generated to intercept application behaviour at the points in execution specified by the developer. Classes from the device context module in the library of concerns are configured to carry out device adaptation e.g., `DeviceMonitor`, `Device`, and contextual information is reasoned about using a rule based engine according to rules provided by the developer. Adaptation uses a rule-based reasoning engine requiring developers to provide a file containing application specific rules to adapt the application according to their requirements. A template file is shown in listing 4.3 in section 4.3.9.1.

The general format of the `adaptDevice` construct statement is:

- `include adaptDevice (RulesForDevicesFile) @ [EvaluateRulesJoinpoints];`

A concrete example of a statement including the adaptation of an application to particular devices is as follows:

- `include adaptDevice (rules.clp) @ [void App.setCurrentUser (User u)];`

5.5.2.3 HL7

The `HL7` construct generates a configured `HL7Aspect` that creates, parses, displays, sends and/or receives healthcare information in applications. The parameters specify what functionality to include, and the points in application where this behaviour should occur.

The general format of the `HL7` construct statement is:

- `include HL7 (create) @ [HL7CreateJoinpoint, MessageType], (parse) @ [HL7ParseJoinpoints] , (display) @ [HL7DisplayJoinpoints], (send) @ [HL7SendJoinpoints], (receive) @ [HL7ReceiveJoinpoints];`

A concrete example of a HL7 statement to create a HL7 message in an application is as follows:

- `include HL7 (create) @ [void App.dischargepatient (String patientID, String surname, String firstName, String DOB, String address, String Doctor, String admitDate, String dischargeDate), ADT_A01 ];`

5.5.2.4 Location

A `LocationAspect` is generated when the `location` construct is used in an ALPH program. The construct takes parameters that define location format, any high-level locations that are to be inferred from location data, and the points in the execution of the base application where location monitoring behaviour should be applied. An external file including application-specific locations is illustrated in listing 4.4 in the description of the implementation details of the location module in section 4.3.10.

The general format of the `location` construct statement is:

- `include location (LocationType) @ [DeviceInitJoinpoint] [JoinpointsRequiringLocation] [HighLevelLocationInformation];`

A concrete example of a statement to include location monitoring in an application is as follows:

- `include location (GPS) @ [Device App.setDevice()] [void locationMethod()] [safeLocations.txt];`

5.5.2.5 SoftwareRoaming

The `softwareRoaming` construct takes parameters to describe the points at both client and server sides where agent roaming should occur. The `LaunchAgentAspect`, `ServerAgentManagerAspect` and `ClientAgentManagerAspect` handle the application-specific calls to the `AgentManager` to handle roaming activity. In the current ALPH implementation, to migrate an agent to a remote node requires the specification of network addresses. In an extended implementation, a collection of remote nodes could be maintained, removing this requirement.

The general format of the `softwareRoaming` construct statement is:

- `include softwareRoaming (DirectoryService) @ [InitJoinpoint] [ServerJoinpoint, ServerObjectName] [ClientJoinpoints];`

A concrete example of a statement to include software roaming in an application is as follows:

- `include softwareRoaming (RMIRegistry) @ [void User.userPage(..)] [ RemoteServerImpl.new(..), RemoteServerImpl] [void App.welcome(..)];`

5.5.2.6 NetworkRoaming

The use of the `networkRoaming` construct generates the creation of a configured `NetworkRoamingAspect` that uses the `NetworkModel`, `NetworkMonitor`, `NetworkDeterminer` and other helper classes from the NetworkRoaming module in the library of pervasive healthcare concerns. Section 4.3.2 includes an example of a network configuration file in XML format, as shown in listing 4.1.

The general format of the `networkRoaming` construct statement is:

- `include networkRoaming (NetworkConfigurationFile) @ [NetworkJoinpoints];`

A concrete example of a statement to include network roaming in an application is as follows:

- `include networkRoaming (NetworkPropList.xml) @ [User Application.currentUser(..)];`

5.5.2.7 Persist

Persistence is included in an application using the `persist` construct in the ALPH language. A `persistenceManagerAspect` is generated to create connections to the application-specific datastore as defined by the developer in the construct parameters.

The general format of the `persist` construct statement is:

- `include persist (datastoreNameLocation, driverType) @ [startingApplication-Point] [objectsToPersist];`

A concrete example of a statement to persist and object in an application is as follows:

- `include persist (MyDatabase, sun.jdbc.odbc.JbdcOdbcDriver) @ [Hospital.run()] [Doctor];`

5.5.2.8 QualityOfService

Using the `qualityOfService` construct generates a `QualityofServiceAspect` that tailors the relevant classes from the library of concerns using the application-specific parameter values supplied by the application developer. An example of an assurances file that is passed as a parameter is illustrated in listing 4.2 in section 4.3.3.

The general format of the `qualityOfService` construct statement is:

- `include qualityOfService (AssurancesFile) @ [QoSJoinpoints];`

A concrete example of a statement to in an application is as follows:

- `include qualityOfService (NetworkPropertyList.xml) @ [* User.*Page(..)];`

5.5.2.9 ServiceDiscovery

The `serviceDiscovery` construct takes parameters to advertise and discover services in pervasive healthcare applications. A `ServiceDiscoveryAspect` is generated using the application-specific parameters to configure the service discovery module in the library of concerns.

The general format of the `ServiceDiscovery` construct statement is:

- `include serviceDiscovery () @ [Action][ServiceName][ServiceDiscoveryJoinpoints() ||ServiceAdvertisementJoinpoints()];`

Concrete examples of statements to discover and advertise a service in an application are as follows:

- `include serviceDiscovery () @ [advertise][PRINTING_SERVICE][ Printer.new(..)];`
- `include serviceDiscovery () @ [discover][PRINTING_SERVICE][void print*(..)];`

5.5.2.10 DeviceDiscovery

The `deviceDiscovery` construct generates a `DeviceDiscoveryAspect` that configures an application-specific `DeviceDiscoveryListener` where specified in the pervasive healthcare application to carry out device discovery functionality.

The general format of the `deviceDiscovery` construct statement is:

- `include deviceDiscovery () @ [DeviceDiscoveryJoinpoints];`

A concrete example of a DeviceDiscovery statement to discover a device in an application is as follows:

- `include deviceDiscovery () @ [Device.new(..)];`

5.5.2.11 LimitedConnectivity

A `LimitedConnectivityAspect` enacts contingency plans at specified points in the developer's application by using the parameters passed to the `limitedConnectivity` construct.

The general format of the `limitedConnectivity` construct statement is:

- `include limitedConnectivity () @ [Action] [LimitedConnectivityJoinPoints];`

A concrete example of a statement to employ a LimitedConnectivity contingency plan, logging is the default plan, in an application is as follows:

- `include limitedConnectivity () @ [log] [* *.*(..)];`

5.5.2.12 EHR

The `EHR` construct generates an `EHRAspect` that creates `EHR` records where specified in the base application. Parameters specifiy these points and provide access to the data to be entered in the `EHR` records.

The general format of the `EHR` construct statement is:

- include EHR (Action) [EntryArchetype] [CompositionArchetype] @ [EHRJoinPoints];

A concrete example of statement that includes an EHR record at an appropriate point in an application is as follows:

- include EHR (Create) [openEHR-EHR-OBSERVATION.laboratory-glucose.v1][openEHR-EHR-COMPOSITION.encounter.v1] @ [void App.glucoseReading (String patientID, String Doctor, String data)];

5.5.2.13 Communication

A `CommunicationAspect` is generated by using the `communication` construct. The aspect creates communication channels on nodes and configures the corresponding `Connection`, `Data` and `IncomingHandler` classes from the communication module in the library of concerns.

The general format of the `communication` construct statement is:

- include communication (Server |Client) @ [Technology] [IP] [CommunicationJoinPoints];

Concrete examples of statement that includes Communication over Sockets in an application are as follows:

- include communication (Server) @ [Sockets] [249.353.142.87) [void App.init(..)];
- include communication (Client) @ [Sockets] [Device.new(..)];

5.5.3 Parameter Values

The parameters in each construct are outlined and described in figure 5.8, along with a range of allowable values for each construct parameter.

5.6 Summary

This chapter described the DSL provided in the ALPH model. DSLs reduce complexity by enabling developers to programme at a high-level of abstraction in their target domain. The ALPH language provides abstractions for pervasive healthcare application developers. These abstractions, in the form of domain-specific constructs, generate aspect-oriented modular components that include domain-specific functionality in base pervasive healthcare applications. ALPH programs are written using provided domain-specific constructs along with application-specific data provided by the developer through parameters. The ALPH language compiler, ALPHc, parses ALPH programs and generates AspectJ application-specific implementations. The architecture of the ALPHc compiler and the parsing process is detailed, and the constructs available to the developer are described. The description of the ALPH language concludes the presentation of the ALPH model. The next chapter, Chapter 6, describes the final step of the ALPH model creation methodology to implement and evaluate applications built using ALPH. The impact of using ALPH in reducing the complexity of pervasive healthcare applications is evaluated through the examination of five different application implementations.

CONSTRUCT	PARAMS	DESCRIPTION	RANGE OF ALLOWABLE VALUES
Distribute	(Technology)	The technology used in distribution	RMI, Sockets
	(objectToDistribute)*	Object to be distributed	Any object in the base application
AdaptDevice	(RulesForDeviceFile)	Set of developer defined rules to define device adaptation behaviour. Used by rule-based engine.	Any valid .clp Jess template file
	(EvaluateRulesJoinpoint)*	A point in execution where device rules should be evaluated	Any AspectJ joinpoint
HL7	(action)	The HL7 behaviour to be carried out	Create, Parse, Display, Send, Receive
	(AssociatedJoinpoint)*	A point in execution where the specified HL7 behaviour should be applied	Any AspectJ joinpoint
Location	(LocationType)	The form of location data to be used	GPS
	(DeviceInitJoinpoint)	A point in execution when a device is initialised	Any AspectJ joinpoint
	(JoinpointsRequiringLocation)*	A point in execution where location behaviour should be applied	Any AspectJ joinpoint
	(HighLevelLocationInformation)	Information about higher-level locations i.e., mathing coordinates to known locations	A .txt file with String location names and coordinate values
SoftwareRoaming	(DirectoryService)	A directory service to manage references to available remote hosts	RMIRegistry
	(InitJoinpoint)	The point in execution where a device is initialised	Any AspectJ joinpoint
	(ServerJoinpoint)	Point in execution where server agent manager behaviour should occur	Any AspectJ joinpoint
	(ServerObjectName)	Remote Object Name	A valid object name
	(ClientJoinpoint)*	A point in execution where clients require software roaming behaviour	Any AspectJ joinpoint
NetworkRoaming	(NetworConfigurationFile)	Developer defined file with network configuration details of available networks	XML file defining a root NetworkList containing networks and their properties
	(NetworkJoinpoint)*	A point in execution where network roaming should occur	Any AspectJ joinpoint
Persist	(DataStoreNameLocation)	Name and location of the datastore to be used	String URL
	(DriverType)	Type of driver to be used to connect to the specified datastore	sun.jdbc.odbc.JbdcOdbcDriver
	(StartingApplicationPoint)	Point in execution where the Application starts	Any AspectJ joinpoint
	(ObjectToPersist)*	Object to be persisted	Any object in the base application
QualityOfService	(AssurancesFile)	Developer defined file with network assurances defined for each method call requiring QoS	XML file defining assurances for methods
	(QoSJoinpoint)*	A point in execution where QoS assurances are required	Any AspectJ joinpoint
ServiceDiscovery	(action)	The service discovery behaviour to be carried out	Discover or Advertise
	(ServiceName)	The name of the service in question	Any valid String service name
	(ServiceDiscoveryJoinpoint)*	A point in execution where service discovery behaviour is required	Any AspectJ joinpoint
	(ServiceAdvertiseJoinpoint)*	A point in execution where a service should be advertised	Any AspectJ joinpoint
DeviceDiscovery	(DeviceDiscoveryJoinpoint)*	A point in execution where device discovery should be performed	Any AspectJ joinpoint
LimitedConnectivity	(LCJoinpoint)*	A point in execution when limited connectivity contingency plan behaviour is required	Any AspectJ joinpoint
EHR	(action)	EHR behaviour to be carried out	Create
	(EntryArchetype)	Entry Archetypeto be used	Any valid openEHR Entry Archetype
	(CompositionArchetype)	Composition Archetype to be used	Any valid openEHR Entry Archetype
	(EHRJoinpoint)*	Point in execution where EHR behaviour is required	Any AspectJ joinpoint
Communication	(Server|Client)	Communication endpoint to be created	Server or Client
	(Technology)	Technology to be used for communication	Java Sockets
	(CommunicationJoinpoint)*	Point in execution where communication behaviouris required	Any AspectJ joinpoint

Fig. 5.8: ALPH Language Construct Parameters

Chapter 6

Evaluation

The ALPH model aims to reduce complexity in pervasive healthcare application code and in its development process. To quantifiably examine using the ALPH model we evaluate the model in terms of variations in indicators of complexity, specifically modularity and abstraction. This chapter presents a comparative evaluation of the ALPH model using five applications to illustrate its effect on application complexity. Implementations using the ALPH model are compared to versions developed using an object-oriented GPL. We examine complexity and how it can be measured in terms of modularity and abstraction using the Goal-Question-Metric (GQM) [43] approach. Results of the comparative analysis are illustrate the variations in modularity and abstraction achieved using the ALPH model.

6.1 Complexity

Complexity in software is often defined in relation to a collection of software engineering attributes known as "ilities". For example "software complexity refers to the extent to which a system is difficult to comprehend, modify and test" [27] and "complexity is often synonymous with understandability or maintainability" [72]. Complexity is therefore observed through the presence of external properties of a program, such as understandability and maintainability [257].

Software quality is also defined as the degree to which software possesses a desired com-

bination of these software engineering ilities such as maintainability, testability, reusability, reliability, interoperability [12]. The complexity and the quality of an application are therefore related [184].

Other definitions of complexity consider not only the software characteristics, but also take into account the interaction of the software with other entities i.e., people or systems "complexity is a characteristic of the software interface which influences the resources another system will expend or commit while interacting with the software" [72]. This illustrates the effect of application complexity on more than just code based quantitative values. These interaction with other people and systems are also affected by the "complexity" of a piece of software.

This thesis addresses complexity in pervasive healthcare applications by providing means to raise modularity and abstraction using the ALPH model for application development. Complexity is introduced in application code and in the development process by poor modularity and inappropriate levels of abstraction. These are identified in complexity literature both explicitly as "modularity" [27] [144] [145] [51] and "abstraction" [176] [179] and indirectly as "size" [27], "magnitute" [27], "decomposability" [27], "structure" [145] [144] and "the relationships between elements" [77].

The ALPH model addresses only the complexity introduced by poor modularity and low-levels of abstraction. Complexity is also introduced from other sources including the control structures, data flows [42] and hierarchies [77] used in systems. These are beyond the scope of this work.

6.2 Goal-Question-Metric Approach

This thesis focuses on increasing modularity and abstraction in pervasive healthcare applications. We measure the effect of the ALPH model on these two indicators of application complexity. High-level concepts such as modularity and abstraction have no corresponding single metric to allow their measurement. They do however, have many lower-level indicators that together, enable their quantification. A process that defines the use of low-level indicators to measure a high-level concept is the Goal-Question-Metric approach, or GQM

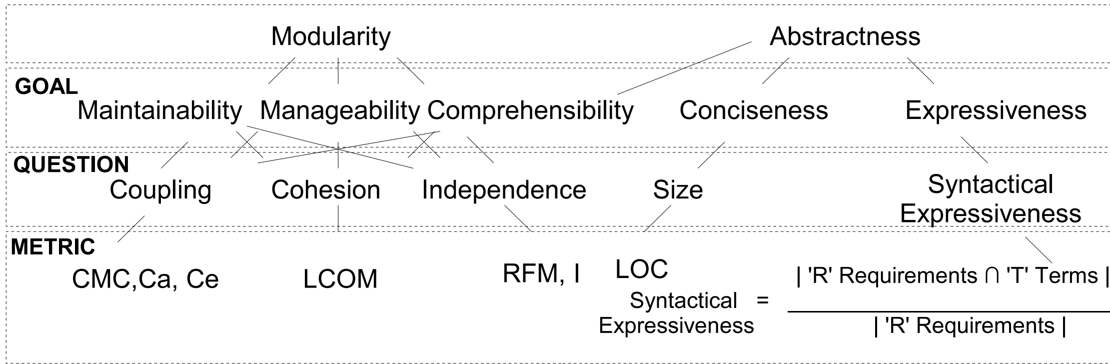

Fig. 6.1: Goal-Question-Metric Approach

[43].

We use GQM to derive quantitative measurements for the high-level conceptual characteristics of modularity and abstraction. This allows us to trace the goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals [43]. We map measurable quantitative-level metrics to operational level questions, which in turn map to conceptual level goals. The result is concrete measurements for modularity and abstraction derived using the established GQM approach.

Figure 6.1 illustrates the GQM approach applied to modularity and abstractness. To quantify changes in modularity, Parnas' benefits of modular programming [201] are used as indicators of modularity. Parnas describes how manageability, maintainability and comprehensibility reflect modularity. These in turn can be broken down into software based characteristics that signify each modular indicator based on their definition, namely coupling, cohesion and independence. Metrics exist to quantifiably measure these software characteristics. Using these metrics e.g., coupling between modules, lack of cohesion in operations, instability, measurements for coupling, cohesion and independence can be calculated and used to indicate Parnas' modularity indicators.

Metric level measurements must also be derived to measure changes in abstraction. To achieve this, literature relating to several aspects of high-level languages was analysed to

identify the assumed benefits of abstraction. Three categories of benefits emerged; expressiveness, conciseness and comprehensibility. Comprehensibility indicators are derived using Parnas' definitions, as discussed under modularity. Conciseness can be easily broken down as size is the most obvious and widely used indicator of conciseness. Program size can be quantifiably measured using a lines of code metric. Expressiveness measurements do not commonly exist. Using existing definitions of expressiveness in languages, the syntactic expressiveness of a language was identified as an indicator of overall expressiveness. A suitable metric was found to quantifiably measure the syntactical expressiveness of a language.

The following sections describe in detail how quantitative metrics are derived to measure the concepts of modularity, section 6.3, and abstraction, section 6.4.

6.3 Modularisation

To evaluate how the approach addressed modularity, we have conducted a study to measure the effect of the approach on the modularity of pervasive healthcare applications. Modularisation can be defined as the separation of concerns in an application into smaller, more independent elements known as modules. The GQM approach is applied to modularisation to identify indicators that can be quantifiably measured to illustrate variations in modularity when using the ALPH model. This section identifies the goals, questions and metrics for modularity.

6.3.1 Conceptual Level Goals

The first aim of the ALPH model is to separate crosscutting pervasive healthcare concerns to improve modularity in pervasive healthcare applications. Modular code reduces the complexity of applications and enables the modules to be developed in isolation as each concentrates and addresses a separate concern [26].

The conceptual level goals used to model modularity are based on Parnas' benefits of modular programming [201], which are:

- manageability

- maintainability
- comprehensibility

These "ilities" describe the ease with which software can be developed in components and composed easily, modified and extended and understood and are themselves indicators of complexity, as referred to in section 6.1.

6.3.1.1 Manageability

Manageability relates to ease with which components can be developed and composed. Applications that employ modularisation can be developed more easily as each module can be implemented independently. Manageability is measured by metrics that can identify the level of autonomy of a module, enabling the module to be developed and modified in isolation e.g., independence and coupling metrics.

6.3.1.2 Maintainability

Maintainability indicates the ease with which modifications and extensions can be made in the future. Modifications made to modular code affect fewer other modules and so increase the flexibility of the overall application. Maintainability assesses the modifiability of the code base and is negatively affected by dependencies between modules. Maintainability relies on metrics that affect how easily the codebase can be modified. Coupling metrics identify such dependencies between modules. The more dependencies, the more difficult it becomes to anticipate the effects a change in a module may have.

6.3.1.3 Comprehensibility

Comprehensibility evaluates how easily developers can understand, learn and use a language, application or development approach. Coupling and cohesion are good indicators of comprehensibility [28], as the number of dependencies a module has greatly increases the effort required to understand the application, as does badly encapsulated functionality. If a module

is highly complex and deals with multiple concerns, the effort to understand the module is greatly increased.

6.3.2 Operational Level Questions

Operational level questions are defined by identifying language features that best describe the goals of measurement. Each of these questions represents one or more of the goals which in turn represent the modularity and abstraction the ALPH model uses to reduce complexity in pervasive healthcare applications. The questions are defined as terms that accurately describe each goal of measurement. For modularity, the operational level questions that represent the conceptual level goals are indicators of maintainability, manageability and comprehensibility. As described in the previous section, three factors are evident as indicative of these goals: couping, cohesion and independence.

6.3.2.1 Coupling

Coupling refers to the dependencies between modules. It has been identified as an indicator of maintainability and comprehensibility [28], as dependencies introduce architectural connections between modules, making the code and control flow harder to understand. Low coupling describes a module that will not be affected by changes in another module indicating good maintainability. Low coupling is also beneficial for manageability as modules with less dependencies are easier to develop in isolation.

6.3.2.2 Cohesion

Cohesion refers to the functionality of a module, for example all methods of a class performing related tasks. High cohesion is an indicator of good encapsulation. A module exhibiting low cohesion is unfocused in its responsibilities and has multiple unrelated tasks intertwined within its code, negatively affecting maintainability. Low cohesion is also an indication that the application developer may have difficulty understanding the module as a component with high cohesion is easier to understand [28].

6.3.2.3 Independence

Independence indicates the isolation of a module. Independence is an important question in the indication of all three goals of modularity. Comprehensibility is negatively affected by modules that have many dependencies as these interactions make the module more complex. More independent modules are more maintainable as fewer modules require subsequent changes. Manageability is concerned with the ability to develop a module in isolation which is closely related to the independence of a module's implementation.

6.3.3 Quantitative Level Metrics

Quantitative level metrics are applicable measurement techniques that offer results to answer the operational level questions. The answers achieved by this step enable the quantifiable measurement of the conceptual level goals. The quantitative level metrics used to answer the questions of coupling, cohesion and independence are a combination of object-oriented and aspect-oriented metrics. The following metric suites are designed for evaluating object-oriented designs, are based on sound measurement theory and have been empirically as well as theoretically validated [259]. They are:

- Chidamber and Kemerer metrics suite (CK metrics) [64]
- Robert Martin's metrics suite (package dependencies metrics) [175]
- Henry and Li metrics suite [165]

The AopMetric[1] suite was used in conjunction with standard object-oriented metrics. This suite provides aspect-oriented extensions to the C and K, Martin and HenryLi metrics suites. Metrics tools were used to ensure a comprehensive metric analysis and are described in section 6.5.

[1] http://aopmetrics.tigris.org/metrics.htm

6.3.3.1 Coupling

Coupling measures coupling between modules (CBM) (also known as coupling between objects (CBO)) measures the number of modules that are linked to the module in question. These links are caused by referencing the module itself or some data within the module [64]. Afferent coupling (Ca) [175] measures the number of external modules that depend on module within the package being measured. Efferent coupling (Ce) [175] measures the number of external modules that depend on module within the package being measured. These metrics represent the question of coupling and quantifiably indicate the dependencies that are evident in application code.

6.3.3.2 Cohesion

Cohesion is measured by the lack of cohesion in operations (LCO) metric, which is an aspect-oriented extension to the Lack of Cohesion in Methods(LCOM) object-oriented metric. LCO is the number of operations working on different class fields minus operations working on common fields [60]. This cohesion metric highlights operations working on various subsets of module fields, indicating multiple tasks in one module. Connected methods and variables in a class have shared variables and method calls, which are indicative of cohesion.

6.3.3.3 Independence

Two metrics quantify the independence of a module, Response for a module (RFM) and instability (I). RFM is the number of methods and advices potentially executed in response to a message received by any given module [60]. This measures the possible communication between the given module and the other ones, indicating the dependence of the module on others [60]. The more possible communications, or invocations from a class, the greater the complexity of the class [64]. I measures the ratio of efferent coupling (Ce) to total coupling (Ce + Ca) such that I = Ce / (Ce + Ca). This metric is an indicator of the package's resilience to change[2] or maintainability.

[2]http://aopmetrics.tigris.org/

6.4 Abstraction

In the ALPH model, our second goal is to raise the level of abstraction. Abstractness is achieved by providing the developer with concise, expressive constructs which shield low-level functionality. This abstraction is encapsulated in the provision of a DSL. While modularisation has been quantified [190] [60] using the described metrics in the previous section, very few studies have quantified improvements offered by abstraction [119]. The lack of measurements for abstraction is a hard and an important open problem' [179]. The GQM approach is particularly useful in quantifying abstraction due to the lack of measurements targeted at high-level languages providing abstraction. To evaluate abstraction capabilities when using the ALPH model, we base our measurement features on the perceived benefits of using high-level abstractions.

6.4.1 Abstraction Benefits

We examined literature associated with high-level languages that provide abstractions to application developers in a target domain. Literature describing, using, critiquing, taxonomising and analysing high-level languages [159] [133] [179][119] [118] [81] were all considered. Claimed benefits were:

- B1 "Programs are concise and can be reused." [159].
- B2 "Allows development quickly and effectively, yielding programs that are easy to understand, reason about, and maintain." [133]
- B3 "Programs are generally easier to write, reason about, and modify." [133]
- B4 "Offers substantial gains in expressiveness and ease of use." [179]
- B5 "Enable a concise representation of a programmer's intention." [119]
- B6 "Programs are more concise." [119]
- B7 "Enhance the productivity of an application developer." [118]

- B8 "Express solutions at the level of abstraction of the problem domain." [81]
- B9 "Allow domain experts to better understand, validate, modify, and often even develop DSL programs." [81]

It emerged that certain benefits were occurring frequently, described using differing terms and synonyms. The categories exposed were:

- Expressiveness i.e., intuitive and domain relevant constructs (B4, B5, B8);
- Conciseness i.e., size (with resulting increased productivity) (B1, B5, B6, B7)
- Comprehensibility i.e., developer understanding; [3] (B2, B3, B4, B9)

6.4.2 Conceptual Level Goals

Abstraction is deconstructed using the GQM approach to quantifiably measure the variations in abstraction when using the ALPH model to develop pervasive healthcare applications. The benefits identified in section 6.4.1 are the conceptual level goals. They represent high-level advantages and are the starting point for the identification of quantitative metrics. Each goal is described in this section under the headings identified in section 6.4.1.

6.4.2.1 Expressiveness

We define expressiveness in a programming language as the support a language provides to a developer in performing tasks using a comprehensive, semantically intuitive syntax. In a DSL, expressiveness is expected to increased compared to GPLs as domain language is generally used as part of the language's syntax [179].

We endeavour to quantify the relationship between the words and the expressiveness of the language. This goal has been identified in conceptual modelling research [168] and in recent work [125] both under the title of expressiveness and in many synonymic terms including language appropriateness, domain appropriateness etc.

[3] Comprehensibility is addressed in detail in the GQM breakdown of modularity and therefore is not repeated in the measurement of abstraction. Its measurement will indicate both benefits achieved by modularisation and abstraction.

6.4.2.2 Conciseness

The fundamental purpose of a DSL is to concisely describes applications in the domain [81]. A DSL exhibits conciseness when one can express the same semantics in a smaller body of material using the given language, than in a GPL. A DSL is typically more concise because the notations and abstractions modelling the domain abstract the developer away from low-level GPL code. They often execute the same functionality as a verbose GPL program but with a reduced body of code. The conciseness of a DSL allows the syntax to be declarative because the domain semantics are clearly defined, therefore constructs have a precise interpretation [119].

6.4.3 Operational Level Questions

The operational level questions are defined as terms that accurately describe each goal of measurement as follows:

- Expressiveness is represented by *Syntactic expressiveness.*
- Conciseness is incorporated into a measure of *Size*, as codebase size is the most direct and common measurement of conciseness.

6.4.3.1 Syntactic Expressiveness

The syntactic expressiveness of a language is defined as the adequate provision of constructs that enable the developer to fulfil the requirements of an application [53]. Fabbrini [94] defines a relationship between the language used in the requirements and the ability of people to understand the language. It has also been defined as follows: "The expressiveness of a language is some measure of the variety of lexical and grammatical constructions it allows" [239].

6.4.3.2 Size

With conciseness as a goal, the crucial and basic indication required is size. The size of the program or module produced by the DSL may be a primitive measure, but size is still the

most widely measured software detail [264] and is the most common concept used in language comparison. We measure DSL size by means of the code that the developer is responsible for implementing. This measurement detail takes into consideration the generative nature of DSLs, eliminating generated code. DSLs are mostly declarative in nature resulting, in generally reduced size.

6.4.4 Quantitative Level Metrics

Metrics are the lowest-level representation of the DSL under measurement. Each operational level question is answered by a corresponding metric;

- The question of syntactic expressiveness is answered by using the *Syntactic Expressiveness Valuation (SEV)* metric.
- The question of size is answered by the most commonly used metric for measuring software size, *Lines of Code (LOC).*

Metrics are applied to yield concrete measurements for each operational level question. An analysis of previous attempts at evaluation each question was undertaken. We attempted to make use of previously verified measurements when possible, as the technique of measurement is more valid when it has been proven to evaluate a subject. Metrics exist for code based measurements which can be used to measure size. Where common metrics were not applicable, i.e., expressiveness, measurements applied to related topics were analysed. Unsuitable metrics [207] [202] were disregarded. Causes for unsuitability were; the fact that a metric could not be applied to a DSL in a way that developers could repeat i.e., huge computation or unavailable data inhibiting repetition, and the difference in definition of measurement subjects. A suitable metric were found in the analysis of expressiveness measures for natural languages [53].

6.4.4.1 Syntactic Expressiveness Valuation

Zipf's law is one of the most noted linguistics laws. It describes the statistical distribution of words with different ranks by means of frequency, revealing the relationship between word

$$\text{Syntactical Expressiveness} = \frac{|\text{'R' Requirements} \cap \text{'T' Terms}|}{|\text{'R' Requirements}|}$$

Fig. 6.2: Syntactic Expressiveness

frequency and its rank in a language. Zipf's law [282] states that the frequency of any word is roughly inversely proportional to its rank in the frequency table i.e., there are a small number of commonly used words, and a large number of infrequently used words. To measure syntactic expressiveness we need to measure the number of domain requirements that can be syntactically expressed by a language. The measure is described using set theory notation as shown in figure 6.2.

In other words, the syntactic expressiveness of a language is defined as the number of requirements that can be syntactically expressed by the language divided by the number of requirements. If we measure the words most frequently used in the domain, we can discount the lesser used words and maintain the ability to perform most of the domain functionality. We can also remove synonyms without reducing the expressiveness of the language as the semantic meaning is maintained [53]. Firstly we identify particular parts of speech (POS) within the requirements using a tool e.g., verbs [53]. The frequency with which these words occur can then be calculated. Next, we can measure the number of domain-specific requirements that can be syntactically expressed by the words divided by the number of requirements in the domain i.e., the syntactical expressiveness. The results range in percentage value SEV for a corresponding fraction of the syntax. Following the examination of previous tests [53] and the application of this metric to DSLs, a value of 50% SEV should be acheived using the fewest possible constructs from the language syntax. As this measurement is appicable at the language level, and not per application, the results for SEV will be included later in this chapter.

6.4.4.2 Lines of Code

Lines of Code (LOC) counts the number of lines of class code. Although crude, LOC is still the most commonly used metric to represent software size [264]. We count the number of lines of class code based on the internal string representation of the JDT compiler, the result of which is independent from original code styling etc. This results in a measurement of program length. We measure the LOC from the application developer's point of view, that is how many lines of code the developer implements, not the amount of code generated.

6.5 Tools

Metrics were computed using various tools. CyVis[4] is a software metrics collection and analysis tool used to compute metrics. AopMetrics[5] was used to calculate object-oriented metrics and aspect-oriented metrics. It is extended to take aspect-oriented language features into account, e.g., advice, pointcuts etc. JDepend[6] traverses Java class file directories and generates design quality metrics for each Java package. These package measurements show higher level dependencies. These tools compute the quantitative level metrics discussed in section 6.3.3 and section 6.4.3.

6.6 Applications

The following sections outline the applications used in the evaluation of the ALPH model.

6.6.1 DBay

DBay is a case study based on an online auction system. It illustrates the various context handling and mobility issues that can affect the average distributed application scenario. It has been modified from a generic online auction site case scenario [13] to make explicit references to interactions with mobile devices. The auction system, known as DBay, was

[4]http://cyvis.sourceforge.net/index.html
[5]http://aopmetrics.tigris.org/
[6]http://clarkware.com/software/JDepend.htm

implemented using two techniques, an object-oriented (OO) approach using Java and using the ALPH model. In the OO version, all pervasive healthcare functionality was coded where needed within the base application. Using the ALPH language, the required crosscutting concerns as listed in table 6.1 were separated in the DSL.

	DBay
Distribution	✓
Communication	✓
Network Roaming	✓
Software Roaming	✓
Service Discovery	✓
Device Discovery	✓
Limited Connectivity	✗
Quality of Service	✓
Device Adaptation	✓
Location	✓
EHR	✗
HL7	✗
Persistence	✗

Table 6.1: DBay Concerns

6.6.2 HL7Browser

The second application used in the evaluation of the ALPH model is an open-source product, "HL7 Browser"[7]. This third party application uses HL7 functionality, and has been developed using standard Java. In this implementation, the HL7 functionality is scattered throughout the application. We refactored the HL7 functionality into an aspect that uses the library in the ALPH model and compared the two implementations. This HL7 functionality is implemented using the ALPH model, as shown in table 6.2.

[7]http://nule.org

	HL7Browser
Distribution	✗
Communication	✗
Network Roaming	✗
Software Roaming	✗
Service Discovery	✗
Device Discovery	✗
Limited Connectivity	✗
Quality of Service	✗
Device Adaptation	✗
Location	✗
EHR	✗
HL7	✓
Persistence	✗

Table 6.2: HL7Browser Concerns

6.6.3 MedHCP

The third application used in the evaluation is MedHCP. MedHCP is a case study scenario of pervasive healthcare [37]. The scenario was conceived by the Centre for Pervasive Computing and staff at a collaborating hospital in Aarhus county, Denmark.

It describes a futuristic picture of a pervasive healthcare application that exploits pervasive computing technology resulting in an advanced pervasive healthcare application. The scenario depicts an ideal use for mobile and pervasive computing devices in the medication of patients in a hospital setting and requires support for many of our identified pervasive healthcare concerns. The requirements identified by the authors of the scenario (mobile devices, composite devices, heterogeneous devices, discovery of resources, location and context awareness) [37] map closely to the concerns we identified in section 3.6, as shown in table 6.3, making it suitable for evaluation purposes.

MedHCP has been implemented in two ways; using traditional OO developmental techniques and using the proposed ALPH approach. The object-oriented approach used a gen-

Fig. 6.3: Mobile Clinical Assistant

	MedHCP
Distribution	✗
Communication	✓
Network Roaming	✗
Software Roaming	✗
Service Discovery	✓
Device Discovery	✓
Limited Connectivity	✗
Quality of Service	✗
Device Adaptation	✗
Location	✓
EHR	✗
HL7	✓
Persistence	✓

Table 6.3: MedHCP Concerns

eral purpose language, Java, for the implementation of all functionality. The ALPH approach was implemented using ALPH to define the incorporation of the required pervasive healthcare functionality and Java for the base application functionality. The application was deployed on the Motion C5 mobile clinical assistant (MCA) created by the Digital Health Group (DHG) at Intel Health [8], as shown in figure 6.3. The Motion C5 MCA is a hospital-grade portable device specifically designed for use by healthcare professionals in hospital environments.

6.6.4 Healthwatcher

Healthwatcher is a web-based healthcare information system developed by the Software Productivity research group from the Federal University of Pernambuco and made available through Lancaster University [120]. This application was developed as a testbed codebase and many implementations are available. We use only the OO Java implementation as a comparison against a refactored version using the ALPH model.

	Healthwatcher
Distribution	✓
Communication	✓
Network Roaming	✗
Software Roaming	✗
Service Discovery	✗
Device Discovery	✗
Limited Connectivity	✗
Quality of Service	✗
Device Adaptation	✗
Location	✗
EHR	✗
HL7	✗
Persistence	✓

Table 6.4: Healthwatcher Concerns

[8] http://www.intel.com/healthcare/ps/mca/index.htm

6.6.5 Rococo

Rococo [10] is a Bluetooth mobile phone software company based in Dublin, Ireland. They build Bluetooth-based mobile phone software for manufacturers including Nokia and Sony Ericsson. Rococo provided access to the codebase for a mobile phone based chat application built using J2ME. We refactored the provided OO application and replaced functionality from the pervasive healthcare behaviour in ALPH with domain-specific instructions as shown in table 6.5.

	Rococo
Distribution	✗
Communication	✗
Network Roaming	✗
Software Roaming	✗
Service Discovery	✓
Device Discovery	✓
Limited Connectivity	✗
Quality of Service	✗
Device Adaptation	✗
Location	✗
EHR	✗
HL7	✗
Persistence	✗

Table 6.5: Rococo Concerns

6.7 Results

This section outlines the results from comparative analysis of quantitative level metric results from the OO and ALPH implementations of the applications described in section 6.6. This section outlines the results from the quantitative metric analysis.

6.7.1 Presentation of results

Metrics implemented by the Aopmetrics project can be applied to both classes and aspects. Therefore, "module" can be used as a common term for classes and aspects. Similarly, methods, advices and introductions will be referred to as operations. All OO results are depicted in graphics in a dark gray colour while the ALPH model results are illustrated using a light white colour.

It should be noted that the code generated by the ALPH model is not included in some graphical representations, e.g., when it creates new modules which would have no comparative component in the OO implementation. All generated code is fully included in explanations and taken into consideration in the overall evaluation except where specified i.e., size measurements considering the implications for the application written by the developer, not the generated implementations.

6.7.2 Coupling

In DBay the ALPH model significantly reduced the coupling of the modules which were most affected by pervasive healthcare crosscutting concerns, as shown in figure 6.4. CBM shows that coupling in the ALPH approach was decreased by up to 42% due to the removal of method calls in the OO version, used to include domain-specific functionality.

As expected, Ce in packages with many calls to other modules for concern behaviour in the OO version decreases in the ALPH implementation, as shown in figure 6.5. They becomes more independent due to the encapsulation of its core functionality and the separation of the pervasive concerns. As shown in figure 6.5, the OO approach has a high external dependency for Packages 1, 2 and 3 as they rely on other modules for pervasive functionality. The ALPH model provides this functionality through aspects introduced by the ALPH DSL, which those packages are not dependent on, therefore reducing their outward dependencies. These results illustrate the modularisation benefits of AOP on the base classes, as they can be oblivious to the functionality in the aspects [103]. Packages 4 and 5 contain the generated behaviour in the ALPH implementation. In the OO approach, they have a low Ce as they has minimal outward dependencies. These dependencies increase using the ALPH model due to the use

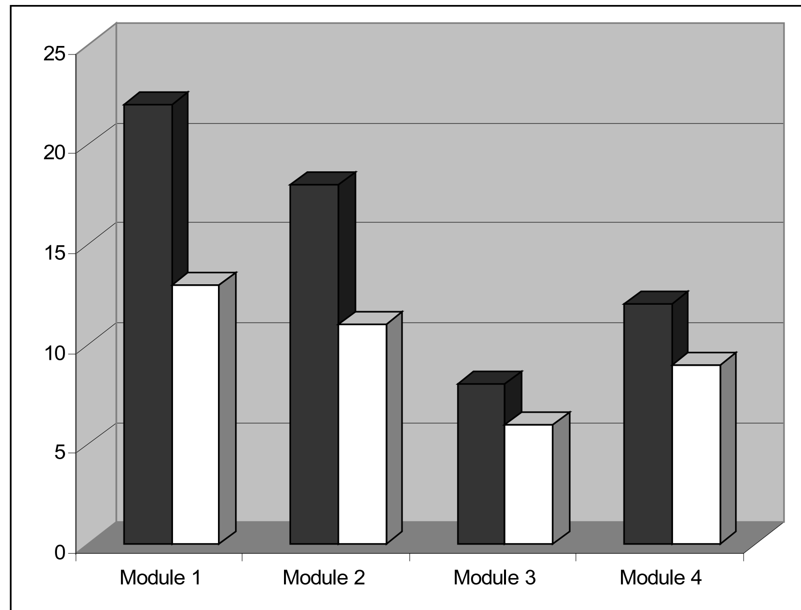

Fig. 6.4: DBay Coupling Between Modules

of base class information in the aspects.

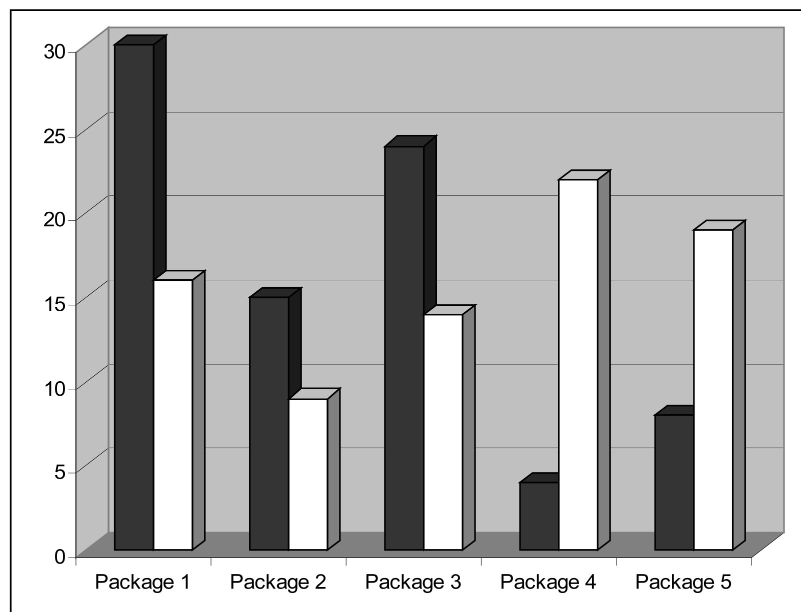

Fig. 6.5: DBay Efferent Coupling

Figure 6.6 shows the increase in Ca in Packages 1, 2 and 3 using the ALPH model. Pointcuts in aspects refer to syntactical references in the base code , e.g., method signatures, as indicators of where crosscutting behaviour should occur. These references introduce inward dependencies on the base application, hence the increase in Ca. Packages 4 and 5 contains the generated pervasive healthcare aspects. The AOP code does not have any additional inward dependencies. There is an expected decrease in dependencies in Package 2 as the OO version contained domain-specific code, which has been removed.

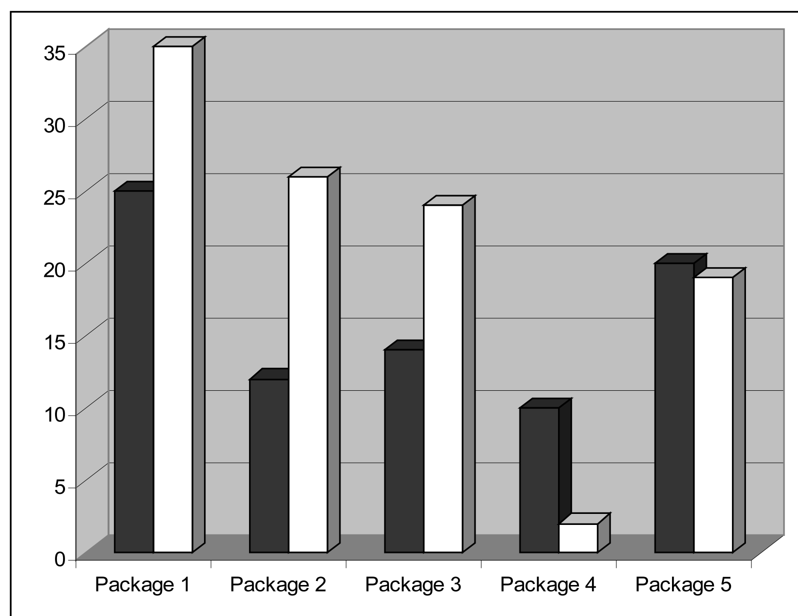

Fig. 6.6: DBay Afferent Coupling

In the HL7Browser application, the package containing the modules most significantly affected by the refactoring of the pervasive healthcare functionality was evaluated using package level metrics. Coupling in the package was reduced by up to 67%, as illustrated by Ce in figure 6.7. This reduction in coupling is due to the reduction in outgoing dependencies as the base application no longer makes calls related to the HL7 functionality. CBM further illustrates a reduction in coupling with a decrease of up to 27% observed in base application modules. The inward dependency introduction of AOP can be seen in the increase of Ca by 33%. Ca is a coupling measurement to quantify the number of modules outside a package

that depend on modules within the package, and so is increased by the use of base application references when using AOP.

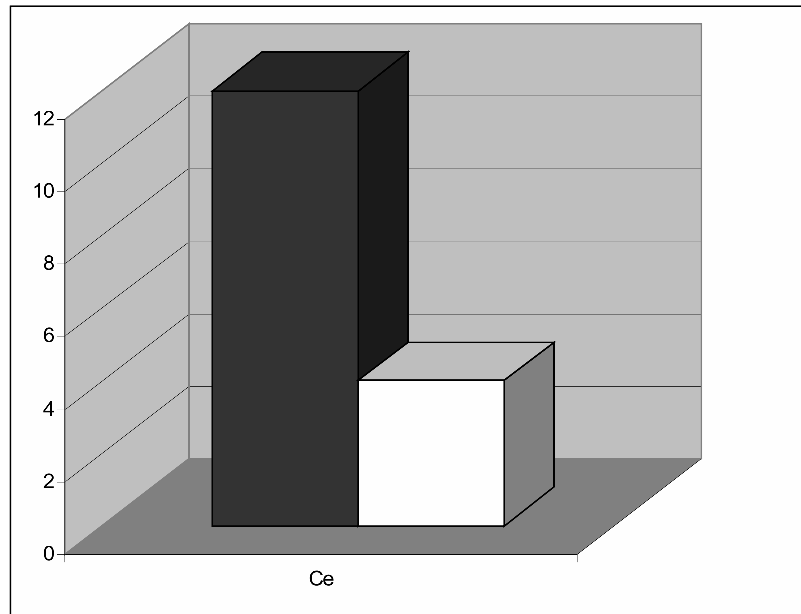

Fig. 6.7: HL7Browser Efferent Coupling

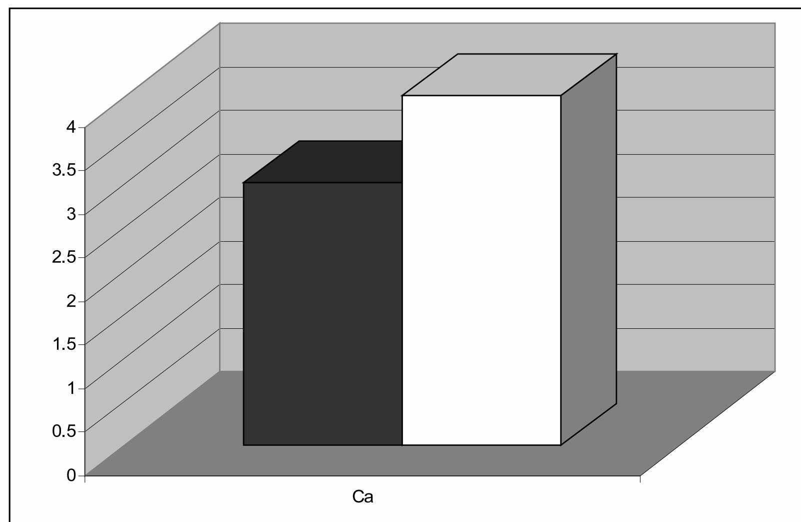

Fig. 6.8: HL7Browser Afferent Coupling

In MedHCP, figure 6.9 illustrates the coupling measurements for the modules that were

most affected by the handling of pervasive healthcare functionality. By modularising the domain-specific concerns, the ALPH model significantly reduced coupling as the base code does not explicitly make use of ubiquitous computing or healthcare code at any time. Coupling was reduced by between 33% and 75% in the affected modules.

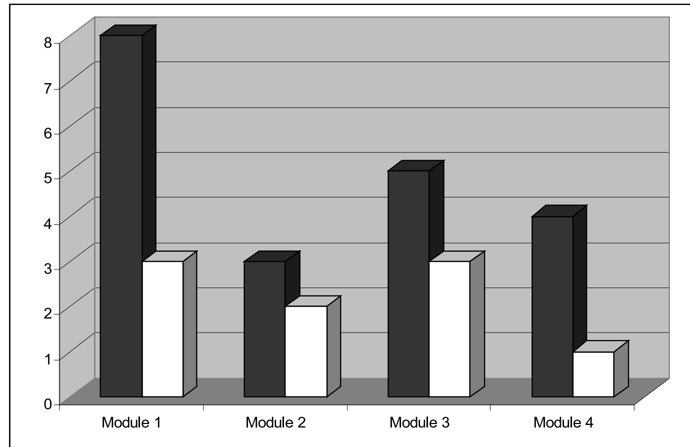

Fig. 6.9: MedHCP Coupling between Modules

The Ca of the three modules most influenced by the pervasive healthcare concerns are illustrated in Figure 6.10. Package 1 and 2 have an increased number of dependencies using the ALPH model. The aspect makes use of information from the base classes therefore introducing dependencies. The dependency is due to the use of syntactical elements such as method names in pointcut designators. The dependencies on Package 3 do not differ in the OO and ALPH approach as Package 3 has no additional dependencies from aspects or other modules.

As shown in Figure 6.11 Package 1 has a high external dependency in the OO approach as it called other modules to perform ubiquitous computing and healthcare functionality. The ALPH model provides this functionality through aspects, on which Package 1 does not depend, therefore reducing Package 1's dependencies by 40%. Package 2 and 3 are also less dependent on external modules using the ALPH model. These results illustrate the modularisation benefits of AOP on the base classes, as they can be oblivious to the functionality in the aspects [103].

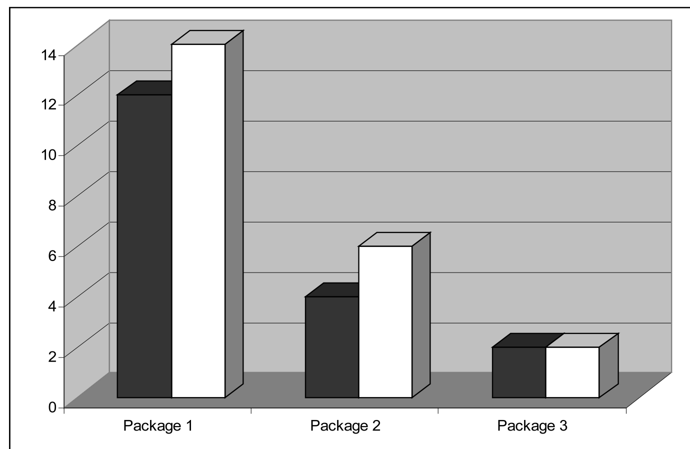

Fig. 6.10: MedHCP Afferent Coupling

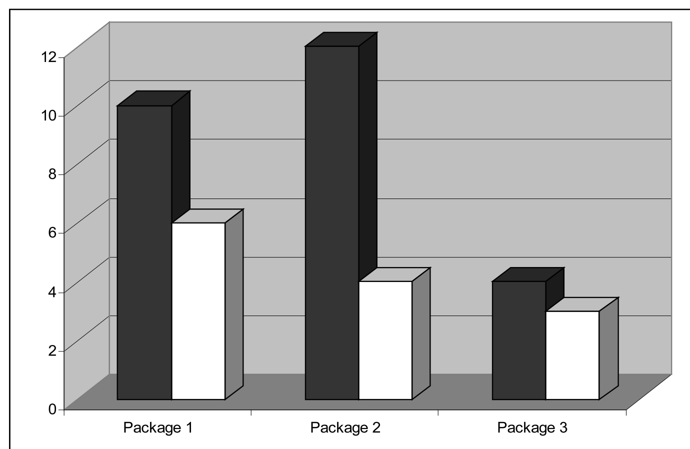

Fig. 6.11: MedHCP Efferent Coupling

In the Healthwatcher application, figure 6.12 shows that the ALPH model reduced the coupling of the modules most affected by pervasive healthcare crosscutting concerns. CBM reductions, however, were smaller than in other evaluation applications. Healthwatcher is a large application and the functionality addressed in the ALPH implementation represented a smaller percentage of code than in other evaluation applications.

The first of the package coupling metrics, Ca, shows a reduction in outward coupling dependencies in a third of the packages using ALPH. One package is hugely increased due to the addition of aspects and the introduction of outward dependencies on base code in other

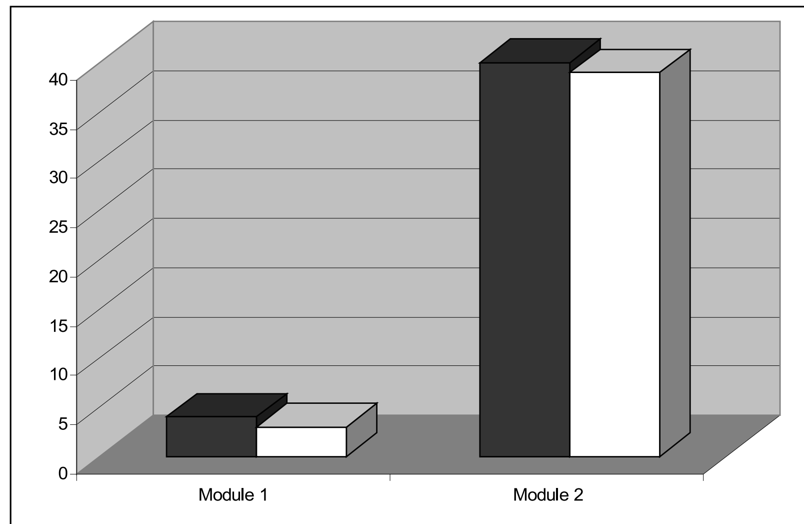

Fig. 6.12: Healthwatcher Coupling Between Modules

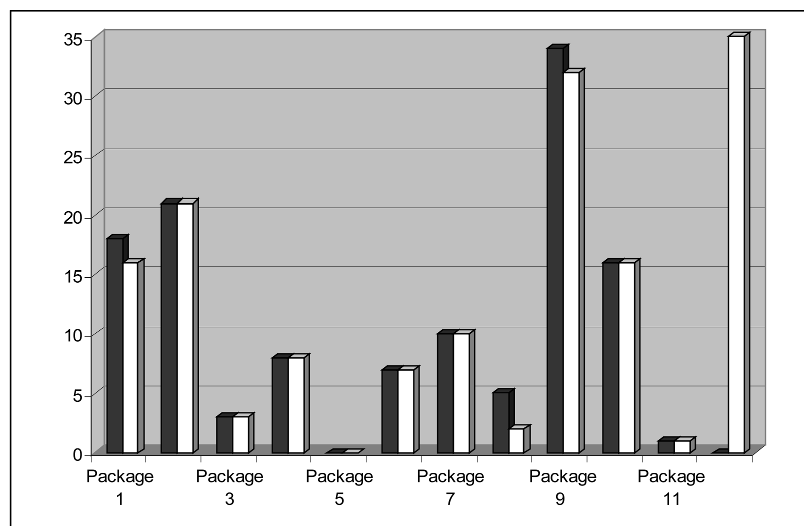

Fig. 6.13: Healthwatcher Efferent Coupling on all Packages

package. The remaining modules remain unchanged, as shown in figure 6.13. Of the modules reduced, 11%, 60% and 6% reductions were observed using the ALPH model as shown in figure 6.14. These reductions are due to the removal of dependencies on other modules for pervasive healthcare functionality.

Ca is the number of modules outside this package that depend on modules within the

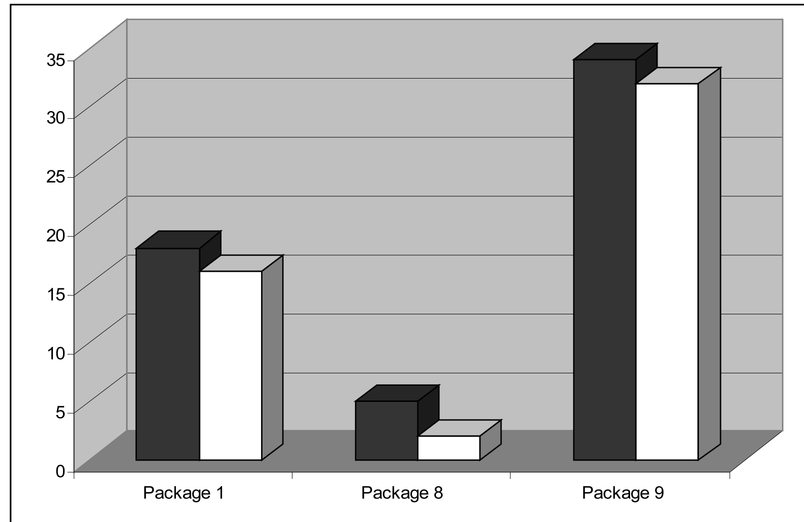

Fig. 6.14: Healthwatcher Efferent Coupling on Reduced Packages

package. Using the ALPH model we expect to see an increase in any module referred to in the aspects provided by ALPH due to the dependencies introduced by references to points in execution in the base code. Figure 6.15 shows that equal numbers of modules had Ca readings reduced and increased using the ALPH model. This illustrates that the modules reduced pregiously provided domain-specific behaviour, as the dependencies inwards, e.g., method calls, were decreased. Modules increased are referred to by aspects, increasing inward dependencies on base application code.

Coupling results in the Rococo application showed a similar pattern to the previous evaluation applications when using the ALPH model for application development. CBM was reduced in all base modules by up to 52%. The new module, module 9, containing the AOP implementation causes an obvious increase by its creation. Outgoing dependencies were significantly reduced due to the removal of calls to other modules for domain-specific functionality. This is now provided by the ALPH model reducing outgoing dependencies in the base application modules as shown in figure 6.16. Again, incoming dependencies were increased in each base module by the AOP references by 11% to 34% as shown in Ca in figure 6.17.

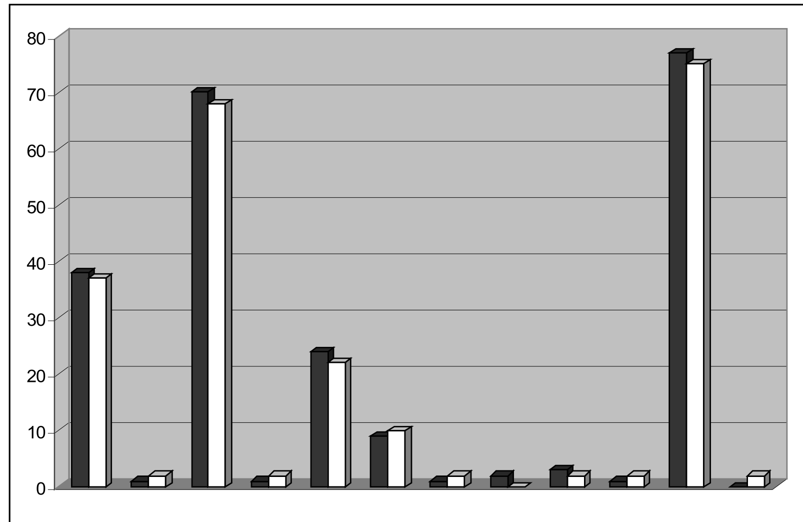

Fig. 6.15: Healthwatcher Afferent Coupling

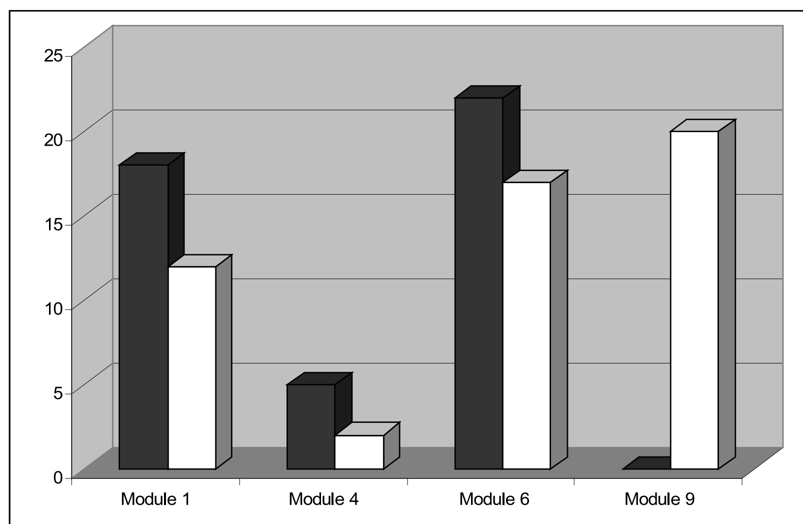

Fig. 6.16: Rococo Coupling Between Modules

6.7.3 Cohesion

In DBay, the LCO value in the OO approach was made up to 8% more cohesive using the ALPH model illustrating the modularisation of the pervasive healthcare concerns outside the base functionality.

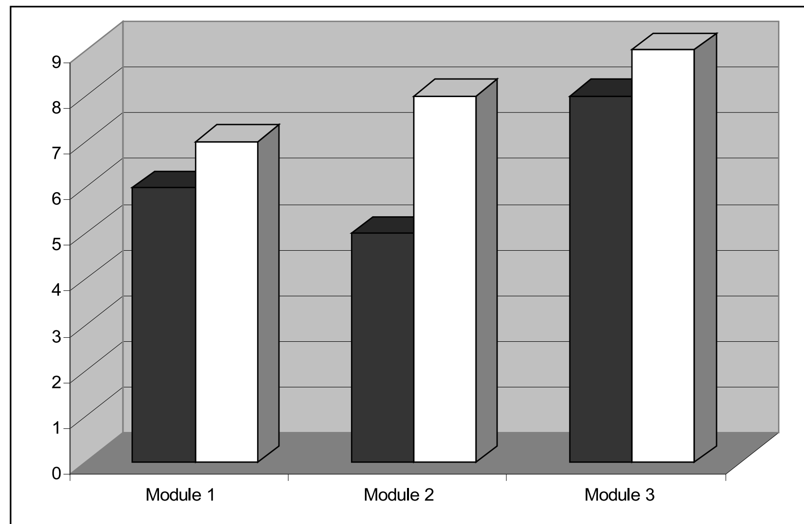

Fig. 6.17: Rococo Afferent Coupling

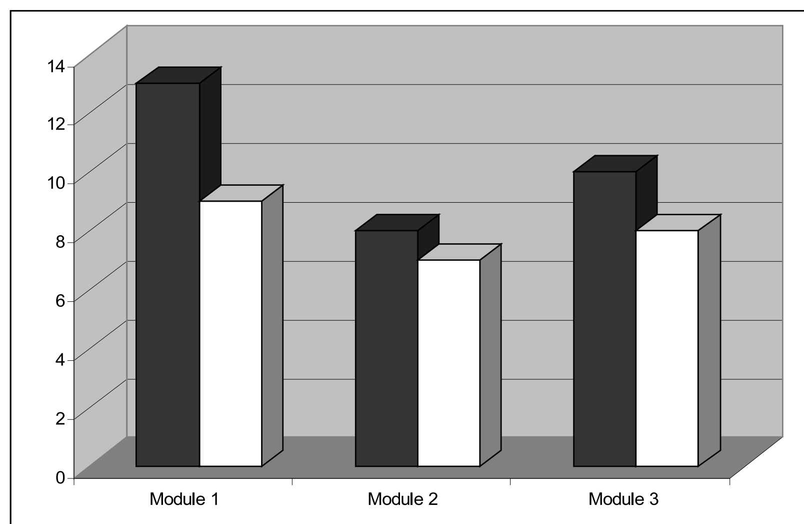

Fig. 6.18: Rococo Efferent Coupling

In HL7Browser, the cohesion of modules, where variations were observed, were also positively affected using the ALPH model as all base modules saw decreases in LCO measurements by up to 11%.

Cohesion observed similar variations to RFM in MedHCP. LCO was increased in three modules, as shown in figure 6.20. This is caused by the same increased number of advice,

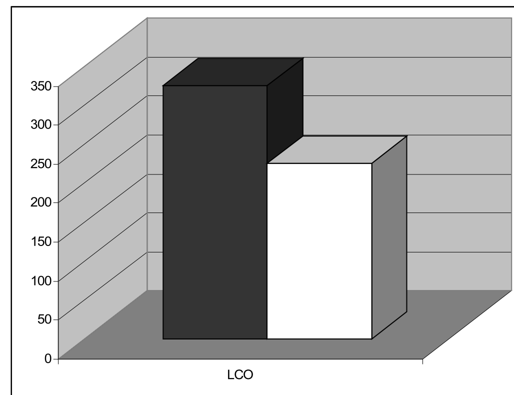

Fig. 6.19: HL7Browser Lack of Cohesion

compared to methods in the OO version, that caused the increase in RFM as the advice is working on base application elements.

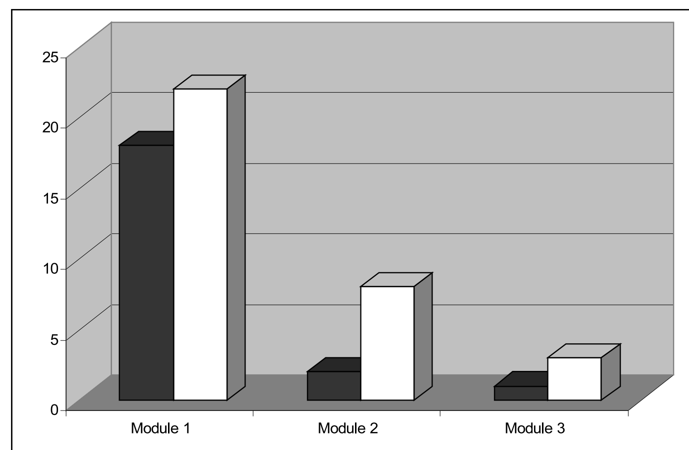

Fig. 6.20: MedHCP Lack of Cohesion

Cohesion was not varied greatly by using the ALPH model in the Healthwatcher system. No real change was observed with the average LCO for methods improving marginally. The ALPH version replaces method calls with equal numbers of potential advice, resulting in no quantifiable variation.

In the Rococo mobile application, cohesion in modules was increased by up to 7% by reducing the number of concerns the base application implements.

6.7.4 Independence

Figure 6.21 summarises the RFM results from the DBay application. The RFM is reduced in the ALPH model by up to 13%. This increases comprehensibility, as when a large number of methods are invoked, the application requires a greater understanding of the execution path and control flow of the application.

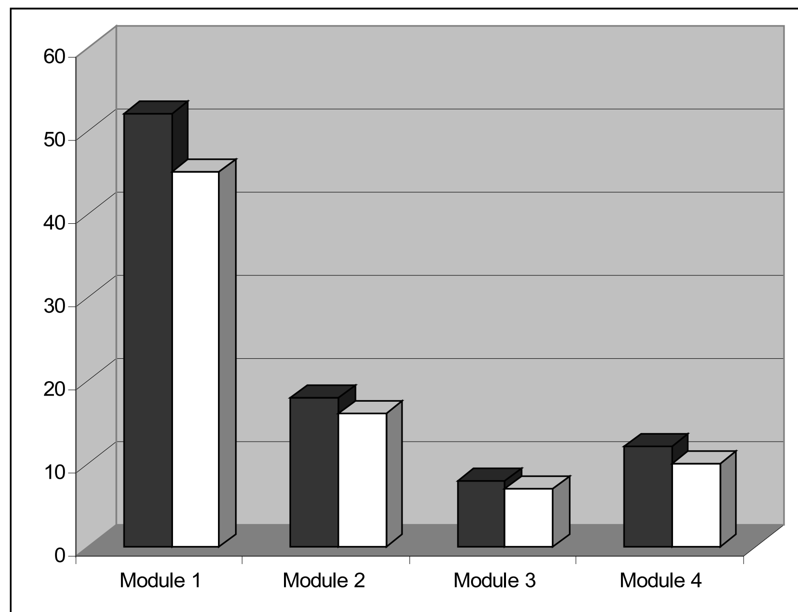

Fig. 6.21: DBay Response for a Module

As shown in figure 6.22 the stability of packages in DBay were both increased and decreased. Increases in stability are due to the ALPH module modularising the crosscutting concerns externally, reducing the total dependencies for the package. Packages made less stable had more dependencies introduced than removed using the ALPH model.

Stability in the HL7Browser application was increased by the ALPH model by 37.5%, as shown by the decrease in the I (Instability) metric. Instability indicates a package's resilience to change. The package is more stable using ALPH due to the removal of package dependencies on other modules for HL7 functionality.

However, the ALPH model produces some negative results. While using AOP, pointcuts refer to syntactical references in the base code as indicators of where crosscutting behaviour

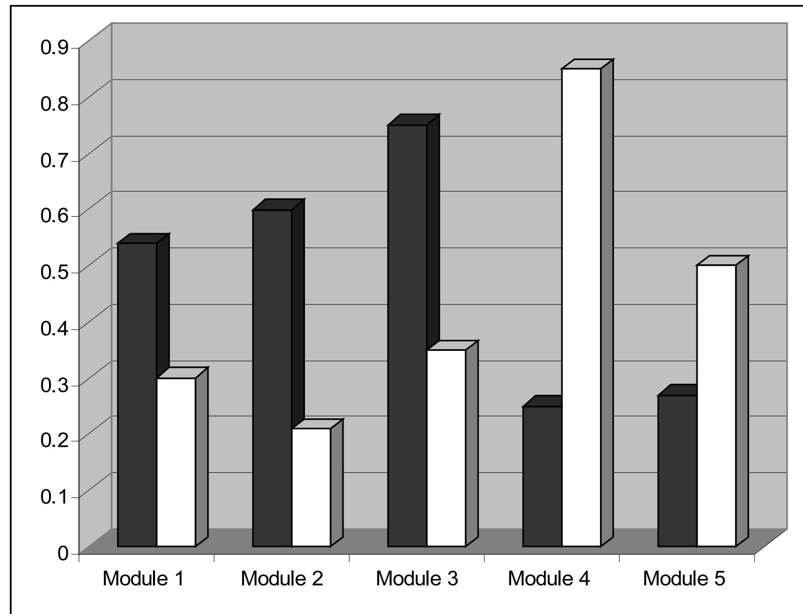

Fig. 6.22: DBay Instability

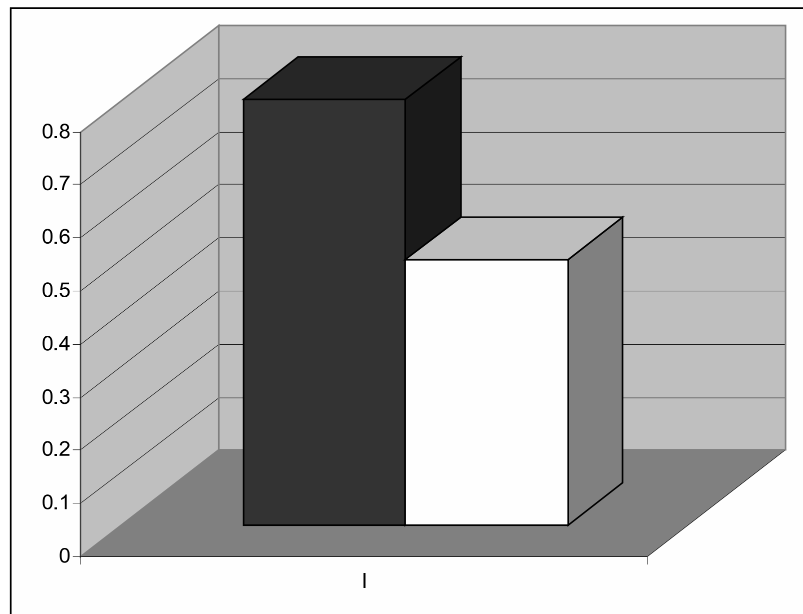

Fig. 6.23: HL7Browser Instability

is to be injected. These references introduce dependencies inwards on the base application which did not exist before the use of AOP in the ALPH model. This is evident in this study by the increase in RFM (Response for a Module) coupling metric in a module by 11%. This measure indicates the number of potential advices that could be executed within the particular module.

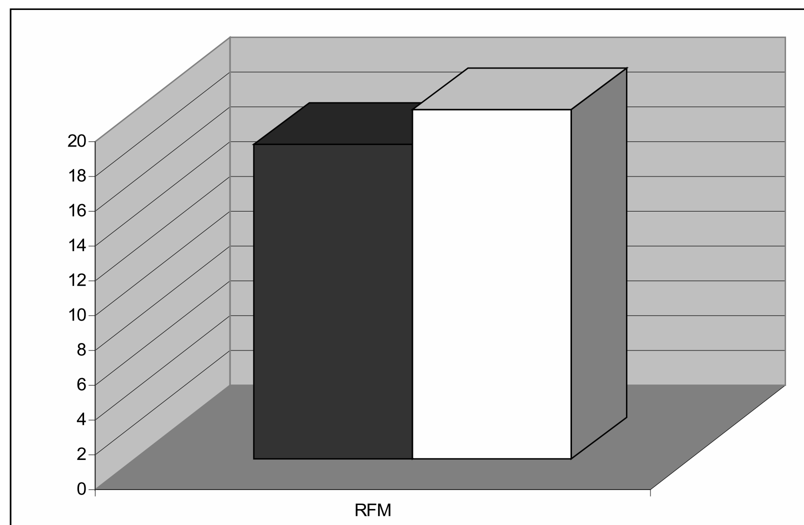

Fig. 6.24: HL7Browser Response for a Module

Figure 6.25 summarises the results for RFM in MedHCP. The RFM is reduced by 20% in module 1 and 60% in modules 3 and 4. This is due to the reduced number of calls to other modules decreasing coupling and complexity.

The increase in RFM for module 2 is caused by the number of potential joinpoints in the ALPH implementation being higher than the number of method calls in the OO version. The RFM metric is extended to take the number of advices implicitly invoked into account. The increase in potential joinpoints result from design decisions in the ALPH implementation, where more than one advice may carry out the behaviour previously implemented in one method. The results illustrate the increased independence of base modules when modularisation using the ALPH model, but also observe a negative result in specific circumstances.

As shown in Figure 6.26, the stability of the three modules that had most crosscutting functionality modularised in MedHCP were all increased. The modules were made more

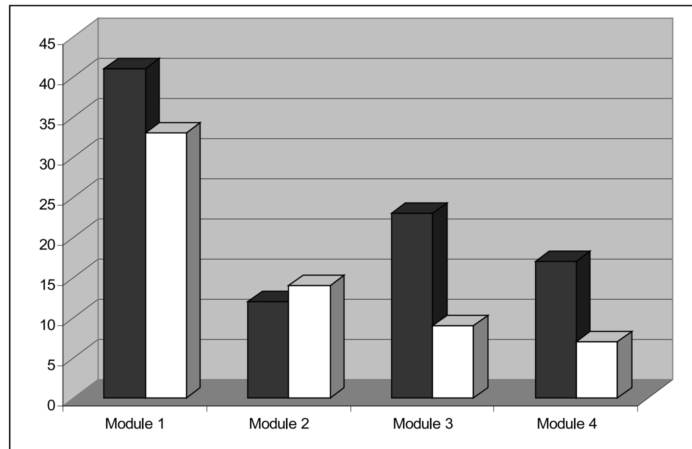

Fig. 6.25: MedHCP Response for a Module

stable by the reduction in dependencies within the base code.

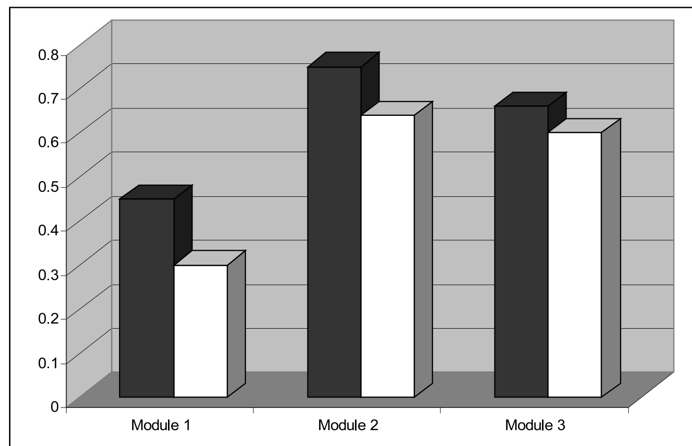

Fig. 6.26: MedHCP Instability

Healthwatcher RFM measurements were reduced by using the ALPH model in comparison to the OO implementation. Figure 6.27 summarises the results from both approaches showing reductions of 7% and 20% in the two most heavily affected modules.

Figure 6.28 shows that 7 of the 12 modules measured in the Healthwatcher application were made more stable using the ALPH model. 4 were increased due to the dependencies introduced by AOP outweighing the reduction in dependencies for the entire module as a

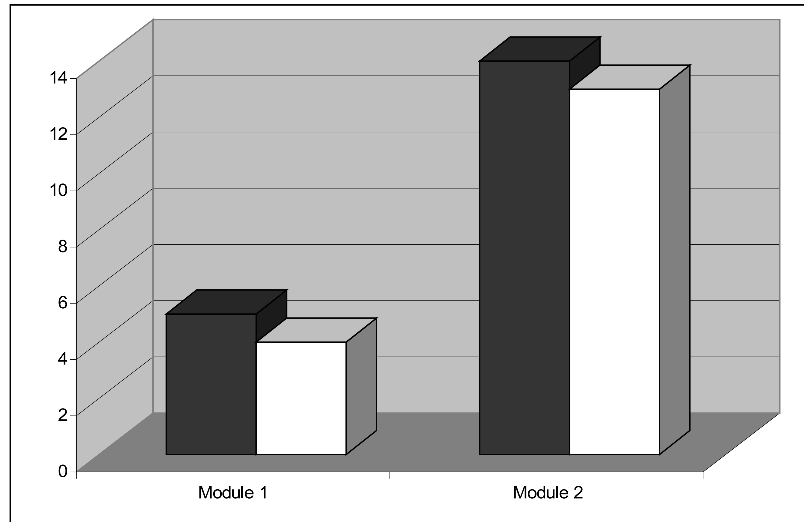

Fig. 6.27: Healthwatcher Response for a Module

whole. The new module containing the aspects is included in these results. 1 remaining module remained unchanged. Of the reductions observed, modules were reduced by 6.23%, 4.29%, 9.9%, 8.36%, 28.58% and 5.6% as illustrated in figure 6.29. This increase in stability is consistent with other evaluation applications. The decrease observed can be explained as some modules had more depedencies introduced than removed, e.g., more advices than method calls.

In the Rococo application RFM shows reductions in two-thirds of the base modules as there are fewer method calls. The increase in the other modules is a result of more potential advice executions introducing a greater dependence on the module's structure. Instability was increased in each base application module by up to 28%, making the application more resilient to changes and more easily maintained.

6.7.5 Size

In DBay, the ALPH model decreased the code written by the application developer by 18%. This illustrates the large proportion of crosscutting code that was tangled within the base application. The reduction in application size illustrates the advantage of using a concise high-level language to encapsulate domain-specific behaviour. The HL7Browser application

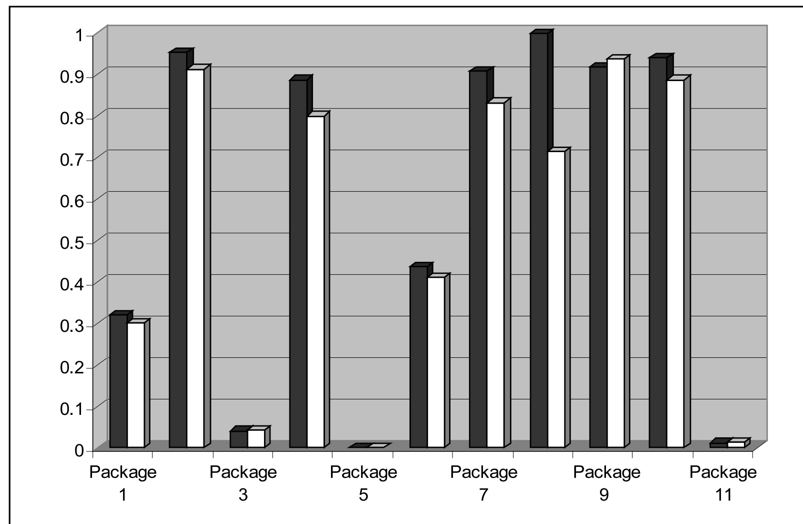

Fig. 6.28: Healthwatcher Instability

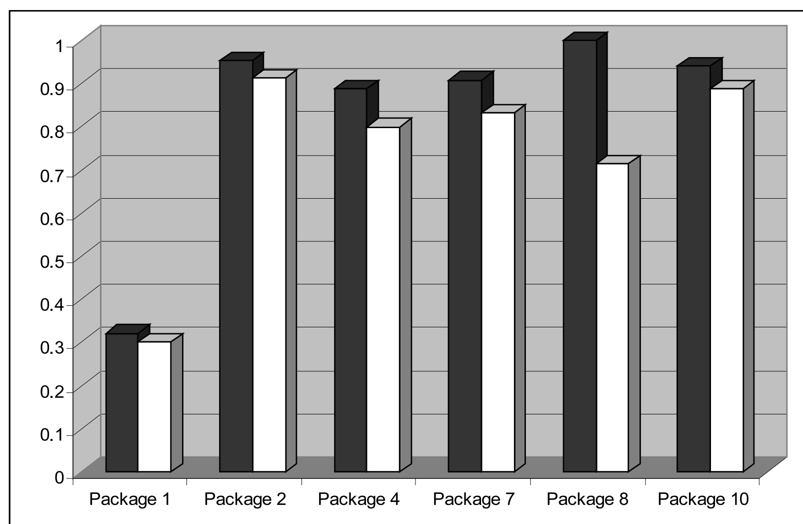

Fig. 6.29: Healthwatcher Instability Reductions

was reduced in size by 8% using the ALPH model. The module most heavily effected by the refactoring of the HL7Browser application was reduced in size by 70%, illustrating the crosscutting that took place.

Measuring the LOC from the application developers' point of view, that is how many lines of code the developer would have to implement, ALPH makes the MedHCP application 25%

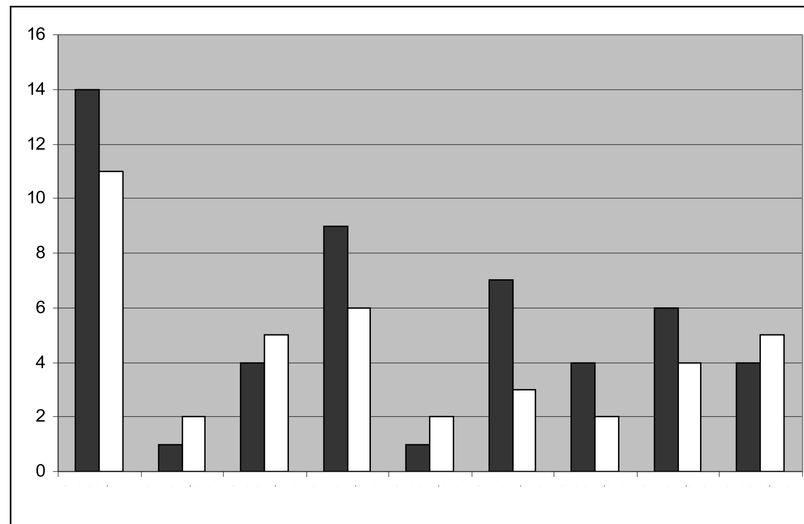

Fig. 6.30: Rococo Response for a Module

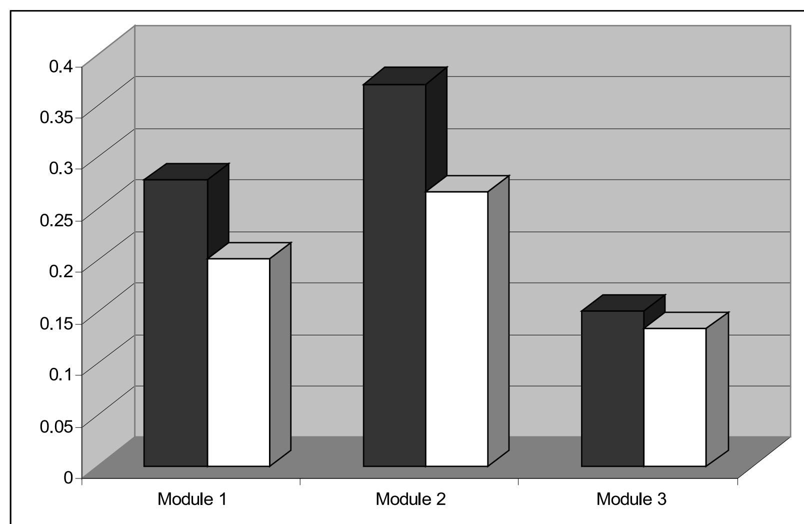

Fig. 6.31: Rococo Instability

more concise. Measurements on MedHCP including generated aspects and ALPH translation files reveal an increase in total LOC by 9%, but this increase is irrelevant to developers as no effort is required on their part in the implementation of these modules. Application size is not reduced after weaving as the crosscutting code still exists in the aspects. This finding has been discovered in other studies on AOP refactored systems [102]. LOC results in AO

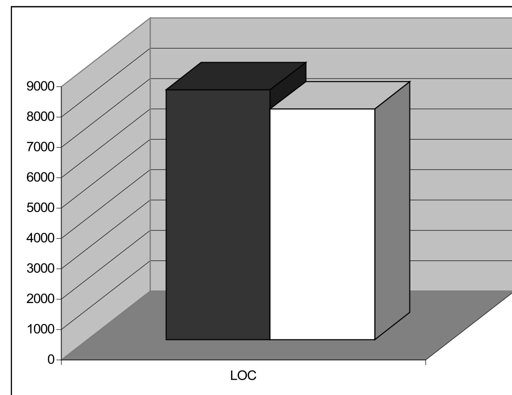

Fig. 6.32: DBay Lines of Code

applications depend on the design of the aspects [169] and as stated earlier, we measure size from the developer's perspective. The LOC metric reveals that the ALPH model reduced the code written by the developer by 13% in the Healthwatcher implementation.

The Rococo mobile application was significantly reduced in size using the ALPH model due to the large volume of discovery code tangled within the base application. This code is modularised by ALPH and results in a 62% reduction in code written by the application developer. This demonstrates the reduction in requirement of domain-specific knowledge on the developers part when using a DSL to programme in a target domain. Figure 6.33 illustrates this result.

6.7.6 Expressiveness

Zipf's law states that a small number of common terms cover the majority of occurences in text. Most other terms occur infrequently or at a low frequency. We measure syntactic expressiveness using a method based on Zipf's law that measures a particular part of speech (POS) within the requirements of a domain [53], exposing the frequency with which terms occur in the domain. This identifies the common domain terms that should be incorporated in a language for that domain. Any language being assessed has its constructs compared to these terms to measure the number of domain-specific requirements that can be syntactically

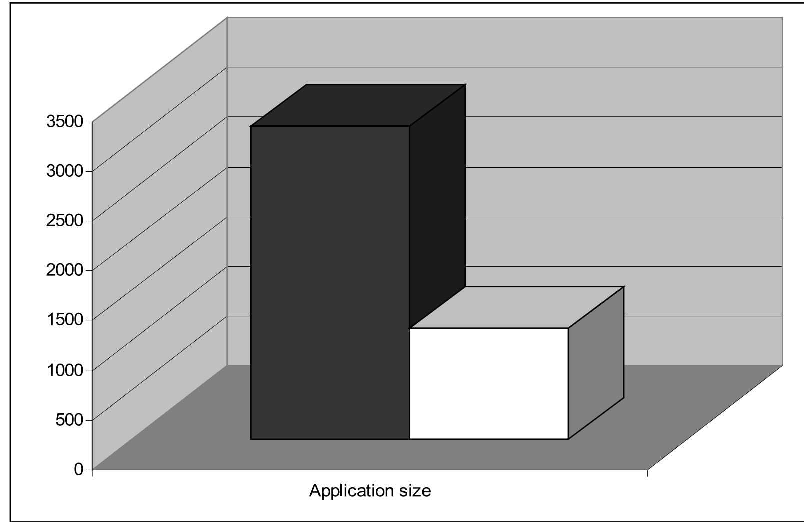

Fig. 6.33: Rococo Lines of Code

expressed by the language. This is then divided by the number of requirements in the domain to achieve the syntactical expressiveness of the language.

An automated POS tagger CLAWS [117] was used to extract verbs from pervasive healthcare application requirements. A list of requirements was compiled from the concerns identified in domain analysis, section 3.6, and from requirements engineering research in the domain [208] [37]. The large set of terms identified included many application-specific verbs relating to the business logic of particular applications. However, as these were not common to many applications they are placed further down the list of frequency in the resulting terms. A set of 29 frequently recurring verbs was considered. Table 6.6 illustrates the most frequently occurring verbs for the pervasive healthcare domain.

Rank (r)	Term	Frequency (f)
1	persist	5
2	send	4
2	receive	4
4	discover	3
5	locate	2
5	connect	2

Table 6.6: Frequency of lexical terms

The syntactic expressiveness for the entire set of requirements is illustrated in figure 6.34 showing the relationship between the word V and the syntactic expressiveness in relation to the domain. It shows that 50% syntactic expressiveness can be achieved when V = 6 i.e. with 20% (6/29) of the verbs detected i.e., the words shown in figure 6.6. This 50% is the target value for a DSL in a domain [53] using the fewest possible constructs. The ALPH model DSL language syntax provides constructs to address 15 of the 29 terms identified in the domain. ALPH only supports 15 constructs as these emerged as reoccurring within the target domain and were also identified as crosscutting. Many of the other verbs identified do not refer to specific behaviour e.g., make, fulfill, ensure and so cannot be easily supported in a DSL. Importantly, it provides constructs for the top 6 most frequently occuring terms enabling it to achieve the 50% SEV using 20% of the terms detected using its syntax. By supporting the most frequently occuring terms the highest proportion of functionality is addressed, in accordance with Zipf's law. Using a GPL, i.e., Java, 0% SEV can be achieved using 100% of its syntax as there is no correlation with its syntax in the requirments of the domain. As this measurement is "per language" the result is valid for all evaluations comparing ALPH with Java.

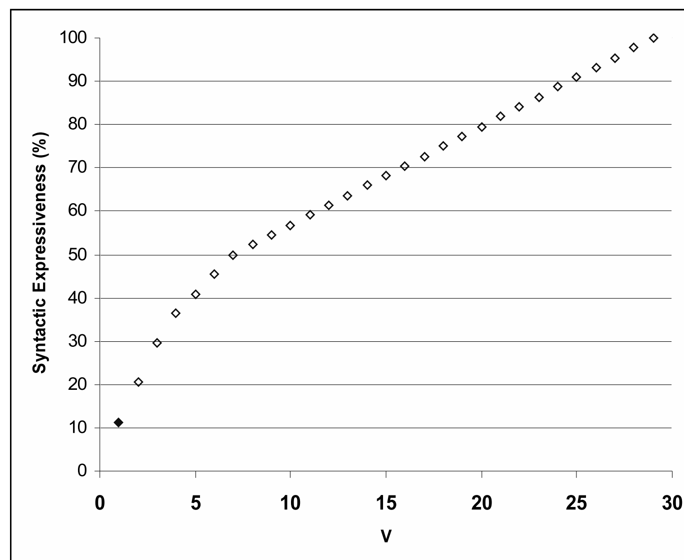

Fig. 6.34: Syntactical Expressiveness

6.7.7 Comparison to Context Toolkit

An earlier ALPH implementation of DBay was also compared to an implementation of the DBay application using the Context Toolkit to modularise context adaptation [190]. The Context Toolkit approach had some negative effect in RFM results due to the introduction of dependencies on its underlying framework classes. Module coupling was increased by the Context Toolkit approach due to the introduction of the widget and supporting classes. The base package's dependencies on external modules, measured by efferent coupling, showed no change in the Context Toolkit approach. The second package's dependencies were increased (as they were using ALPH) as the widget uses additional external modules. Incoming dependencies for the base package remained unchanged by the Context Toolkit approach. The Context Toolkit increased this package's Ca as both Package 1 and Context Toolkit classes make use of the widget in Package 2. The Context Toolkit did not improve the stability of any part of the application. Using the AO approach, the package containing the base code was made 3 times more stable than both the OO approach and the Context Toolkit approach. The Context Toolkit reduced the base code by 15% by removing context handling implementation.

6.8 Discussion

This chapter has described the evaluation of the modularisation and abstraction that can be achieved using the ALPH model in the development of pervasive healthcare applications. Comparative analysis of five applications, consisting of real world scenarios and codebases, were conducted to assess the variations in modularity and abstraction indicators. It was noted that larger codebases observed smaller variations, e.g., Healthwatcher, and that the extent to which the crosscutting functionality affected the base application accounted for very large variations e.g., HL7Browser's majority of HL7 functionality in the original codebase.

Modularity is represented by maintainability, manageability and comprehensibility which are in turn represented by coupling, cohesion and independence. Cohesion was both increased and decreased in applications, due to the number of advices introduced compared to the num-

ber of methods removed. In certain applications, more advices were introduced, accessing varied base elements and decreasing cohesion. Independence and coupling observed an overall beneficial effect from the use of the ALPH model. The base application was made more independent with less outward dependencies. However, as described throughout the description of application-specific results, the use of AOP in the ALPH model introduced dependencies inwards on the base application. This is a recognised source of increase in afferent coupling in AOP approaches [221]. The dependency is mainly due to the use of syntactical elements from the base application in pointcut designators. The dependency here is a trade off for the modularisation that is achieved by the removal of all crosscutting pervasive healthcare functionality out of the base classes. This negative results for Ca, and its subsequent calculation in I, and RFM are all due to these inwards dependencies. This makes the aspects very fragile to the base code dependencies. In the ALPH model, the developer defines where in the base code the pervasive healthcare behaviour should be applied and so is therefore, unlike the code itself, not completely oblivious to these dependencies. RFM increases are also caused by additional functionality added to base applications that did not exist before e.g., more fault tolerance in limited connectivity contingency plans add more dependencies as advice executions are greater than previous method calls.

Abstraction benefits in the form of conciseness and expressiveness were observed in each evaluation application. Codebases were reduced in size by 13% to 62% reducing the amount of domain-specific code in the application and also reducing the requirement for application developers to implement such code. This reduction in knowledge requirement is a fundamental goal of any high-level language providing abstractions for a domain. The ALPH model results indicate that it increases the level of programming abstraction to a high-level for pervasive healthcare application development. Expressiveness was also achieved by the ALPH langauge by providing construct syntax to sufficiently fulfill the requirements of applications in the pervasive healthcare domain.

6.9 Summary

Using the goals, questions and metrics derived using the GQM approach, an overall view of how the collaborative use of AOP and DSL affected modularity and abstraction, and therefore complexity, can be deduced. Table 6.7 summarises the results observed in this chapter.

	LOC	CBM	RFM	LCO	Ce	Ca	I	SEV
DBay	-18%	≤-42%	-13%	-8%	Decrease	Increase	Both	50%(20%V)
HL7Browser	-8%	-27%	+11%	-11%	Decrease	Increase	Decrease	50%(20%V)
MedHCP	-25%	≤-75%	-60%-+25%	+20%-+200%	Decrease	Increase	Decrease	50%(20%V)
Healthwatcher	-13%	≤-10%	≤-20%	-1%	Decrease	Increase	Both	50%(20%V)
Rococo	-62%	≤-52%	2/3(-)	-7%	Decrease	Increase	Decrease	50%(20%V)

Table 6.7: Results Summary

Modularity benefits maintainability, manageability and comprehensibility observed mixed results. Their indicators of coupling, cohesion, and independence produced largely positive results, but incurred negative results in modules where the use of AOP created new inwards dependencies. Coupling was reduced in the majority of modules but was also negatively affected by inward dependencies created by the use of AOP. Cohesion revealed mixed results for the same reason but was reduced in 4 out of 5 applications. Independence indicators instability and response for modules also produced mixed results due to the AOP dependencies. Both were decreased in the majority of applications, but were also subject to increases in modules referenced by aspects. This implies that modularity was increased in the majority of modules, but that the use of AOP also has a negative effect on modularity measurements due to its dependency on base application references. Abstraction indicators were positively affected by the ALPH model across the board. Size was reduced and expressiveness was increased, leading to an overall increase in abstraction. These conclusions are discussed further in chapter 7. This includes a discussion on how the results observed relate to the research question posed in this thesis in section 7.2.

The next chapter discusses the conclusions drawn from the contents of this thesis. A summary of the most significant contributions of this work is presented along with conclusions from the creation, use and evaluation of the ALPH model in the aim of reducing complexity in pervasive healthcare applications. A discussion of research issues that remain open for future work is also presented.

Chapter 7

Conclusions and Future Work

This thesis describes the design and implementation of an approach to reduce complexity in pervasive healthcare applications. The approach is developed as the ALPH model, built on two software engineering approaches to combat complexity; modularity and abstraction. A library of pervasive healthcare concerns and a DSL providing domain-specific constructs allow the customisation of pervasive healthcare functionality to accomodate application-specific requirements. This chapter summarises the achievements and contributions of the work, discusses conclusions made following the completion of the work and describes the potential areas for future work.

7.1 Achievements

Pervasive healthcare, the field concerned with the application of pervasive technology in healthcare applications, requires applications to have access to healthcare information in a mobile, adaptable environment. The integration of mobility and context-awareness behaviour in healthcare applications has proven difficult [139] [185]. Incorporating these concerns increases complexity [185] in new and existing healthcare applications [262]. Mobility and context-awareness are inherently crosscutting, affecting multiple parts of applications and resulting in poor modularity. In the work described in this thesis, this lack of modularisation causes poorly encapsulated concerns and reduced maintainability, manageability and com-

prehensibility in applications resulting in complexity in pervasive healthcare applications. Application developers implementing pervasive healthcare applications are required to use GPL to include domain-specific behaviour. This requires programming at a low-level of abstraction resulting in complex, verbose code with no semantic relation to the domain at which the application is targeted [81] [132]. It also requires the developer to have extensive knowledge of the domain [179]. Programming at an inappropriate level of abstraction for the pervasive healthcare domain is also addressed in the ALPH model to combat complexity in pervasive healthcare applications.

In analysing the state of the art in approaches to modularising and abstracting pervasive healthcare concerns, limitations of existing approaches were highlighted. Firstly, no approach addresses the modularisation of a comprehensive set of pervasive healthcare concerns. Approaches supporting the incorporation of singular concerns and subsets of pervasive healthcare concerns were examined. Many of these approaches provide useful modularisations of pervasive computing concerns using AOP, but no existing approach addresses healthcare specific concerns. Secondly, abstraction has not been investigated for a comprehensive set of concerns within the pervasive healthcare domain by any existing work. DSLs exist for single healthcare concerns and for pervasive computing concerns but no DSL addresses both the pervasive and healthcare specific concerns as required by pervasive healthcare applications. Thirdly, no existing approach exists to address both modularity and abstraction in pervasive healthcare applications. The combined advantages of modularity for crosscutting concerns and abstraction can be achieved by DSALs. Pervasive computing concerns have been addressed by existing DSALs but no existing work addresses healthcare application development using a DSAL. While existing techniques discussed in Chapter 2 can be used to modularise and abstract segments of pervasive healthcare applications, the limitations of these approaches restrict applications from meeting the requirements of applications in the pervasive healthcare domain. To adequately meet the requirements of this emergent domain, a new approach to support pervasive healthcare application development, in a modular way and at an appropriate level of abstraction, is required.

This thesis presented the ALPH model designed to address the issues with modularity

and abstraction in pervasive healthcare applications discussed in Chapter 1 and identified in the limitations of Chapter 2. The ALPH model provides a DSL of domain-specific constructs linked to a library of modular pervasive healthcare concerns. Chapter 3 describes the methodology used in the development of the ALPH model. Domain analysis conducted in the identification of a comprehensive set of crosscutting pervasive healthcare concerns is also described in Chapter 3. Requirements, application literature, case studies and codebases from the pervasive healthcare domain were analysed and the following set of crosscutting pervasive healthcare concerns were identified: Distribution, Communication, Network Roaming, Software Roaming, Service Discovery, Device Discovery, Limited Connectivity, Quality of Service, Device adaptation, Location, HL7 messaging, EHR support and Persistence. The ALPH model addresses complexity caused by difficulties in modularity and inappropriate levels of abstraction in pervasive healthcare application development by employing two software engineering approaches to combat complexity; modularity and abstraction. The crosscutting pervasive heathcare concerns identified in Chapter 3 were implemented using AOP to achieve improvements in modularisation. These modular components were assembled in a library of pervasive healthcare aspects. The design and implementation details of the library of concerns is described in Chapter 4. To enable application developers to programme domain-specific functionality at a higher-level of abstraction in pervasive healthcare applications, the ALPH model provides a DSL called the ALPH language. The language consists of domain-specific constructs that model the modular concerns in the library of aspects. ALPH programs written by application developers are constructed by providing application-specific information as parameters to the domain-specific constructs to define what pervasive healthcare behaviour to include in the application and where to include it. The ALPH compiler, ALPHc, transforms the ALPH program constructs into customised AspectJ implementations by using components from the library of concerns tailored with the application-specific parameter details. The ALPH language, ALPHc and the domain-specific constructs provided are described in Chapter 5.

The evaluation of the ALPH model is described in Chapter 6. Five applications were used in a comparative study to evaluate the variations in modularity and abstraction us-

ing the ALPH model. The GQM approach was used to deduce quantitative-level metrics to measure modularity and abstraction indicators. The comparative studies compared OO implementations with ALPH implementations. Modularity indicators revealed both positive and negative results. Results indicate increases in cohesion due to the encapsulation of crosscutting concerns, and decreases when more advices were created than previous method calls in OO versions. Independence metrics were both increased and decreased indicating that although dependencies were removed from base application components, they were reintroduced by the aspects in the ALPH model. Coupling revealed similar results, with decreases in dependencies from base application on other modules, but increases in the dependencies inwards on the base application. This, as discussed in Chapter 6, is due to the reliance of aspects on base application syntax in pointcut designators. These mixed results in modularity indicators show that while AOP does achieve the clean separation of crosscutting concerns, "modularisation" benefits are balanced by increases in coupling elsewhere. Results measuring conciseness and expressiveness revealed a significant increase in the level of abstraction achieved when using the ALPH model. Applications were significantly reduced in size by the use of domain-specific constructs to introduce pervasive healthcare functionality, reducing the level of domain knowledge required by the developer. The ALPH language was also shown to be more expressive in the domain of pervasive healthcare resulting in semantically intuitive, expressive code. The results for modularity and abstraction reveal a reduction in complexity in pervasive healthcare application development from the developer's point of view, while acknowledging the coupling effect of AOP on oblivious base application code.

In summary, the research presented in this thesis has focused on investigating the provision of a model that supports the incorporation of pervasive healthcare functionality including distributed healthcare, mobility and context-awareness behaviour in applications in a modular manner and at a high-level of abstraction.

The main contributions of this thesis are summarised as:

- A set of pervasive healthcare crosscutting concerns have been identified from the analysis of requirements, applications, literature, case studies and codebases in the domain. Concerns were selected based on their reoccurrence and crosscutting nature. This set of

pervasive healthcare concerns is indicative of functionality that is commonly required by applications, yet complex in its implementation. The identification process in domain analysis is described in chapter 3 and the resulting set of concerns are outlined in section 3.6.

- An overview of approaches with respect to the modularisation and/or abstraction of pervasive healthcare concerns as presented in chapter 2. Approaches are evaluated with a focus on their support for the modularisation (section 2.2 and/or abstraction (section 2.3) of pervasive healthcare concerns as identified through domain analysis.

- A modular design for a set of pervasive healthcare concerns, outlined in chapter 4. These have been implemented using AspectJ and assembled into a library of modular pervasive healthcare aspects. The use of AOP enables the modularisation of crosscutting concerns into modules outside the base application. The library of concerns provides modular implementations for 12 common crosscutting pervasive healthcare concerns described in section 4.3.

- A DSL, the ALPH language, providing a high-level of abstraction for application developers in the pervasive healthcare domain as described in chapter 5. Constructs abstract developers from low-level pervasive healthcare functionality implementation details. This abstraction results in a reduction in the knowledge required by the developer and produces expressive, concise, domain-specific code. The constructs in the ALPH language are outlined in section 5.5.

- The application of a DSAL in the domain of pervasive healthcare had not been investigated previously. Specifically, the application of AOP to healthcare concerns was previously not investigated. Similarly, the application of DSL techniques to pervasive healthcare has had very little investigation.

- A comparative analysis of five applications using the ALPH model to include a subset of pervasive healthcare concerns produces results that illustrate a quantitative study on the use of DSLs, or more specifically a DSAL, and GPLs. The comparative study is

detailed in chapter 6.

7.2 Research Question

This thesis addresses the research question of whether the collaboration of aspect-oriented programming and domain-specific languages significantly reduces complexity in pervasive healthcare applications. Following the formation of the ALPH model and the investigation of its effect on applications in the domain, the findings from this work form a view on the research question under consideration. The use of AOP and DSLs reveal both positive and negative effects on complexity factors in the quantitative evaluation of the research question. The expressiveness achieved by using higher-level constructs has a positive effect on complexity in all applications. Positive effects were also observed on complexity factors including size, independence, coupling and cohesion. The negative results are caused by AOP's references to base application code. These results negate the positive coupling effects observed in the base applications using AOP. From a higher level view, the dependencies have been reorganised rather than actually removed. From a purely quantitative perspective, this points to an indeterminate effect on complexity. However, the reorganisation of the dependencies has a positive effect on all complexity factors in the base applications when viewed separately from the AOP components. This separation increases modularity indicators maintainability, manageability and comprehensibility in base application code. Also, from the developer's perspective, the dependencies are isolated in one clear module. These are all positive effects on complexity from the developer's perspective, adding to the view that the collaboration of DSLs and AOP does significantly reduce complexity in pervasive healthcare application development. The ALPH model DSL provides a set of domain functionality to application developers. This reduces their requirement for domain knowledge and reduces the amount of code they are required to produce. While application size was positively effected, the effect of reduced knowledge requirement is not quantified in the comparative study in this thesis. This also contributes to the view that the use of DSLs can significantly reduce complexity.

These positive features that were not quantified in the scope of this thesis would require further user studies to confirm and enumerate. However, these features and effects were

observed in the creation and use of the ALPH model. Therefore, it is the finding of this thesis that the collaboration of aspect-oriented programming and domain-specific languages does significantly reduce complexity in pervasive healthcare applications. This finding is qualified with the acknowledgement of dependencies created when using AOP, as discussed further in section 6.8. Also in the next section, there is a discussion on the general knowledge that emerged from this study.

7.3 Discussion

The work presented in this thesis aims to reduce complexity in pervasive healthcare applications using modularity and abstraction. From the resulting evaluation, questions arise as to whether the use of AOP increased modularity significantly as some negative results indicated that dependencies introduced at times offset any coupling benefits achieved. It suggests that the dependencies are moved around rather than eliminated. This question has arisen in previous works and the dependencies created by AOP acknowledged [221]. Coupling, cohesion and independence measures can illustrate the advantages of increased and decreased dependencies but cannot fully quantify the beneficial part AOP plays in the ALPH model by allowing generated code to be woven easily into the base application from clearly separated modules. This non-intrusive characteristic of AOP is advantageous in approaches such as the ALPH model approach. The model can be used as easily with existing applications as with building new applications and it can be extended or modified without implications for base applications. This characteristic, referred to as obliviousness, is a topic of debate [247] [130] [103] [73]. Oblivious base applications cause complex aspects and increase the dependence of the aspects on the base code [247], as was observed in the ALPH model evaluation. This tradeoff can be reduced by approaches including the enforcement of design rules in base applications [247] or explicit joinpoints [130], but such approaches impose constraints on base applications. This is not optimal in the use of the ALPH model as a base application may be an existing application with complex refactoring not being a viable option. Therefore the tradeoff of complex, tightly coupled aspects in the ALPH model is tolerated to achieve as much separation of crosscutting pervasive healthcare concerns from the base code as possible.

Another argument is that the base code may be used in isolation as the unit of measurement to evaluate improvements in modularisation using AOP. Taking this approach, our evaluation would reveal improvements in all metrics for coupling, cohesion, independence, showing a vast improvement in modularisation from the external encapsulation of crosscutting concerns. However, this approach is not satisfactory as the references held by aspects need to be accounted for in a conclusive study.

The ALPH model is limited in its scope and by its implementation language. The ALPH model only supports the described concerns in the manner described. Many more concerns exist and many different implementation options exist for the concerns supported. These are out of the scope of this work and require explicit implementation by application developers. The model as implemented provides sufficient functionality to enable this work to investigate the research question by carrying out a proof by construction. As ALPH is a pre-processor to the AspectJ language, it inherits the constraints and scope of this language. For example, the AspectJ joinpoint model considers joinpoints including method or constructor call or executions, class or object initialisation, field read and write access, exception handlers and but does not consider loops, super calls, throws clauses or multiple statements. In the evaluation applications these constraints did not cause problems, but in larger applications it is possible that AspectJ's joinpoint model may not support all requirements. Another limitation of the the ALPH model described in this thesis is its restriction to base applications implemented in the Java language. As the current implementation is based on AspectJ implementations of pervasive healthcare concerns, base applications written in Java are required. To support other base languages, the ALPH model would require implementations in appropriate corresponding aspect languages and the refactoring of the ALPHc compiler.

7.4 Future Work

During the duration of the work presented in this thesis, a number of issues worthy of further investigation were identified. This section outlines the key areas identified for future work. Specifically these relate to the extensibility of the ALPH model and the measurement of semantic expressiveness in languages.

7.4.1 Extensibility

Ideally, DSLs model a particular domain comprehensively. This is a difficult task but an achievable one in a finite, static domain. In reality, very few domains remain static over time. This leaves DSLs susceptible to becoming obsolete. The way to counteract this and to ensure the language is useful in practice is to make it extensible [108].

The ALPH model is extensible in its existing state. It can be extended in two ways. The first is the creation of a new construct by extending the syntax and semantics in the formal definition. The language designer must also extend the aspect library with code to support the new construct. The new construct would then be recognised by the translator and the corresponding functionality included once the definition has been changed and the compiler regenerated. Secondly, construct parameterisation provides a means to customise a construct's behaviour. ALPH constructs use parameters to specify what functionality should be included in a particular application, where it should be included and what application-specific details are to be used. A new option can be added to any construct by defining a new aspect containing the required functionality and extending the constructs definition to recognise the new option as a parameter. Although these illustrate that ALPH can be extended, this process would involve significant programming language development knowledge on the application developer's part. Building an abstraction on the extension process is a possible extension to this work to simplify the process of extending ALPH.

7.4.2 Semantic Expressiveness

In the investigation of measuring expressiveness in languages, semantic expressiveness was identified as an interesting and poorly investigated area in DSLs. Semantic expressiveness is how well a language correctly reflects the domain reality that it represents according to users [90]. It considers how users understand the language and how it relates to domain-specific tasks and entities. This concept is dependent on user interpretation and so necessarily, is a subjective measurement. "Although semantic expressiveness can be theoretically verified it is the user's perception of how well the model helps understanding the underlying reality that determines whether the claimed benefits will be achieved" [207]. Some further investigation

was undertaken and a measurement initially proposed by Dunn and Grabski [90] for determining the perceived semantic expressiveness of accounting systems was investigated and is possible to adapt to programming languages to measure user interpretations. Future user studies comparing a domain-specific task using both a GPL and the ALPH language would add a qualitative result to the evaluation of the expressiveness provided by ALPH.

7.5 Summary

This chapter summarised the motivation for the research undertaken and the most significant achievements of the work presented in this thesis. In particular, it outlined how this work contributed to knowledge in the domains of pervasive healthcare, AOP, DSLs and DSALs. The ALPH model is the first of its kind in supporting pervasive healthcare application development, providing modular separation of domain-specific functionality and a high-level of programming abstraction for application developers in the domain. Comparative analysis showed that a high-level of abstraction eased application development for developers in the domain and that separation of concerns was achieved despite modularity results indicating that AOP offsets many removed coupling dependencies. The chapter concluded with suggestions for future work arising from the research undertaken in relation to this thesis.

Bibliography

[1] Antlr, another tool for language recognition. http://www.antlr.org/.

[2] Apache camel. http://camel.apache.org/.

[3] Centre for pervasive healthcare, dept. of computer science, university of aarhus. http://www.pervasivehealthcare.dk/.

[4] Hapi: Hl7 application programming interface. http://hl7api.sourceforge.net/.

[5] Health level 7 (hl7) international standards organisation. http://www.hl7.org/.

[6] Hl7browser. http://www.nule.org.

[7] Javacc, java compiler compiler. https://javacc.dev.java.net/.

[8] The national healthlink project, health service executive, ireland. http://www.healthlink.ie/.

[9] The openehr foundation. http://www.openehr.org.

[10] Rococo software ltd. http://www.rococosoft.com.

[11] World health organization. in: International classification of functioning, disability and health (icf), world health organization, geneva, switzerland (2001).

[12] Ieee standard for a software quality metrics methodology. *IEEE Std 1061-1992* (Mar 1993).

[13] Fondue software development auction case study, 2001. http://lgl.epfl.ch/research/fondue/case-studies/auction/problem-description.html.

[14] Abhishek, S., and Conway, M. Survey of context aware frameworks - analysis and criticism. Tech. rep., University of North Carolina of Chapel Hill, USA, 2006.

[15] Abowd, G. D., Atkeson, C. G., Hong, J., Long, S., Kooper, R., and Pinkerton, M. Cyberguide: a mobile context-aware tour guide. *Wireless Networks 3* (1997), pp. 421–433.

[16] Abowd, G. D., Dey, A. K., Brown, P. J., Davies, N., Smith, M., and Steggles, P. Towards a better understanding of context and context-awareness. In *HUC '99: Proceedings of the 1st international symposium on Handheld and Ubiquitous Computing* (1999), pp. 304–307.

[17] Ague, T., and Schaeffer, R. Preventing medication errors with new technologies. In *Healthcare Technology* (2007), vol. 4, pp. 84–86.

[18] Aho, A. V., Sethi, R., and Ullman, J. D. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[19] Alvarez, J., Gutierrez, I., and Sicilia, M.-A. Personalization as a cross-cutting concern in web servers: A case study on java servlet technology. In *ACP4IS '07: Proceedings of the Third AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (2004).

[20] Alves, S., Koldehofe, B., Miranda, H., and Taiani, F. Design of a backup network for catastrophe scenarios. In *IWCMC '09: Proceedings of the 2009 International Conference on Wireless Communications and Mobile Computing* (Leizpig, Germany, 2009), pp. 613–617.

[21] Arnold, K., Scheifler, R., Waldo, J., O'Sullivan, B., and Wollrath, A. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[22] Ashbrook, D., and Starner, T. Using gps to learn significant locations and predict movement across multiple users. *Personal Ubiquitous Comput. 7*, 5 (2003), pp. 275–286.

[23] Bächlin, M., Förster, K., and Tröster, G. Swimmaster: a wearable assistant for swimmer. In *Ubicomp '09: Proceedings of the 11th international conference on Ubiquitous computing* (Orlando, USA, 2009), pp. 215–224.

[24] Badrinath, B. R., Acharya, A., and Imieliński, T. Impact of mobility on distributed computations. *SIGOPS Oper. Syst. Rev. 27*, 2 (1993), pp. 15–20.

[25] Baldauf, M., Dustdar, S., and Rosenberg, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing 2*, 4 (2007), pp. 263–277.

[26] Baldwin, C. Y., and Clark, K. B. *Design Rules vol I, The Power of Modularity.* MIT Press, 2001.

[27] Banker, R. D., Datar, S. M., and Zweig, D. Software complexity and maintainability. In *ICIS'89: Proceedings of the International Conference on Information Systems* (Boston, USA., 1989), pp. 247–255.

[28] Bansiya, J., and Davis, C. G. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng. 28*, 1 (2002), pp. 4–17.

[29] Baran, P. On distributed communications networks. *Communications Systems, IEEE transactions on 12*, 1 (1964), pp. 1–9.

[30] Bardram, E. Activity-based computing: support for mobility and collaboration in ubiquitous computing. *Personal Ubiquitous Comput. 9*, 5 (2005), pp. 312–322.

[31] Bardram, J., Kjaer, T. A. K., and Nielsen, C. Supporting local mobility in healthcare by application roaming among heterogeneous devices. In *Proceedings of the 5th International Conference on Human Computer Interaction with Mobile Devices and Services* (Udine, Italy, 2003), pp. 161–176.

[32] Bardram, J. E. Applications of context-aware computing in hospital work – examples and design principles. In *Proceedings of the 2004 ACM Symposium on Applied Computing* (Nicosia, Cyprus, 2004), pp. 1574–1579.

[33] Bardram, J. E. The personal medical unit – a ubiquitous computing infrastructure for personal pervasive healthcare. In *UbiHealth 2004: The 3nd International Workshop on Ubiquitous Computing for Pervasive Healthcare Applications.* 2004.

[34] Bardram, J. E. Pervasive healthcare as a scientific discipline. *Methods of information in medicine 47*, 3 (2008), pp. 178–185.

[35] Bardram, J. E., Baldus, H., and Favela, J. Pervasive computing in hospitals. In *Pervasive Healthcare: Research and Applications of Pervasive Computing in Healthcare.* CRC Press, 2006, pp. 49–78.

[36] Bardram, J. E., and Bossen, C. Mobility work: The spatial dimension of collaboration at a hospital. *Comput. Supported Coop. Work 14*, 2 (2005), pp. 131–160.

[37] Bardram, J. E., and Christensen, H. B. Middleware for pervasive healthcare - a white paper. In *Advanced Topic Workshop Middleware for Mobile Computing* (2001).

[38] Bardram, J. E., Hansen, T. R., Mogensen, M., and Soegaard, M. Experiences from real-world deployment of context-aware technologies in a hospital environment. In *UbiComp '06: International Conference on Ubiquitous Computing* (Orange County, USA, 2006), pp. 369–386.

[39] Bardram, J. E., Kjær, T. K., and Nielsen, C. Mobility in healthcare – reporting on our initial observations and pilot study. Tech. rep., Centre for Pervasive Computing, Aarhus, Denmark, 2003.

[40] Barkuus, L., Barkhuus, L., Dey, A., and Dey, A. Location-based services for mobile telephony: a study of users' privacy concerns. In *In INTERACT' 03: International Conference on Human-Computer Interaction* (Zurich, Switzerland, 2003), pp. 709–712.

[41] Barron, P., and Cahill, V. Yabs:: a domain-specific language for pervasive computing based on stigmergy. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering* (New York, USA, 2006), pp. 285–294.

[42] Basili, V., and Hutchens, D. An empirical study of a syntactic complexity family. *Software Engineering, IEEE Transactions on SE-9*, 6 (Nov. 1983), 664–672.

[43] Basili, V. R., Caldiera, G., and Rombach, H. D. The goal question metric approach. In *Encyclopedia of Software Engineering*. 1994.

[44] Beale, T. The health record-why is it so hard? In *IMIA Yearbook of Medical Informatics 2005: Ubiquitous Health Care Systems.* (2005), H. R. K. C, Ed., pp. 301–304.

[45] Bentley, J. Programming pearls: little languages. *Commun. ACM 29*, 8 (1986), pp. 711–721.

[46] Bergstrand, F. A., and Gabrielsson, M. N. Mobile informatics in a hospital environment: Moving from stationary to mobile supporting services in a medical environment. Master's thesis, University of Goeteborg, Goeteborg, Sweden, 2008.

[47] Bohnenberger, T., Jameson, A., Krüger, A., and Butz, A. Location-aware shopping assistance: Evaluation of a decision-theoretic approach. In *Mobile HCI '02: Proceedings of the 4th International Symposium on Mobile Human-Computer Interaction* (London, UK, 2002), pp. 155–169.

[48] Borah, K. Indus: a new platform for ubiquitous computing. *Ubiquity 2005*, October (2005), pp. 1–1.

[49] Borah, K. Indus: an object oriented language for ubiquitous computing. *SIGPLAN Not. 41*, 2 (2006), pp. 18–24.

[50] Borje, L. *Theoretical analysis of information systems*, 2d ed. ed. Studentlitteratur, Lund,, 1967.

[51] Bowen, J. B. Are current approaches sufficient for measuring software quality? *SIGSOFT Softw. Eng. Notes 3*, 5 (1978), pp. 148–155.

[52] Bowie, J., and Barnett, G. O. Mumps: An economical and efficient time-sharing system for information management. *Computer Programs in Biomedicine 6* (1976), pp. 11–22.

[53] Boyd, S., Zowghi, D., and Farroukh, A. Measuring the expressiveness of a constrained natural language: An empirical study. In *RE '05: Proceedings of the 13th IEEE International Conference on Requirements Engineering* (Washington, USA, 2005), pp. 339–352.

[54] Bricon-Souf, N., and Newman, C. Context awareness in health care: A review. *International Journal of Medical Informatics 76*, 1 (2007), pp. 2–12.

[55] Bruce, D. What makes a good domain-specific language? apostle, and its approach to parallel discrete event simulation, 1997.

[56] Bruntink, M., van Deursen, A., and Tourwe, T. Isolating idiomatic crosscutting concerns. In *ICSM'05: Proceedings of the 21st IEEE International Conference on Software Maintenance* (Sept. 2005), pp. 37–46.

[57] Bulling, A., Roggen, D., and Tröster, G. Eyemote — towards context-aware gaming using eye movements recorded from wearable electrooculography. In *Proceedings of the 2nd International Conference on Fun and Games* (Eindhoven, The Netherlands, 2008), pp. 33–45.

[58] Cassou, D., Bertran, B., Loriant, N., and Consel, C. A generative programming approach to developing pervasive computing systems. In *GPCE '09: Proceedings of the 8th international conference on Generative programming and component engineering* (New York, USA, 2009).

[59] Ceccato, M., and Tonella, P. Adding distribution to existing applications by

means of aspect oriented programming. In *Proceedings of the Fourth IEEE International Workshop on Source Code Analysis and Manipulation* (2004), pp. 107–116.

[60] Ceccato, M., and Tonella, P. Measuring the effects of software aspectization. In *First Workshop on Aspect Reverse Engineering* (2004).

[61] Chalmers, D., and Sloman, M. A Survey of Quality of Service in Mobile Computing Environments. *IEEE Communications Surveys 2*, 2 (April 1999).

[62] Chen, G., and Kotz, D. A survey of context-aware mobile computing research. Tech. rep., Dartmouth College, Hanover, USA, 2000.

[63] Chen, I.-R., DaSilva, L., and Midkiff., S. Editorial: Mobile and pervasive computing. *The Computer Journal 47*, 4 (2004).

[64] Chidamber, S. R., and Kemerer, C. F. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng. 20*, 6 (1994), pp. 476–493.

[65] Choudhri, A., Kagal, L., Joshi, A., Finin, T., and Yesha, Y. Patientservice: electronic patient record redaction and delivery in pervasive environments. In *Procceedings of the 5th International Workshop on Enterprise Networking and Computing in Healthcare Industry, 2003, Healthcom 2003.* (2003), pp. 41–47.

[66] Clarke, L. A., Avrunin, G. A., and Osterweil, L. J. Using software engineering technology to improve the quality of medical processes. In *ICSE Companion '08: Companion of the 30th international conference on Software engineering* (Leizpig, Germany, 2008), pp. 889–898.

[67] Clarke, S., and Baniassad, E. *Aspect-Oriented Analysis and Design, The Theme Approach.* Addison-Wesley Professional, 2005.

[68] Cleaveland, J. C. Building application generators. *IEEE Software 5*, 4 (1988), pp. 25–33.

[69] Colyer, A. Managing complexity in middleware. In *ACP4IS '03: The Second AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software* (2003).

[70] Consel, C., Hamdi, H., Réveillère, L., Singaravelu, L., Yu, H., and Pu, C. Spidle: a dsl approach to specifying streaming applications. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering* (Erfurt, Germany, 2003), pp. 1–17.

[71] COSTEA, A. On measuring software complexity. *Journal of Applied Quantitative Methods 2* (2007).

[72] Curtis, B. Measurement and experimentation in software engineering. *Proceedings of the IEEE 68*, 9 (Sept. 1980), 1144–1157.

[73] Curtis, C., and T., L. G. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Tech. rep., 2003.

[74] D., G., J., K., T., H., and Dowd, B. Medical groups' adoption of electronic health records and information systems. *Health Affair 24*, 5 (2005), 1323–1333.

[75] Dantas, A., , Dantas, A., and Borba, P. Adaptability aspects: An architectural pattern for structuring adaptive applications with aspects. In *Third Latin American Conference on Pattern Languages of Programming* (2003).

[76] Davies, N., Cheverst, K., Mitchell, K., and Efrat, A. Using and determining location in a context-sensitive tour guide. *Computer 34*, 8 (2001), pp. 35–41.

[77] Davis, J. S., and LeBlanc, R. J. A study of the applicability of complexity measures. *IEEE Trans. Softw. Eng. 14*, 9 (1988), pp. 1366–1372.

[78] Dedecker, J. Ambient-oriented programming in ambienttalk: combining mobile hardware with simplicity and expressiveness. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (San Diego, USA, 2005), ACM, pp. 196–197.

[79] den Haan, J. Dsl development: 7 recommendations for domain specific language design based on domain-driven design. http://www.theenterprisearchitect.eu/.

[80] Denning, P. B74-20 systematic programming: An introduction. *Computers, IEEE Transactions on C-23*, 2 (Feb. 1974), pp. 222–223.

[81] Deursen, A. v., Klint, P., and Visser, J. Domain-specific languages: An annotated bibliography. *ACM SIGPLAN Notices 35*, 6 (2000), pp. 26–36.

[82] Dijkstra, E. W. Chapter i: Notes on structured programming. *Academic Press Ltd, London, UK* (1972), 1–82.

[83] Dijkstra, E. W. On the role of scientific thought. In *Selected Writings on Computing: A Personal Perspective*. Springer-Verlag, 1982, pp. 60–66.

[84] Dineen, M. Real-time display of dublin traffic information on the web. Master's thesis, University of Dublin, Trinity College Dublin, 2000.

[85] Downing, T. B. *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Inc., Foster City, CA, USA, 1998.

[86] Driver, C., and Clarke, S. Hermes: A software framework for mobile, context-aware trails. In *Workshop on Computer Support for Human Tasks and Activities at Pervasive 2004* (apr 2004).

[87] Driver, C., and Clarke, S. An application framework for mobile, context-aware trails. *Pervasive Mob. Comput. 4*, 5 (2008), pp. 719–736.

[88] Driver, C., Linehan, E., Spence, M., Tsang, S. L., Chan, L., and Clarke, S. Facilitating dynamic schedules for healthcare professionals. In *Pervasive Health Conference and Workshops, 2006* (2006), pp. 1–10.

[89] Ducasse, S., Girba, T., and Kuhn, A. Distribution map. In *ICSM '06: 22nd IEEE International Conference on Software Maintenance* (2006), pp. 203–212.

[90] Dunn, C. L., and Grabski, S. V. Perceived semantic expressiveness of accounting systems and task accuracy effects. *International Journal of Accounting Information Systems* (2000).

[91] Eaddy, M., Aho, A., and Murphy, G. Identifying, assigning, and quantifying crosscutting concerns. In *ACoM '07: First International Workshop on Assessment of Contemporary Modularization Techniques* (2007).

[92] Eichelberg, M., Aden, T., Riesmeier, J., Dogac, A., and Laleci, G. B. A survey and analysis of electronic healthcare record standards. *ACM Comput. Surv. 37*, 4 (2005), pp. 277–315.

[93] Elmasri, R., and Navathe, S. B. *Fundamentals of Database Systems (5th Edition).* Addison Wesley, March 2006.

[94] Fabbrini, F., Fusani, M., Gervasi, V., Gnesi, S., and Ruggieri, S. Achieving quality in natural language requirements, 1998.

[95] Fabry, J. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code.* PhD thesis, Vrije Universiteit Brussel, Vakgroep Informatica, Laboratorium voor Programmeerkunde (PROG), 2005.

[96] Fabry, J., Tanter, E., and DHondt, T. Kala: Kernel aspect language for advanced transactions. *Sci. Comput. Program. 71*, 3 (2008), pp. 165–180.

[97] Feng, L., Chen, D., and Torngren, M. Self configuration of dependent tasks for dynamically reconfigurable automotive embedded systems. In *47th IEEE Conference on Decision and Control, 2008. CDC 2008.* (Dec. 2008), pp. 3737–3742.

[98] Fenton, N. E., and Neil, M. Software metrics: roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering* (New York, NY, USA, 2000), ACM Press, pp. 357–370.

[99] Ferguson, P., and Huston, G. *Quality of service: delivering QoS on the Internet and in corporate networks.* John Wiley & Sons, Inc., New York, NY, USA, 1998.

[100] Ferguson, T. From patients to end users (editorial). *British Medical Journal* (2009).

[101] Figueiredo, E., Garcia, R., Kulesza, U., and Lucena, C. Assessing aspect-oriented artifacts: Towards a toolsupported quantitative method. In *QAOOSE '0: Workshop on Quantitative Approaches in Object-Oriented Software Engineering* (2005).

[102] Filho, L. C., Garcia, A., and Rubira, C. M. F. A quantitative study on the aspectization of exception handling. In *In Proceedings of ECOOP'2005 Workshop on Exception Handling in OO Systems* (2005).

[103] Filman, R. E., and Friedman, D. P. Aspect-oriented programming is quantification and obliviousness. In *Aspect-Oriented Software Development*, R. E. Filman, T. Elrad, S. Clarke, and M. Akşit, Eds. Addison-Wesley, Boston, 2005, pp. 21–35.

[104] Fontelo, P. A., and Chismar, W. G. Introduction to minitrack: Pdas, hand-held devices and wireless healthcare environments. In *Proceedings of the 37th Hawaii International Conference on System Sciences* (2004).

[105] Forgy, C. L. Rete: a fast algorithm for the many pattern/many object pattern match problem. In *Expert systems: a software methodology for modern applications.* IEEE Computer Society Press, Los Alamitos, CA, USA, 1990, pp. 324–341.

[106] Forman, G. H., and Zahorjan, J. The challenges of mobile computing. *Computer 27*, 4 (1994), 38–47.

[107] France, R., and Rumpe, B. Model-driven development of complex software: A research roadmap. In *FOSE '07: Future of Software Engineering* (May 2007), pp. 37–54.

[108] Freeman, S., and Pryce, N. Evolving an embedded domain-specific language in java. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (San Diego, USA, 2006), pp. 855–865.

[109] Fritsch, S., Senart, A., and Clarke, S. Addressing dynamic contextual adaptation with a domain-specific language. In *SEPCASE '07: Proceedings of the 1st In-*

ternational Workshop on Software Engineering for Pervasive Computing Applications, Systems, and Environments (2007).

[110] Fuentes, L., and Gamez, N. A feature model of an aspect-oriented middleware family for pervasive systems. In *NAOMI '08: Proceedings of the 2008 workshop on Next generation aspect oriented middleware* (2008).

[111] Fuentes, L., and Jiménez, D. An aspect-oriented ambient intelligence middleware platform. In *MPAC '05: Proceedings of the 3rd international workshop on Middleware for pervasive and ad-hoc computing* (2005).

[112] Fuentes, L., Jiménez, D., and Pinto, M. Development of ambient intelligence applications using components and aspects. *J. UCS 12*, 3 (2006), 236–251.

[113] Fuentes, L., and Jimnez, D. Combining components, aspects, domain specific languages and product lines for ambient intelligent application development. In *Advances in Pervasive Computing, Proceedings of Pervasive 2006* (2006), vol. 207.

[114] Fuggetta, A., Picco, G. P., and Vigna, G. Understanding code mobility. *IEEE Transactions on Software Engineering 24*, 5 (1998), 342–361.

[115] Furfaro, A., Nigro, L., and Pupo, F. Aspect oriented programming using actors. In *Proceedings. 22nd International Conference on Distributed Computing Systems Workshops* (2002), pp. 493–498.

[116] Gaddah, A., and Kunz, T. A survey of middleware paradigms for mobile computing. Tech. rep., Carleton University, 2003.

[117] Garside, R., Leech, G., and Sampson, G. *The CLAWS word-tagging system.* Longman Group United Kingdom Limited, 1987.

[118] Goldrei, S. E. C. The design, implementation and use of domain specific languages. Tech. rep., School of Information Technologies, University of Sydney, 2004.

[119] Gray, J., and Karsai, G. An examination of dsls for concisely representing model traversals and transformations. In *HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences* (Hawaii, USA, 2003).

[120] Greenwood, P. e. a. On the contributions of an end-to-end aosd testbed. In *Workshops in Aspect-Oriented Requirements Engineering and Architecture Design* (2007).

[121] Hachichi, A., Thomas, G., Martin, C., Folliot, B., Patarin, S., and Bologna, D.-U. D. A generic language for dynamic adaptation. In *In 11th International European Parallel Processing Conference (EuroPar)* (2005).

[122] Hankerson, M. B. Towards a taxonomy of aspect-oriented programming. Master's thesis, East Tennessee State University, USA, 2003.

[123] Hansen, T. R., Bardram, J. E., and Soegaard, M. Moving out of the lab: Deploying pervasive technologies in a hospital. *IEEE Pervasive Computing 5*, 3 (2006), 24–31.

[124] Harrison, W., and Ossher, H. Subject-oriented programming: a critique of pure objects. In *OOPSLA '93: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications* (Anaheim, USA, 1993), pp. 411–428.

[125] Haugen, ., and Mohagheghi, P. A multi-dimensional framework for characterizing domain specific languages. In *DSM 07: 7th OOPSLA Workshop on Domain-Specific Modeling* (2007).

[126] Heathfield, H., Pitty, D., and Hanka, R. Evaluating information technology in health care: barriers and challenges. *British Medical Journal 316*, 7149 (June 1998), 1959–1961.

[127] Henricksen, K., and Robinson, R. A survey of middleware for sensor networks: state-of-the-art and future directions. In *MidSens '06: Proceedings of the international workshop on Middleware for sensor networks* (2006).

[128] Herndon, R.M., J., and Berzins, V. The realizable benefits of a language prototyping language. *Software Engineering, IEEE Transactions on 14*, 6 (Jun 1988), 803–809.

[129] Hoersch, W. L., and Lopes, C. V. Separation of concerns. Tech. rep., College of Computer Science, Northeastern University, USA, 1995.

[130] Hoffman, K., and Eugster, P. Towards reusable components with aspects: an empirical study on modularity and obliviousness. In *ICSE '08: Proceedings of the 30th international conference on Software engineering* (Leipzig, Germany, 2008), pp. 91–100.

[131] Hohenstein, U. Using aspect-orientation to add persistency to applications. In *BTW, Datenbanksysteme in Business, Technologie und Web* (2005), pp. 235–244.

[132] Hudak, P. Building domain-specific embedded languages. *ACM Comput. Surv.*, 196.

[133] Hudak, P. Modular domain specific languages and tools. In *ICSR '98: Proceedings of the 5th International Conference on Software Reuse* (Victory B.C., Canada, 1998).

[134] Iakovidis, I. Towards personal health record: current situation, obstacles and trends in implementation of electronic healthcare record in europe. *International journal of medical informatics 52*, 1 (1998), 105–115.

[135] Ibrahim, N., and Mouel, F. L. A survey on service composition middleware in pervasive environments. *International Journal of Computer Science Issues 1* (2009), 1–12.

[136] Jirani, J., and Pearson, C. A survey of context aware and service discovery mobile computing middleware. Tech. rep., Christopher Pearson University of Florida, 2008.

[137] Johanson, B., Fox, A., and Winograd, T. The interactive workspaces project: experiences with ubiquitous computing rooms. *Pervasive Computing, IEEE 1*, 2 (2002), 67–74.

[138] John, G. K., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., marc Loingtier, J., and Irwin, J. Aspect-oriented programming. In *Proceedings of 11th European Conference on Object-Oriented Programming* (Jyvaeskylae, Finland, 1997), pp. 220–242.

[139] Jorgensen, J., and Bossen, C. Requirements engineering for a pervasive health care system. In *Proceedings of the 11th IEEE International Requirements Engineering Conference, 2003* (Sept. 2003), pp. 55–64.

[140] Jorgensen, J., and Bossen, C. Executable use cases: requirements for a pervasive health care system. *Software, IEEE 21*, 2 (March-April 2004), 34–41.

[141] Jørgensen, J. B. Coloured petri nets in development of a pervasive health care system. In *ICATPN '03: International Conference on Applications and Theory of Petri Nets* (Eindhoven, The Netherlands, 2003), pp. 256–275.

[142] Jouve, W., Lancia, J., Palix, N., Consel, C., and Lawall, J. High-level programming support for robust pervasive computing applications. In *PerCom 2008: Sixth Annual IEEE International Conference on Pervasive Computing and Communications* (Hongkong, China, March 2008), pp. 252–255.

[143] Katehakis, D. G., and Tsiknakis, M. Electronic health record (ehr). In *Wiley Encyclopedia of Biomedical Engineering*, M. Akay, Ed. John Wiley and Sons Inc, 2006.

[144] Kemerer, C. Software complexity and software maintenance: A survey of empirical research. *Annals of Software Engineering 1*, 1 (December 1995), 1–22.

[145] Khoshgoftaar, T., and Munson, J. Predicting software development errors using software complexity metrics. *Journal on Selected Areas in Communications 8*, 2 (1990), 253–261.

[146] Kiczales, G. Aop: Going beyond objects for better separation of concerns in design and implementation.

[147] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. Getting started with aspectj. *Communications of the ACM 44* (2001), 59–65.

[148] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. G. An overview of aspectj. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming* (Budapest, Hungary, 2001), pp. 327–353.

[149] Killijian, M.-O., Cunningham, R., Meier, R., Mazare, L., and Cahill, V. Towards group communication for mobile participants. In *POMC '01: ACM Workshop on Principles of Mobile Computing* (2001).

[150] Kjaer, K. E. A survey of context-aware middleware. In *SE'07: Proceedings of the 25th conference on IASTED International Multi-Conference* (Innsbruck, Austria, 2007), pp. 148–155.

[151] Knuth, D. E. backus normal form vs. backus naur form. *Commun. ACM 7*, 12 (1964), 735–736.

[152] Ko, L.-F., Lin, J.-C., Chen, C.-H., Chang, J.-S., Lai, F., Hsu, K.-P., Yang, T.-H., Cheng, P.-H., Wen, C.-C., Chen, J.-L., and Hsieh, S.-L. Hl7 middleware framework for healthcare information system. In *HEALTHCOM' 06: 8th International Conference on Health Networking, Applications and Services* (2006), pp. 152–156.

[153] Korhonen, I., and Bardram, J. E. Guest editorial introduction to the special section on pervasive healthcare. *IEEE Transactions on Information Technology in Biomedicine 8*, 3 (2004).

[154] Kramer, S., and Kaindl, H. Coupling and cohesion metrics for knowledge-based systems using frames and rules. *ACM Trans. Softw. Eng. Methodol. 13*, 3 (2004), 332–358.

[155] Kristoffersent, S., and Ljungberg, F. Representing modalities in mobile com-

puting - a model of it use in mobile settings. In *MC'98: Proceeding of Interactive Applications of Mobile Computing* (1998).

[156] Krueger, C. W. Software reuse. *ACM Comput. Surv. 24*, 2 (1992), 131–183.

[157] Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., von Staa, A., and Lucena, C. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM '06: 22nd IEEE International Conference on Software Maintenance* (Philadelphia, USA, 2006), pp. 223–233.

[158] L., F., D., J., and M., P. An ambient intelligent language for dynamic adaptation. In *Workshop on Object Technology for Ambient Intelligence* (2005).

[159] Ladd, D. A., and Ramming, J. C. Two application languages in software production. In *VHLLS'94: Very High Level Languages Symposium Proceedings* (1994).

[160] Lee, C., and Helal, S. Protocols for service discovery in dynamic and mobile networks. *International Journal of Computing Research 11* (2002), 1–12.

[161] Lee, K.-W., Cho, E.-S., and Kim, H. An eca rule-based task programming language for ubiquitous environments. In *ICACT '06: The 8th International Conference on Advanced Communication Technology* (Phoenix Park, Republic of Korea, 2006).

[162] Lee, K.-W., and Kim, H. A new programming language for ubiquitous applications. In *Workshop on Building Software for Pervasive Computing* (2005).

[163] Leite, J., Rossi, G., Balaguer, F., Maiorana, V., Kaplan, G., Hadad, G., and Oliveros, A. Enhancing a requirements baseline with scenarios. In *Proceedings of the Third IEEE International Symposium on Requirements Engineering* (1997), pp. 44–53.

[164] Levine, J., Mason, T., and Brown, D. *Lex & yacc, 2nd edition.* O'Reilly, 1992.

[165] Li, W., and Henry, S. Object-oriented metrics that predict maintainability. *J. Syst. Softw. 23*, 2 (1993), 111–122.

[166] Li, Y., Yang, H., and Chu, W. Fusing ambiguous domain knowledge slices in a reverse engineering process. In *APSEC '00: Proceedings of the Seventh Asia-PacificSoftware Engineering Conference* (Singapore, 2000), pp. 266–273.

[167] Li, Y., Yang, H., and Chu, W. Generating linkage between source code and evolvable domain knowledge for the ease of software evolution. In *Proceedings of the International Symposium on Principles of Software Evolution* (2000), pp. 196–205.

[168] Lindland, O. I., Sindre, G., and Solvberg, A. Understanding quality in conceptual modeling. *IEEE Softw. 11*, 2 (1994), 42–49.

[169] Lippert, M., and Lopes, C. V. A study on exception detection and handling using aspect-oriented programming. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering* (Limerick, Ireland, 2000), pp. 418–427.

[170] Lobato, C., Garcia, A., Romanovsky, A., Sant'Anna, C., and Kulesza, U. Mobility as an aspect: The AspectM framework. In *WASP '04: Proceedings of the 1st Brazilian Workshop on Aspect-Oriented Software Development* (2004).

[171] Lopes., C. V. *D: A Language Framework For Distributed Programming.* PhD thesis, College of Computer Science of Northeastern University, USA, 1997.

[172] Lopes, C. V., and Kiczales, G. Recent developments in aspectj. In *ECOOP'98 Workshop Reader* (1998), Springer Verlag, pp. 398–401.

[173] Loughran N., e. a. Survey of aspect-oriented middleware. Tech. rep., 2005. AOSD-Europe Deliverable D8.

[174] Lu, Y., Xiao, Y., Sears, A., and Jacko, J. A review and a framework of handheld computer adoption in healthcare. *International Journal of Medical Informatics 74*, 5 (June 2005), 409–422.

[175] Martin, R. Oo design quality metrics-an analysis of dependencies. In *Workshop Pragmatic and Theoretical Directions in Object-Oriented Software Metrics* (1994).

[176] Mathiassen, L., and Stage, J. Complexity and uncertainty in software design. In *CompEuro '90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering* (May 1990), pp. 482–489.

[177] McCune, J. M., Perrig, A., and Reiter, M. K. Seeing is believing; using camera phones for human verifiable authentication. *International Journal of Security and Networks 4*, 1/2 (2009), 43–56.

[178] Mercadal, J., Palix, N., Consel, C., and Lawall, J. L. Pantaxou: a domain-specific language for developing safe coordination services. In *GPCE '08: Proceedings of the 7th international conference on Generative programming and component engineering* (Nashville, USA, 2008), pp. 149–160.

[179] Mernik, M., Heering, J., and Sloane, A. M. When and how to develop domain-specific languages. *ACM Comput. Surv. 37*, 4 (2005), 316–344.

[180] Mernik, M., Lenic, M., Avdicausevic, E., and Zumer, V. Compiler/interpreter generator system lisa. In *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on* (Hawaii, USA, 2000).

[181] Meystre, S., and Mller, H. Open source software in the biomedical domain: Electronic health records and other useful applications. *Swiss Med. Inform 55* (2005), 3–15.

[182] Michahelles, F., and Schiele, B. Sensing and monitoring professional skiers. *IEEE Pervasive Computing 4*, 3 (2005), 40–46.

[183] Mildenberger, P., Eichelberg, M., and Martin, E. Introduction to the dicom standard. *JournalEuropean Radiology 12*, 4 (2002), 920–927.

[184] Misra, S., and Misra, A. Evaluating cognitive complexity measure with weyuker properties. In *Proceedings of the Third IEEE International Conference on Cognitive Informatics* (2004), pp. 103–108.

[185] Mu noz, M. A., Rodríguez, M., Favela, J., Martinez-Garcia, A. I., and González, V. M. Context-aware mobile communication in hospitals. *Computer 36*, 9 (2003), 38–46.

[186] Munnelly, J., and Clarke, S. Alph: a domain-specific language for crosscutting pervasive healthcare concerns. In *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages* (2007).

[187] Munnelly, J., and Clarke, S. A domain-specific language for ubiquitous healthcare. In *Third International Conference on Pervasive Computing and Applications, 2008. ICPCA 2008* (Oct. 2008), vol. 2, pp. 757–762.

[188] Munnelly, J., and Clarke, S. Infrastructure for ubiquitous computing: improving quality with modularisation. In *ACP4IS '08: Proceedings of the 2008 AOSD workshop on Aspects, components, and patterns for infrastructure software* (2008), pp. 1–7.

[189] Munnelly, J., and Clarke, S. Hl7 healthcare information management using aspect-oriented programming. In *CBMS '00: 22nd IEEE International Symposium on Computer-Based Medical Systems* (Aug. 2009), pp. 1–4.

[190] Munnelly, J., Fritsch, S., and Clarke, S. An aspect-oriented approach to the modularisation of context. In *PerCom '07: Fifth Annual IEEE International Conference on Pervasive Computing and Communications* (New York, USA 2007), pp. 114–124.

[191] Munoz, M. A., Rodriguez, M., Favela, J., Martinez-Garcia, A. I., and Gonzalez, V. M. Context-aware mobile communication in hospitals. *Computer 36*, 9 (2003), 38–46.

[192] Muras, J. A., Cahill, V., and Stokes, E. K. A taxonomy of pervasive healthcare systems. In *Pervasive Health Conference and Workshops, 2006* (29 2006-Dec. 1 2006), pp. 1–10.

[193] N, S., and H., W. Editorial: pervasive healthcare. selected papers from the pervasive healthcare 2008 conference. *Methods of information in medicine 47* (2008), 175–177.

[194] Nasr, E., McDermid, J., and Bernat, G. A technique for managing complexity of use cases for large complex embedded systems. *IEEE International Symposium on Object-Oriented Real-Time Distributed Computing 0* (2002), 0225.

[195] Navarro, L. D. B., Südholt, M., Vanderperren, W., De Fraine, B., and Suvée, D. Explicitly distributed aop using awed. In *AOSD '06: Proceedings of the 5th international conference on Aspect-oriented software development* (Bonn, Germany, 2006), pp. 51–62.

[196] Olla, P., and Jan, J. *Personal Health Records Systems Go Mobile.* Mobile Health Solutions for Biomedical Applications. IGI global, Medical Information Science Reference, April 2009, ch. II.

[197] O'Neill, J. T. Computer programming languages for health care. *Proceeding of the Annual Symposium Computer Applied Medical Care* (1979), 304311.

[198] Orwat, C., Graefe, A., and Faulwasser, T. Towards pervasive computing in health care – A literature review. *BMC Medical Informatics and Decision Making 8*, 1 (2008), 26.

[199] Ossher, H., and Tarr, P. L. Operation-level composition: A case in (join) point. In *ECOOP '98: Workshop ion on Object-Oriented Technology* (1998), pp. 406–409.

[200] Ould, M. A. *Testing in software development.* Cambridge University Press, New York, NY, USA, 1987.

[201] Parnas, D. L. On the criteria to be used in decomposing systems into modules. *Communications of the ACM 15* (1972), 1053–1058.

[202] Patig, S. Measuring expressiveness in conceptual modeling. In *CAiSE* (2004), pp. 127–141.

[203] Pawlak, R., Seinturier, L., Duchien, L., and Florin, G. Jac: A flexible solution for aspect-oriented programming in java. In *REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns* (Kyoto, Japan, 2001), pp. 1–24.

[204] Pierre, S. Mobile computing and ubiquitous networking: concepts, technologies and challenges. *Telematics and Informatics 18*, 2-3 (2001), 109 – 131.

[205] Pinto, M., Amor, M., Fuentes, L., and Troya, J. M. Supporting heterogeneous users in collaborative virtual environments using aop. In *CooplS '01: Proceedings of the 9th International Conference on Cooperative Information Systems* (Trento, Italy, 2001), pp. 226–238.

[206] Pinto, M., Fuentes, L., and Troya, J. M. Daop-adl: an architecture description language for dynamic component and aspect-based development. In *GPCE '03: Proceedings of the 2nd international conference on Generative programming and component engineering* (Erfurt, Germany, 2003), pp. 118–137.

[207] Poels, Maes, P. Construction and pre-test of a semantic expressiveness measure for conceptual models. Working Papers of Faculty of Economics and Business Administration, 2004.

[208] Raatikainen, K., Christensen, H. B., and Nakajima, T. Application requirements for middleware for mobile and pervasive systems. *SIGMOBILE Mob. Comput. Commun. Rev. 6*, 4 (2002), 16–24.

[209] Raghupathi, W., and Tan, J. Strategic it applications in health care. *Commun. ACM 45*, 12 (2002), 56–61.

[210] Ranganathan, A., Chetan, S., Al-Muhtadi, J., Campbell, R. H., and Mickunas, M. D. Olympus: A high-level programming model for pervasive computing environments. *Percom '05: IEEE International Conference on Pervasive Computing and Communications 0* (2005), 7–16.

[211] Rao, B., and Minakakis, L. Evolution of mobile location-based services. *Commun. ACM 46*, 12 (2003), 61–65.

[212] Rashid, A., and Chitchyan, R. Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (Boston, USA, 2003), pp. 120–129.

[213] Rashid, A., and Kortuem, G. Adaptation as an aspect in pervasive computing. In *Workshop on Building Software for Pervasive Computing at* (2004).

[214] Rebernak, D., Mernik, M., Wu, H., , and Gray, J. Domain-specific aspect languages for modularizing crosscutting concerns in grammar. In *DSAL '06: Proceedings of the 1st workshop on Domain specific aspect languages* (New York, USA, 2006), ACM.

[215] Regio, M., and Greenfield, J. Designing and implementing an hl7 software factory. In *Proceedings of the International Workshop on Software Factories at OOPSLA* (2005).

[216] Reichle, R., Wagner, M., Khan, M. U., Geihs, K., Lorenzo, J., Valla, M., Fra, C., Paspallis, N., and Papadopoulos, G. A. A comprehensive context modeling framework for pervasive computing systems. In *DAIS '08: International Conference on Distributed Applications and Interoperable Systems*. Springer Verlag, Oslo, Norway, 2008, pp. 281–295.

[217] Richard, G. G., and Spencer, M. *Service Discovery Protocols and Programming*. McGraw-Hill Professional, 2001.

[218] Rindfleisch, T. C. Privacy, information technology, and health care. *Commun. ACM 40*, 8 (1997), 92–100.

[219] Rishel, W. Hl7 with corba and ole: software components for healthcare. In *Proceedings of AMIA Annual Fall Symposium* (Aug. 1996), pp. 95–99.

[220] Robinson, J., Wakeman, I., and Chalmers, D. Composing software services in the pervasive computing environment: Languages or apis? *Pervasive Mob. Comput. 4*, 4 (2008), 481–505.

[221] Ronningen, and Steinmoen. Increasing readability with aspect-oriented programming, 2003.

[222] Sadjadi, S. M., and McKinley, P. K. A survey of adaptive middleware. Tech. rep., Department of Computer Science, Michigan State University, 2003.

[223] Saha, D., and Mukherjee, A. Pervasive computing: A paradigm for the 21st century. *Computer 36*, 3 (2003), 25–31.

[224] Sarker, S., and Wells, J. D. Understanding mobile handheld device use and adoption. *Commun. ACM 46*, 12 (2003), 35–40.

[225] Sartipi, K., and Yarmand, M. Standard-based data and service interoperability in ehealth systems. In *ICSM '08: IEEE International Conference on Software Maintenance* (Beijing, China 2008), pp. 187–196.

[226] Satyanarayanan, M. Pervasive computing: vision and challenges. *Personal Communications, IEEE 8*, 4 (Aug 2001), 10–17.

[227] Schantz, R. E., and Schmidt, D. C. Middleware for distributed systems - evolving the common structure for network-centric applications. In *In Encyclopedia of Software Engineering*. John Wiley And Sons, Inc., 2001.

[228] Schilit, B., Adams, N., and Want, R. Context-aware computing applications. In *In Proceedings of the Workshop on Mobile Computing Systems and Applications* (1994).

[229] Schmidt, A., Beigl, M., and w. Gellersen, H. There is more to context than location. *Computers and Graphics 23* (1998), 893–901.

[230] Sendn, M., and Lors, J. L. Aspectual decomposition in the design of a framework for context-aware user interfaces. In *DSOA'05: Proceedings of Actas del Taller de Desarrollo de Software Orientado a Aspectos* (2005).

[231] Sevilla, D., Garcia, J., and Gomez, A. Aspect-oriented programing techniques to support distribution, fault tolerance, and load balancing in the corba-lc component

model. In *NCA '07: Sixth IEEE International Symposium on Network Computing and Applications* (2007), pp. 195–204.

[232] Shepperd, M. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal 3*, 2 (Mar 1988), 30–36.

[233] Shonle, M., Lieberherr, K., and Shah, A. Xaspects: an extensible system for domain-specific aspect languages. In *OOPSLA '03: Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (Anaheim, USA, 2003), pp. 28–37.

[234] Shortliffe, E. The evolution of electronic medical records. *Academy of Medicine 74* (1999), 414–419.

[235] Siau, K., and Shen, Z. Mobile communications and mobile services. *Int. J. Mob. Commun. 1*, 1-2 (2003), 3–14.

[236] Skov, M. B., and Hoegh, R. T. Supporting information access in a hospital ward by a context-aware mobile electronic patient record. *Personal and Ubiquitous Computing 10*, 4 (2006), 205–214.

[237] Slawson, S. E., Justham, L. M., West, A. A., Conway, P. P., Caine, M. P., and Harrison, R. Accelerometer profile recognition of swimming strokes. In *The Engineering of Sport* (2008), Springer, pp. 81–87.

[238] Soares, S., Borba, P., and Laureano, E. Distribution and persistence as aspects. *Softw. Pract. Exper. 36*, 7 (2006), 711–759.

[239] Somers, H. *Computers and Translation: A translator's guide.* John Benjamins(Amsterdam), 2003.

[240] Sommerville, I., and Sawyer, P. *Requirements Engineering: A Good Practice Guide.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[241] Sooriamurthi, R. Introducing abstraction and decomposition to novice programmers. *SIGCSE Bull. 41*, 3 (2009), 196–200.

[242] Soule, P., Carnduff, T., and Lewis, S. A distribution definition language for the automated distribution of java objects. In *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages* (2007).

[243] Sousan, W., Winter, V., Zand, M., and Siy, H. Ertsal: a prototype of a domain-specific aspect language for analysis of embedded real-time systems. In *DSAL '07: Proceedings of the 2nd workshop on Domain specific aspect languages* (2007).

[244] Spackman, K. A., D, P., Campbell, K. E., D, P., Ct, R. A., and (hon, D. S. Snomed rt: A reference terminology for health care. In *J. of the American Medical Informatics Association* (1997), pp. 640–644.

[245] Spinczyk, O., Gal, A., and Schröder-Preikschat, W. Aspectc++: an aspect-oriented extension to the c++ programming language. In *CRPIT '02: Proceedings of the Fortieth International Conference on Tools Pacific* (Darlinghurst, Australia, 2002), pp. 53–60.

[246] Study, H. H. S. E. C. Bridging from paper charts to an ehr at hamilton health sciences.

[247] Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. Information hiding interfaces for aspect-oriented design. In *ESEC '05: Proceedings of the 10th European software engineering conference* (2005), pp. 166–175.

[248] Sutcliffe, A. Scenario-based requirements engineering. In *Proceedings of the 11th IEEE International Requirements Engineering Conference* (2003), pp. 320–329.

[249] Suvée, D., Vanderperren, W., and Jonckers, V. Jasco: an aspect-oriented approach tailored for component based software development. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development* (Boston, USA, 2003), pp. 21–29.

[250] Tanter, É. Aspects of composition in the Reflex AOP kernel. In *Proceedings of the*

5th International Symposium on Software Composition (SC 2006) (Vienna, Austria, 2006), pp. 98–113.

[251] Tanter, É., Gybels, K., Denker, M., and Bergel, A. Context-aware aspects. In *Software Composition* (2006), pp. 227–242.

[252] Tarr, P., Ossher, H., Harrison, W., and Sutton, Jr., S. M. N degrees of separation: multi-dimensional separation of concerns. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Angeles, USA, 1999), pp. 107–119.

[253] Tatsubori, M. Separation of distribution concerns in distributed java programming. In *OOPSLA '01: Addendum to the 2001 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Tampa, USA, 2001).

[254] Tegarden, D. P., Sheetz, S. D., and Monarchi, D. E. A software complexity model of object-oriented systems. *Decis. Support Syst. 13*, 3-4 (1995), 241–262.

[255] Teo, H.-S. An activity-driven model for context-awareness in mobile computing. In *MobileHCI '08: Proceedings of the 10th international conference on Human computer interaction with mobile devices and services* (2008), pp. 545–546.

[256] Thibault, S. *Domain-Specific Languages: Conception, Implementation, and Application.* PhD thesis, University of Rennes, 1998.

[257] Tian, J., and Zelkowitz, M. V. Complexity measure evaluation and selection. *IEEE Transactions on Software Engineering 21*, 8 (1995), 641–650.

[258] Timbermont, S., Adams, B., and Haupt, M. Towards a dsal for object layout in virtual machines. In *DSAL '08: Proceedings of the 2008 AOSD workshop on Domain-specific aspect languages* (2008).

[259] Tsang, S. L. *Supporting Personalised Recommendations in Context-aware Applications.* PhD thesis, University of Dublin, Trinity College, Department of Computer Science, 2009.

[260] Van Cutsem, T., Mostinckx, S., Boix, E., Dedecker, J., and De Meuter, W. Ambienttalk: Object-oriented event-driven programming in mobile ad hoc networks. In *Chilean Society of Computer Science, 2007. SCCC '07. XXVI International Conference of the* (Nov. 2007), pp. 3–12.

[261] van Deursen, A., and Klint, P. Little languages: little maintenance. *Journal of Software Maintenance 10*, 2 (1998), 75–92.

[262] Varshney, U. Pervasive healthcare. *Computer 36*, 12 (2003), 138–140.

[263] Ververidis, C., and Polyzos, G. Service discovery for mobile ad hoc networks: a survey of issues and techniques. *Communications Surveys And Tutorials, IEEE 10*, 3 (Quarter 2008), 30–45.

[264] Vickers, P. An introduction to function point analysis, 2003.

[265] Walker, R. J., Baniassad, E. L. A., and Murphy, G. C. An initial assessment of aspect-oriented programming. In *ICSE '99: Proceedings of the 21st international conference on Software engineering* (Los Angeles, USA, 1999), pp. 120–130.

[266] Wang, B., Bodily, J., and Gupta, S. K. S. Supporting persistent social groups in ubiquitous computing environments using context-aware ephemeral group service. *PerCom '04: Proceedings of the IEEE International Conference on Pervasive Computing and Communications 0* (2004), 287.

[267] Wang, D. K., and Wang, J. K. Towards the distributed processing of mobile software agents. *SIGAPP Applied Computing Review 9*, 2 (2001), 2–5.

[268] Want, R., Schilit, B. N., Adams, N. I., Gold, R., Pedersen, K., Goldberg, D., Ellis, J. R., and Weiser, M. An overview of the parctab ubiquitous computing experiment. *IEEE Personal Communications 2*, 6 (1995), 28–43.

[269] Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P. Scenarios in system development: current practice. *Software, IEEE 15*, 2 (Mar/Apr 1998), 34–45.

[270] WEISER, M. Some computer science issues in ubiquitous computing. *SIGMOBILE Mob. Comput. Commun. Rev. 3*, 3 (1999), 12.

[271] WENONAH, C. B., JAQUES, AND JANAKIRAM, D. Dynocola: Enabling dynamic composition of object behaviour. In *Proceedings of 2nd International Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE) at ECOOP 2005* (2005).

[272] WILCOX, M. *Implementing LDAP.* Wrox Press Ltd., Birmingham, UK, UK, 1999.

[273] WILE, D. Supporting the dsl spectrum. *Journal of Computing and Information Technology 9*, 4 (2001).

[274] WILE, D. S., AND RAMMING, J. C. Guest editorial introduction to the special edition on domain-specific languages. *IEEE Transactions on Software Engineering 25*, 3 (1999).

[275] WILLIAMS, C. A language for generating hl7 reformatting programs. In *Proceedings of the 19th Annual International Conference of the IEEE for Engineering in Medicine and Biology Society* (1997), pp. 945–948.

[276] WIRTH, N. *Compiler construction.* Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1996.

[277] WULF, W. A. Languages and structured programs (1977). In *Current Trends in Programming Methodology*, R. Yeh, Ed. Prentice Hall, 1977.

[278] Y., L., Y., H., AND W., C. A concept-oriented belief revision approach to domain knowledge recovery from source code. 31–52.

[279] Z., S., Z., L. M. L. A., M., L., AND A, L. Infrastructural software requirements of pervasive health care. In *IADIS International Conference on Applied Computing* (Salamanca, Spain, 2007), pp. 557–562.

[280] ZHANG, C., AND JACOBSEN, H.-A. Refactoring middleware with aspects. *IEEE Transactions on Parallel and Distributed Systems 14*, 11 (2003), 1058–1073.

[281] Zimmermann, A., Lorenz, A., and Oppermann, R. An operational definition of context. In *CONTEXT '07: Proceedings of the 6th International and Interdisciplinary Conference* (Roskilde, Denmark, 2007), pp. 558–571.

[282] Zipf, G. K. *Human Behavior and the Principle of Least Effort.* Addison-Wesley (Reading MA), 1949.

Appendix A

Generated Construct Output

```
1 public aspect AConstructorInvocation {
2
3     static ObjectFactoryInterface objectFactory = ObjectFactoryClient.getInstance();
4
5     pointcut constructorInvocation(): call(ServerImpl.new(..)) ;
6
7     ServerImpl around(): constructorInvocation(){
8         ....
9         Class<ServerImpl> c = ServerImpl.class;
10        className = c.getName();
11        ...
12        signature = c.getConstructor().getParameterTypes();
13        ...
14        object =  (ServerImpl) objectFactory.createWrappedObject(className,signature,
              argus);
15        return object;
16    }
17 }
18
19 // Aspect 2
20
21 public aspect AClientCallInterception{
22
23     pointcut methodInvocation(ServerImpl obj):
24     !within(ServerImpl) && call(public * IServer.*(..) throws !(Exception+) ) &&
           target(obj) ;
25
```

```
26      Object around(ServerImpl obj) : methodInvocation(obj){
27          ...
28          Signature sig = thisJoinPointStaticPart.getSignature();
29          Class c = sig.getDeclaringType();
30          rclassName = c.getName();
31          ...
32          MethodSignature msig = (MethodSignature)thisJoinPointStaticPart.getSignature()
              ;
33          signature = msig.getParameterTypes();
34          argus = thisJoinPoint.getArgs();
35          String newMethodName = thisJoinPoint.getSignature().getName().concat("_Remote
              ");
36          Method method = objectFactory.getMethod(className, newMethodName, signature);
37          realObject = obj.getRemoteStub();
38          object = method.invoke(realObject,argus);
39          ...
40          return object;
41      }
42 }
43
44 // Aspect 3
45
46 public aspect AImplementationRemoteServer {
47
48      declare parents : ServerImpl implements IRemoteServer;
49      declare parents : ServerImpl implements IRemoteClass;
50
51      private Remote ServerImpl.remoteStub = null;
52
53      public ServerImpl.new(Remote remoteStub){
54              this.remoteStub = remoteStub;
55      }
56      public Remote ServerImpl.getRemoteStub(){
57          return remoteStub;
58      }
59
60      public void createEmptyRemoteMethods(){
61          ...
62      }
63
64
65 }
```

Listing A.1: Distribution Output

```
1 public aspect DeviceAdaptation {
2     private RuleBasedEngine Rbe = new RuleBasedEngine(  rules.clp  );
3     pointcut deviceConnected(User u) : set(User App.currentUser) && args(u);
4 
5     after(User u) : deviceConnected (u){
6         ...
7                 String kindOfDevice = u.getClass().getSimpleName();
8                     Rbe.createTemplate();
9                     Rbe.assertTemplate(kindOfDevice.toUpperCase());
10                    Rbe.launchRules( );e
11                    Rbe.run();
12                }
13                catch (JessException e) {
14                    e.printStackTrace();
15                }
16        ...
17    }
18 }
```

Listing A.2: AdaptDevice Output

```
1 public aspect HL7Aspect {
2 
3     pointcut createHL7(String patientID, String surname, String firstName, String DOB,
           String address, String Doctor, String admitDate, String dischargeDate) : call
          (void dischargepatient(String, String, String , String , String , String ,
          String, String)) && args(patientID, surname, firstName, DOB, address, Doctor,
          admitDate, dischargeDate);
4 
5 
6     after(String patientID, String surname, String firstName, String DOB, String
          address, String Doctor, String admitDate, String dischargeDate):
          writeDischarge(patientID, surname, firstName, DOB, address, Doctor, admitDate,
           dischargeDate) {
7         createHL7Message(patientID, surname, firstName, DOB, address, Doctor,
              admitDate, dischargeDate);
8     }
9 
```

```
10
11 public void createHL7Message(String patientID, String surname, String firstName,
      String DOB, String address, String Doctor, String admitDate, String dischargeDate)
       throws HL7Exception {
12
13        ...
14            ADT_A01 adt = new ADT_A01();
15            MSH mshSegment = adt.getMSH();
16            mshSegment.getFieldSeparator().setValue("|");
17            mshSegment.getEncodingCharacters().setValue("^~\\&");
18            mshSegment.getDateTimeOfMessage().getTimeOfAnEvent().setValue(
                  dateTimeOfMessage)
19            mshSegment.getMessageType().getMessageType().setValue(ADT_A01);
20            ...
21            Parser pipeParser = new PipeParser();
22            String encodedMessage = pipeParser.encode(adt);
23            Parser xmlParser = new DefaultXMLParser();
24            String encodedXmlMessage = xmlParser.encode(adt);
25 }
26
27    public void parseHL7(String notification)
28        XMLParser xmlParser = new XMLParser(notification);
29        String message =xmlParser.getMessageType();
30        xmlParser.getMessageSections();
31        ...
32    }
33
34 public void displayHL7(String notification) {
35          ...
36  }
37
38 public void sendHL7(String notification) {
39          ...
40  }
41
42 public void receiveHL7(String notification) { }
43          ...
44 }
45 }
```

Listing A.3: HL7 Output

```
1 public aspect LocationAspect {
2  ...
3     pointcut deviceConnected(Device d) : set(Device App.device) && args(d);
4  ...
5     after (Device d) returning() : deviceConnected(d) {
6          ...
7                  this.monitorUserDevice(d);
8                  this.monitorLaunched = true;
9          ...
10    }
11
12    private void monitorUserDevice(Device deviceToMonitor) {
13         this.lm = new LocationMonitor();
14         this.lm.addLocationProducers(deviceToMonitor.getLp());
15    }
16    public void stopMonitoring(){
17         ...
18          if(!lm.getLocationProducers().isEmpty() ){
19          lm.getLocationProducers().removeAllElements();
20          lm.getThreadForMonitoring().interrupt();
21         ...
22         }
23 }
24
25 Aspect 2
26
27 public aspect EnsureViableLocation {
28
29    LocationMonitor lm = new LocationMonitor();
30    LocationLookupService lls = new LocationLookupService();
31    private static String fileAbsolutePath = "safeLocations.txt";
32
33    pointcut validateLocation() : call(void locationMethod());
34
35    void around(): validateLocation(){
36         ...
37         lm.updateCurrentLocation();
38         Double xLoc = new Double(lm.getProperty("X"));
39         Double yLoc = new Double(lm.getProperty("Y"));
40         GPSLocation cloc = new GPSLocation(xLoc.doubleValue(),yLoc.doubleValue());
41         String currentLoc = lls.getLocationFromGPS(cloc);n
```

```
42          ensureLocationIsValid(cloc);
43          ...
44          if (found){
45              proceed();
46          }
47      }
48
49 public void addLocation(String location){
50          ...
51      }
52
53
54      public static boolean ensureLocationIsValid (GPSCoordinates gpsc)throws
            IOException{
55          ...
56              BufferedReader in = new BufferedReader(new FileReader(fileAbsolutePath));
57                  ...
58                  x0 = Double.parseDouble(tok.nextToken());
59                  y0 = Double.parseDouble(tok.nextToken());
60                  lengthArea = Double.parseDouble(tok.nextToken());
61                  widthArea = Double.parseDouble(tok.nextToken());
62                  ...
63                  locationIsSafe =compareWithTheUserLocation(gpsc, x0, y0, lengthArea,
                        widthArea);
64                  ...
65          return locationIsSafe;
66      }
67
68      public static boolean compareWithTheUserLocation(GPSCoordinates userGpsC,double x0
            ,double y0,double lengthArea,double widthArea){
69          ...
70      }
71
72 }
```

Listing A.4: Location Output

```
1
2 Aspect 1
3
4 public aspect LaunchAgentAspect {
5     pointcut startAgent() : execution(public void User.userPage());
```

```
6
7     before(): startAgent(){
8         MobileAgent a = new ImplementedAgent("rmi://192.353.74.78/ClientAgentManager
              ","rmi:// 142.325.54.75/ServerAgentManager");
9         System.out.println("Agent " + a.getName() + " launched");
10    }
11 }
12
13 Aspect 2
14
15 import java.rmi.RemoteException;
16
17 public aspect ServerAgentManagerAspect{
18
19     public static AgentManager RemoteServerImpl.agtManager = null;
20
21     pointcut constructorInvocation(): call(RemoteServerImpl.new(..)) ;
22
23     after(): constructorInvocation(){
24         try{
25             RemoteServerImpl.agtManager= new AgentManager("rmi://142.325.54.75/
                   ServerAgentManager");
26             }
27         catch (RemoteException e){e.printStackTrace();}
28     }
29 }
30
31 Aspect 3
32
33 import java.rmi.RemoteException;
34
35 public aspect ClientAgentManagerAspect{
36
37     public static AgentManager App.agtManager = null;
38
39     pointcut constructorInvocation(): execution(public static void App.welcome());
40
41     before(): constructorInvocation(){
42         try{
43             App.agentManager= new AgentManager("rmi://192.353.74.78/ClientAgentManager
                   ");
44             }
```

```
45          catch (RemoteException e){e.printStackTrace();}
46      }
47
48 }
```

Listing A.5: SoftwareRoaming Output

```
1 public aspect NetworkRoamingAspect{
2
3          pointcut userIsConnected(User u) : set(User Application.currentUser) && args(u
               );
4
5          after (User u) returning() : userIsConnected(u) {
6               ...
7                        NetworkDatabase networkDB = NetworkDatabase.getInstance();
8                        networkDB.LoadDatabaseFromXmlFile("NetworkPropList.xml");
9                        this.monitorNetworks(u);
10              ...
11         }
12         private void monitorNetworks(User currentUser) {
13              this.nwMonitor = new NetworkMonitor(currentUser);
14              this.monitorLaunched = true;
15         }
16
17         public void stopMonitoring(){
18              nwMonitor.getMonitoringThread().interrupt();
19              monitorLaunched = false;
20         }
21
22     }
```

Listing A.6: NetworkRoaming Output

```
1 public aspect PersistenceManager extends PersistInterface{
2        ...
3        pointcut initConnection(): call(Hospital.run(..));
4        pointcut ConstructorCall() : execution(Doctor.new(..));
5        pointcut accessorsAndMutators() : call(* set*(..)) || && (within(Doctor);
6
7        after() : initConnection(){
8              createConnection("MyDatabase", "sun.jdbc.odbc.JbdcOdbcDriver");
9        }
```

```
10
11        after(Doctor aDoctor) : ConstructorCall() && target (aDoctor){
12              aDoctor.uniqueIdentifier = uniqueID;
13              persistObject(aDoctor);
14        }
15
16        after(Doctor aDoctor) : accessorsAndMutators() && target (Doctor){
17           persistObject(aDoctor);
18        }
19
20      public void persistObject(Doctor aDoctor){
21         ...
22          stmt.execute(SQL);
23          }
24
25      public void createConnection(String datastore, String driver)
26          ...
27          Class.forName(driver);
28          String DatastoreName = "jdbc:odbc:"+datastore;
29          Connection conn = DriverManager.getConnection(DatastoreName);
30          ...
31      }
32
33 }
```

Listing A.7: Persist Output

```
1 public aspect QualityOfService {
2
3     AssuranceGroup requiredAssurances = new AssuranceGroup();
4     Vector<NetworkDeterminer> networkDeterminers = new Vector<NetworkDeterminer>(0);
5     Vector<NetworkModel> validNetwork = new Vector<NetworkModel>(0);
6     NetworkModel bestNetwork;
7
8         pointcut assureQualityOfService(User u) :
9         target(u) && call(* User.*Page(..));
10
11    before(User u) : assureQualityOfService(u){
12
13        Signature currentMethodSignature = thisJoinPoint.getSignature();
14        AssuranceReqDB database = AssuranceReqDB.getInstance();
15        database.loadAssuranceFile();
```

```
16          AssuranceGroup asGroup = database.getAssuranceFor(currentMethodSignature);
17            ...
18            setAssurances(asGroup);
19            updateNetworkDeterminers(u);
20            UpdateValidNetwork();
21            determineBestNetwork();
22 ...
23            }
24     }
25
26     public void setAssurances(AssuranceGroup assGrp){
27          ...
28     }
29     public void UpdateValidNetwork(){
30          ...
31     }
32     public void updateNetworkDeterminers(User u){
33         ...
34     }
35     public void determineBestNetwork(){
36         ...
37     }
38 }
```

Listing A.8: QualityOfService Output

```
1 public aspect ServiceDiscoveryAspect {
2
3         public OutgoingServiceDiscoveryRequest output;
4         public DirectorService directory;
5         public String userIP;
6
7         pointcut serviceDiscovery(): call(void *.print(..)) && this(object);
8         pointcut advertiseService():call(Printer.new(..))&& this(object);
9
10        after(Object object): advertiseService(){
11            ...
12            directory.advertise(PRINTING_SERVICE, thisJoinPoint.getThis().getClass()
                  , object);
13
14        }
15
```

```
16          after(Object object): serviceDiscovery(){
17               ...
18               startServiceListener();
19               sendServiceRequest(PRINTING_SERVICE, object);
20          }
21
22
23          public void sendServiceRequest(){
24              ....
25              IP =object.getIP();
26              output = new OutgoingServiceDiscoveryRequest();
27              output.sendObject(new ServiceDiscoveryRequest("PRINTING_SERVICE" +"," +
                    userIP+","));
28          }
29
30           public void startServiceListener(){
31              ...
32              userIP = object.getIP();
33              ServiceDiscoveryListener mL = new ServiceDiscoveryListener(userIP));
34              mL.start();
35           }
36 }
```

Listing A.9: ServiceDiscovery Output

```
1 public aspect DeviceDiscoveryAspect {
2
3    public OutgoingDeviceDiscoveryRequest output;
4    public String IP;
5
6    pointcut deviceDiscovery(): call( Device.new() ) && this(object);
7
8    after(Object object):deviceDiscovery(){
9         startDeviceListener();
10        sendDeviceRequest(object);
11   }
12
13   public void sendDeviceRequest(){
14        ...
15   }
16
17   public void startDeviceListener(){
```

```
18          ...
19
20  }
21
22 }
```

Listing A.10: DeviceDiscovery Output

```
1 public aspect LimitedConnectivityAspect{
2
3    ContigencyPlan contigencyPlan = new PreemptionContigencyPlan();
4    LoggingContigencyPlan logContigencyPlan = new LoggingContigencyPlan();
5
6    pointcut log(): execution(Request makeRequest(..));
7    pointcut handleEvent(): execution(* *.*(..);
8
9    before(): handleEvent(){
10         contigencyPlan.enactContigency();
11   }
12
13     after(): log(){
14         ...
15         logContigencyPlan.enactContigency();
16         ...
17     }
18 }
```

Listing A.11: LimitedConnectivity Output

```
1
2 import java.util.ArrayList;
3 import java.util.LinkedList;
4 import java.util.List;
5 import java.util.Set;
6
7 import org.openehr.rm.*;
8
9 public aspect EHRAspect{
10
11
12 pointcut EHRCreate(String patientID, String Doctor, String String data): call(App.
      glucoseReading (String, String, String)) & args(patientID, Doctor, data);
```

```
13
14 after(String patientID, String Doctor, String String data):EHRCreate(patientID, Doctor
      , data){
15 createObservation(patientID, Doctor, data);
16 createComposition(patientID, Doctor, data);
17 createEHR(patientID, Doctor, data);
18 }
19
20
21
22 public Observation createObservation(..) {
23 String archetypeId = "openEHR-EHR-OBSERVATION.laboratory-glucose.v1";
24 ..
25 Archetyped arch = new Archetyped(new ArchetypeID(archetypeId), "1.1");
26
27 PartyIdentified provider = new PartyIdentified(performer, Doctor
28                                    null);
29 return new Observation(archetypeId, meaning, arch,
30 TestTerminologyAccess.ENGLISH, TestTerminologyAccess.LATIN_1,
31                                    subject, provider, data, termServ);
32          }
33
34
35 public Composition createComposition() {
36 UIDBasedID id = new HierObjectID("1.11.2.3.4.5.0");
37 String archetypeId = "openEHR-EHR-COMPOSITION.encounter.v1";
38 String archetypeNodeId = archetypeId;
39 Data result = new Data(data);
40
41 PartyRef performer = new PartyRef(new HierObjectID("1.3.3.1.2.42.1.199"), "PERSON");
42 PartyProxy composer = new PartyIdentified(performer, Doctor, null);
43  .
44
45 Composition composition = new Composition(id, archetypeNodeId, name, archetypeDetails,
       feederAudit, links, parent, content, language, context, composer, category,
      territory, termServ);
46                    return composition;
47          }
48
49
50 public EHR createEHR() {
51          HierObjectID systemID = new HierObjectID("1.3.3.1.2.42.1.199");
```

```
52          HierObjectID ehrID = new HierObjectID("1.3.3.1.2.42.1.199");
53          DvDateTime timeCreated = new DvDateTime();
54          List<ObjectRef> contributions = new LinkedList<ObjectRef>();
55          HierObjectID id1 = new HierObjectID("sysID002");
56          ObjectRef ehrStatus = new ObjectRef(id1, "namespace","PERSON");
57          ObjectRef directory = null;
58
59          List<ObjectRef> compositions = new LinkedList<ObjectRef>();
60          compositions.add(createComposition());
61          EHR ehr = new EHR(systemID, ehrID, timeCreated, contributions, ehrStatus,
                directory, compositions);
62          return ehr;
63          }
64 }
```

Listing A.12: EHR Output

```
1 public aspect CommunicationAspect {
2
3           public ClientSocketConnection csc;
4
5           pointcut ClientCommunication(): call( Device.new() ) ;
6
7            after():ClientCommunication(){
8                        startListener();
9                        csc = new ClientSocketConnection();
10
11               }
12
13
14               public void startListener(){
15                        Receiver mL = new Receiver ();
16                       mL.start();
17               }
18
19
20 }
21
22
23
24 public aspect CommunicationAspect {
25
```

```
26
27 pointcut ServerCommunication(): call(void App.init(..));
28
29             after():ServerCommunication(){
30 startListener();
31                         ServerSideSocket socket = new ServerSideSocket(249.353.142.87)
                               ;
32                  .
33                }
34
35  .
36
37          public static void starttListener(){
38                  Reciever mL = new Reciever();
39                  mL.start();
40          }
41  .
42          }
43 }
```

Listing A.13: Communication Output

Appendix B

ALPH Language Definition

```
1 options {
2           STATIC = false;
3           JDK_VERSION = "1.5";
4 }
5
6 PARSER_BEGIN(AlphParser)
7 package alph.compiler.parser;
8
9 import java.io.PrintStream;
10 import java.util.ArrayList;
11 import java.util.List;
12
13 import alph.util.Joiner;
14 import alph.compiler.builders.*;
15 import alph.compiler.syntax.*;
16
17 public class AlphParser {
18 }
19
20 PARSER_END(AlphParser)
21
22 SKIP : { " " }
23 SKIP : { < "//" (~["\n","\r"])* (<EOL>)? > }
24 TOKEN : { < EOL : "\n" | "\r" | "\r\n" > }
25 TOKEN : { < INCLUDE : "include" > }
26 TOKEN : { < OPEN_PAREN : "(" > }
27 TOKEN : { < CLOSE_PAREN : ")" > }
```

```
28 TOKEN : { < OPEN_BRACKET : "[" > }
29 TOKEN : { < CLOSE_BRACKET : "]" > }
30 TOKEN : { < AT : "@" > }
31 TOKEN : { < COMMA : "," > }
32 TOKEN : { < SEMICOLON : ";" > }
33 TOKEN : { < DOT_DOT : ".." > } // TOKEN : { < JAVATYPE : "String" | "int" | "void" | "
       boolean" | "double" > }
34 TOKEN : { < #DIGITS : (["0" -"9"])+ > }
35 TOKEN : { < #LETTER : (["a"-"z","A"-"Z"]) > }
36 TOKEN : { < #DIGIT : (["0" -"9"]) > }
37 TOKEN : { < #ALPHNUM_IDENT : ( < LETTER > | < DIGIT > ) ( <LETTER> | <DIGIT> | "." | "
       _" | "*" )* > }
38 TOKEN : { < IDENT : <ALPHNUM_IDENT> > }
39
40
41 List<Include> start(PrintStream printStream) throws NumberFormatException : {
42     List<Include> incls;
43 }
44 {
45     (
46         incls = Includes()
47         // <EOL>
48         { return incls; }
49     )
50     <EOF>
51 }
52
53 List<Include> Includes() : {
54     Include incl;
55     List<Include> incls = new ArrayList<Include>();
56 }
57 {
58     (
59         incl = Include() <EOL>
60         {
61             incls.add(incl);
62         }
63         |
64         <EOL>
65     )*
66     { return incls; }
67 }
```

```
68
69 Include Include() : {
70     List<Action> actions;
71     Token moduleType;
72     Include incl;
73 }
74 {
75     <INCLUDE>
76     moduleType = <IDENT>
77     actions = Actions()
78     <SEMICOLON>
79     {
80         incl = new Include(moduleType.toString(), actions);
81         return incl;
82     }
83 }
84
85 List<Action> Actions() : {
86     List<Action> acts = new ArrayList<Action>();
87     Action act;
88 }
89 {
90     act = Action()
91     { acts.add(act); }
92     (
93         <COMMA> act = Action()
94         { acts.add(act); }
95     )*
96     { return acts; }
97 }
98
99 Action Action() : {
100     Token actionNameTok;
101     String actionName;
102     Action act;
103     List<Option> options;
104 }
105 {
106     <OPEN_PAREN>
107     (
108         actionNameTok = <IDENT> <CLOSE_PAREN>
109         { actionName = actionNameTok.toString(); }
```

```
110     |
111         <CLOSE_PAREN>
112         { actionName = ""; }
113     )
114     <AT>
115     options = Options()
116     {
117         act = new Action(actionName, options);
118         return act;
119     }
120 }
121
122
123 List<Option> Options() : {
124     List<Option> opts = new ArrayList<Option>();
125     Option opt;
126 }
127 {
128     opt = Option()
129     { opts.add(opt); }
130     (
131         opt = Option()
132         { opts.add(opt); }
133     )*
134     { return opts; }
135 }
136
137 Option Option() : {
138     Option opt;
139     OptionInfo optInfo;
140     List<OptionInfo> optInfos = new ArrayList<OptionInfo>();
141 }
142 {
143     <OPEN_BRACKET>
144     optInfo = OptInfo ()
145     { optInfos.add(optInfo); }
146     (
147         <COMMA>
148         optInfo = OptInfo ()
149         { optInfos.add(optInfo); }
150     )*
151     <CLOSE_BRACKET>
```

```
152     {
153         opt = new Option(optInfos);
154         return opt;
155     }
156 }
157
158 OptionInfo OptInfo() : {
159     Token tok1, tok2;
160     JoinPoint joinPoint;
161     OptionInfo optInfo;
162 }
163 {
164     tok1 = <IDENT>
165     { optInfo = new OptionInfoString(tok1.toString()); }
166     (
167         tok2 = <IDENT>
168         {
169             Parameter p = new Parameter(tok1.toString(), tok2.toString());
170             optInfo = p;
171         }
172         (
173            joinPoint = JoinPoint (tok1.toString(), tok2.toString())
174            { optInfo = joinPoint; }
175         )?
176     )?
177     { return optInfo; }
178 }
179
180 JoinPoint JoinPoint(String retType, String funName) : {
181     String paramStr;
182     List<Parameter>params;
183     JoinPoint joinPoint;
184 }
185 {
186     //retType = <IDENT>
187     //funName = <IDENT>
188     <OPEN_PAREN>
189     (
190         <DOT_DOT>
191         { joinPoint = JoinPoint.newDotDot(retType, funName); }
192  |
193         params = Params ()
```

```
194         { joinPoint = new JoinPoint(retType, funName, params); }
195     )
196     <CLOSE_PAREN>
197     {
198         return joinPoint;
199     }
200 }
201
202 List<Parameter> Params() : {
203     List<Parameter>params = new ArrayList<Parameter>();
204     Parameter param;
205 }
206 {
207     (
208         param = Param ()
209         { params.add(param); }
210         (
211             <COMMA> param = Param ()
212             { params.add(param); }
213         )*
214     )?
215     { return params; }
216 }
217
218 Parameter Param() : {
219     Token pType, pName;
220 }
221 {
222     pType = <IDENT>
223     pName = <IDENT>
224     { return new Parameter(pType.toString(), pName.toString()); }
225 }
```

Listing B.1: ALPH Language Definition