University of Crete

Computer Science Department

ClassMATE: Classroom **M**ultiplatform **A**ugmented **T**echnology **E**nvironment

by

ASTERIOS LEONIDIS

MASTER'S THESIS

Heraklion, September 2010

University Of Crete
Computer Science Department

ClassMATE

Classroom **M**ultiplatform **A**ugmented **T**echnology **E**nvironment

by

ASTERIOS LEONIDIS

A thesis submitted in partial fulfillment of the
requirements for the degree of

Master of Science

Author: ____
Leonidis Asterios, Department Of Computer Science

Board of enquiry:

Supervisor ____
Constantine Stephanidis, Professor

Member ____
Anthony Savidis, Associate Professor

Member ____
Dimitris Plexousakis, Professor

Approved by: ____
Panos Trahanias, Professor
Chairman of the Graduate Studies Committee

Heraklion, September 2010

Abstract

The evolution of Information Technology (IT) for more than three decades has drastically affected the way users interact with personal computers and increased their expectations from technology. Towards this objective, researchers developed novel concepts to provide content-rich invisible computing applications, eventually leading to the emergence of the Ambient Intelligence paradigm. Ambient Intelligence is a vision of the future which offers great opportunities to enrich everyday activities (e.g., on the road, at home, at work, etc.). Considering that ICT (Information and Communication Technologies) has been proven to play an important role in education, this thesis investigates the promising potentials of AmI in the education domain.

The notion of Smart Classroom has been around already for a few years. In a Smart Classroom, conventional classroom activities are enhanced with the use of pervasive and mobile computing. However, the majority of the current approaches towards the realization of the Smart Classroom addresses various issues unilaterally either from the technological or the educational perspective, neglecting the main objective of supporting the student during the learning process. The ClassMATE system reported in this thesis aims to provide numerous essential educational-related facilities both for the student and for the teacher. ClassMATE, in collaboration with the PUPIL system, incarnates a functional prototype of the envisioned Ambient Intelligence Classroom (in the context of ICS-FORTH AmI Programme).

ClassMATE constitutes the backbone infrastructure of the Ambient Intelligence Classroom system, aiming to provide "intelligent" facilities to enhance the educational process. These facilities include: (i) a context-aware classroom orchestration process based on information gathered through ambient environment monitoring, (ii) a mechanism to address heterogeneous interoperability of AmI services and devices, (iii) a synchronous and asynchronous communication scheme, (iv) a user profiling and (v) a content classification mechanism in order to deliver personalized content based on the context of use and the actual needs of every individual learner.

The thesis discusses the overall ClassMATE architecture and presents in details the implementation of the above mentioned mechanisms.

Περίληψη

Η εξέλιξη της κοινωνίας της πληροφορίας κατά την διάρκεια των τελευταίων δεκαετιών έχει επηρεάσει δραστικά τον τρόπο με τον οποίο οι χρήστες αλληλεπιδρούν με τους υπολογιστές, καθώς και τις προσδοκίες τους από την τεχνολογία. Στοχεύοντας την ικανοποίηση των χρηστών, δημιουργήθηκαν νέες καινοτόμες ιδέες για ευφυή συστήματα παροχής πλούσιου διαδραστικού περιεχόμενου μέσω του «αόρατου» υπολογιστή. Οι τάσεις αυτές αποτελούν τον πρόδρομο για καινοτόμα περιβάλλοντα διάχυτης νοημοσύνης.

Η διάχυτη νοημοσύνη οραματίζεται την απλοποίηση, διευκόλυνση και επέκταση των ανθρώπινων καθημερινών δραστηριοτήτων, για παράδειγμα στον δρόμο, στο σπίτι και στην εργασία, όπου η πρόσβαση σε πληροφορίες θα είναι διαρκής και απεριόριστη. Γνωρίζοντας ότι η τεχνολογία παίζει ένα σημαντικό ρόλο στον τομέα της εκπαίδευσης, αυτή η εργασία εξετάζει τις δυνατότητες και τους νέους ορίζοντες που ανοίγονται από ένα εκπαιδευτικό περιβάλλον διάχυτης νοημοσύνης.

Η έννοια της "έξυπνης" τάξης εμφανίστηκε τα τελευταία χρόνια και αναφέρεται σε ένα τεχνολογικά επαυξημένο εκπαιδευτικό περιβάλλον. Στην "έξυπνη" τάξη, οι συμβατικές εκπαιδευτικές δραστηριότητες υποστηρίζονται από διάχυτες και φορητές υπολογιστικές συσκευές. Ωστόσο, στην πλειοψηφία τους οι υπάρχουσες προσεγγίσεις προσπαθούν να αντιμετωπίσουν μονομερώς διάφορα εκπαιδευτικά ζητήματα, εξετάζοντάς τα είτε από τεχνολογικής είτε από εκπαιδευτικής σκοπιάς, και δεν προσφέρουν ολοκληρωμένες λύσεις για την υποστήριξη του μαθητή καθ'όλη την διάρκεια της μάθησης. Το σύστημα ClassMATE, που αναπτύχθηκε στο πλαίσιο αυτής της εργασίας, στοχεύει να υποστηρίξει τις δραστηριότητες μαθητών και καθηγητών, προσφέροντας καθοδήγηση και υποστήριξη σε όλα τα στάδια της εκπαιδευτικής διαδικασίας. Το ClassMATE, σε συνεργασία με το σύστημα PUPIL, ενσαρκώνει ένα πρωτότυπο της οραματιζόμενης Τάξης Διάχυτης Νοημοσύνης (στα πλαίσια του προγράμματος AmI Programme του ICS-FORTH).

Το ClassMATE αποτελεί την ραχοκοκκαλιά της Τάξης Διάχυτης Νοημοσύνης και την εμπλουτίζει με "έξυπνους" μηχανισμούς, με στόχο την υποστήριξη και διευκόλυνση των εκπαιδευτικών δραστηριοτήτων. Συγκεκριμένα, παρέχει τα εξής: (α) έναν "ψηφιακό" συντονιστή της τάξης, οι αποφάσεις του οποίου προσαρμόζονται στις εκάστοτε συνθήκες του περιβάλλοντος χρήσης, (β) έναν επεκτάσιμο μηχανισμό που επιτρέπει την υποστήριξη ποικίλων υπηρεσιών και συσκευών που δύναται να υπάρξουν σε περιβάλλοντα διάχυτης νοημοσύνης, (γ) ένα δίκτυο "σύγχρονης" και "ασύγχρονης" επικοινωνίας, (δ) ένα σύστημα διαχείρισης δυναμικά δημιουργούμενων προφίλ μαθητών, και τέλος (ε) έναν αυτόματο

μηχανισμό κατηγοριοποίησης του εκπαιδευτικού περιεχομένου που επιτρέπει την προσωποποιημένη παροχή και παρουσίαση του ώστε να καλύψει τις τρέχουσες ανάγκες του εκάστοτε μαθητή και πλαισίου χρήσης.

Αυτή η εργασία παρουσιάζει την αρχιτεκτονική του ClassMATE, και εμβαθύνει στις λεπτομέρειες υλοποίησης των μηχανισμών που αναφέρθηκαν παραπάνω.

Ευχαριστίες (Acknowledgements)

Πρωτίστως θα ήθελα να ευχαριστήσω τον επόπτη της μεταπτυχιακής μου εργασίας, Καθ. Κωνσταντίνο Στεφανίδη, για την συνεχή καθοδήγηση και υποστήριξη που μου προσέφερε τα τελευταία σχεδόν τέσσερα χρόνια στο πλαίσιο της συνεργασίας μου με το Εργαστήριο Αλληλεπίδρασης Ανθρώπου-Υπολογιστή του Ινστιτούτου Πληροφορικής του Ιδρύματος Τεχνολογίας και Έρευνας. Μέσω αυτής της συνεργασίας θεμελίωσα τις γνώσεις μου στο συγκεκριμένο τομέα και εξελίχθηκα στην επιστήμη των υπολογιστών, θέτοντας στόχο σε ερευνητικά αποτελέσματα υψηλής ποιότητας.

Εν συνεχεία θα ήθελα να ευχαριστήσω θερμά τη Μαργαρίτα Αντόνα, τη Σταυρούλα Ντοά και τον Γιώργο Μαργέτη για την συμβολή τους στην εκπόνηση της εργασίας μου καθώς εκτός από τις πολύτιμες συμβουλές και την βοήθεια τους, δημιούργησαν ένα εξαιρετικό περιβάλλον συνεργασίας, στο οποίο απέδωσα το μέγιστο των δυνατοτήτων μου για να πραγματοποιηθεί αυτή η εργασία. Για όλα αυτά, αλλά και για την προσωπική συμπαράσταση που μου προσέφεραν, τους είμαι ευγνώμων.

Επιπλέον, θα ήθελα να ευχαριστήσω τους φίλους μου Έφη, Γιάννη Λ. , Χρύσα και Γιάννη Δ. που μου συμπαραστάθηκαν κατά την διάρκεια της εκπόνησης της μεταπτυχιακής μου εργασίας προσφέροντας μου τις απαραίτητες στιγμές ηρεμίας όταν τις είχα ανάγκη.

Ιδιαίτερα, θα ήθελα να ευχαριστήσω την Μαρία, καθώς η παρουσία της στην ζωή μου, μου έδωσε την απαραίτητη ώθηση για να κυνηγήσω τις προσδοκίες μου και η αμέριστη συμπαράσταση της σε όλες τις φάσεις της παρούσης εργασίας ήταν καθοριστική για την ολοκλήρωση της.

Τέλος, το μεγαλύτερο ευχαριστώ το οφείλω στους γονείς μου Δημήτρη και Αναστασία καθώς και στον αδερφό μου Βασίλη, οι οποίοι παρά την απόσταση που μας χώριζε, με στήριζαν και με εμψύχωναν καθημερινά. Η αγάπη τους και η κατανόηση τους με βοήθησαν κατά τη διάρκεια των σπουδών μου και ελπίζω αυτή η εργασία να αποτελεί μια μικρή «ανταμοιβή» για τις θυσίες που έκαναν για μένα.

To my parents

Demetres and Anastasia

Table of Contents

ABSTRACT .... **IV**

ΠΕΡΙΛΗΨΗ .... **V**

ΕΥΧΑΡΙΣΤΙΕΣ (ACKNOWLEDGEMENTS) .... **VII**

LIST OF FIGURES .... **XI**

LIST OF TABLES .... **XII**

1 INTRODUCTION **1**

1.1 AMBIENT INTELLIGENCE 1
1.2 THE SMART CLASSROOM 1
1.3 OBJECTIVES OF THE CLASSMATE SYSTEM 2
1.4 OVERVIEW OF CLASSMATE 4
1.5 STRUCTURE OF THIS THESIS 5

2 RELATED WORK **6**

2.1 AMBIENT ENVIRONMENTS 6
2.2 TOWARDS A TECHNOLOGICALLY-AUGMENTED CLASSROOM 8
2.3 ADAPTIVE HYPERMEDIA 11

3 THE CLASSMATE APPROACH **14**

3.1 THE CLASSMATE ARCHITECTURE 15

4 AMBIENT ENVIRONMENT MANAGEMENT **18**

4.1 THE CONTEXT MANAGER 18
4.1.1 Platform Expert *19*
4.1.1.1 Service Factory Interface 22
4.1.1.2 Service Info & Factory Entries 23
4.1.1.3 Service Factory Registry 25
4.1.1.4 Configuration files (Platform.xml, Global Service Definitions.xml) 26
4.1.2 ClassMATE Events *28*
4.1.2.1 Base Event Args 30
4.1.2.2 Abstract Event producer 31
4.1.2.3 Event Proxy 32
4.1.2.4 Event Registry 34
4.1.2.5 ClassMATE Message Events 36
4.1.2.6 ClassMATE Commands 36
4.1.3 Artifact Director *46*
4.1.3.1 Mime Command Handler 48
4.1.4 Class Orchestrator *49*
4.1.4.1 Security Manager 49
4.1.5 Application Launcher *50*
4.1.5.1 Application Registry 51
4.1.5.2 Mimetype - Application Map 52
4.1.6 State Serialization *53*
4.1.6.1 Resource Format Pair 54
4.1.6.2 State Class 55
4.1.6.3 State Manager 56
4.1.7 Initialization Process *57*
4.1.8 Migration Process *58*

4.2 DEVICE MANAGER 59
4.2.1 Towards a universal Multitouch solution 59
4.2.2 Book Localizer 62

5 CONTENT PERSONALIZATION 63

5.1 USER PROFILE 63
5.2 DATASPACE 67
5.2.1 Related technologies overview 67
5.2.1.1 Learning Object Metadata (LOM) 67
5.2.1.2 Sparql 68
5.2.1.3 SemWeb 68
5.2.2 Metadata 69
5.2.2.1 LOM Types 70
5.2.2.2 LOM Metadata Structure & LOM Entry 71
5.2.3 Resource Reference Format 72
5.2.3.1 HotSpot 73
5.2.3.2 Multimedia 75
5.2.3.3 Hint 76
5.2.4 Content Classification and Personalized Delivery 76
5.2.4.1 Taxonomies Overview 77
5.2.4.2 Taxonomies Installation 78
5.2.4.3 Content Collection Mechanism 79
5.2.4.4 Content Classification 82
5.2.5 Data Repository 83
5.2.5.1 File Manager 83
5.2.5.2 Content Population 84

6 CONCLUSIONS AND FUTURE WORK 85

6.1 SUMMARY 85
6.2 CONCLUSION 86
6.3 FUTURE WORK 87

7 BIBLIOGRAPHY 88

APPENDIX A 91

List of figures

FIGURE 1: THE ENVISIONED SMART CLASSROOM 3

FIGURE 2: THE ENVISIONED ARCHITECTURE OF THE SMART CLASSROOM SYSTEM 17

FIGURE 3: CONTEXT MANAGER ARCHITECTURE DIAGRAM 18

FIGURE 4: INTERFACE-BASED ABSTRACT SERVICE FACTORY APPROACH 21

FIGURE 5: PLATFORM EXPERT'S METHOD TO CONCRETE SERVICE OBJECTS 22

FIGURE 6: ABSTRACT FACTORY DESIGN PATTERN 23

FIGURE 7: PROCESS TO INSTANTIATE A CONCRETE SERVICE OBJECT 24

FIGURE 8: SAMPLE USAGE OF THE SERVICE RESOLUTION MECHANISM 25

FIGURE 9: GENERIC SEND EVENT METHOD 31

FIGURE 10: THE PROXY DESIGN PATTERN 32

FIGURE 11: THE EVENT PROXY RATIONALE 34

FIGURE 12: EVENT DISTRIBUTION MECHANISM 36

FIGURE 13: CLASSMATE COMMAND TYPE HIERARCHY 38

FIGURE 14: SENDER AND RECEIVE MAP OF COMMAND EVENTS 38

FIGURE 15: CLASSMATE COMMAND JOURNEY 46

FIGURE 16: MIME COMMAND HANDLING PROCESS 51

FIGURE 17: APPLICATION'S STATE SERIALIZATION PROCESS 54

FIGURE 18: PLATFORM INITIALIZATION PROCESS 57

FIGURE 19: APPLICATION MIGRATION PROCESS 58

FIGURE 20: WINDOWS 7 SENDS MESSAGES FROM MULTITOUCH HARDWARE TO AN APPLICATION 60

FIGURE 21: WINDOWS 7 MANIPULATION OVERIEW 61

FIGURE 22: PHYSICAL COURSE BOOK LOCALIZATION PROCESS 62

FIGURE 23: AUTOMATIC CONTENT DISCOVERY PROCESS 66

FIGURE 24: THE LOM DATATYPE 71

FIGURE 25: LEARNING OBJECT METADATA (LOM) SPECIFICATION 72

FIGURE 26: SEMWEB BRIDGE BETWEEN RDF DATA AND A RELATIONAL DATABASE 78

FIGURE 27: A SAMPLE QUERY FAMILY THAT DISCOVERS RELATED IMAGES 81

List of tables

TABLE 1: THE SMART CLASSROOM INTENDED FOR THE ICS-FORTH AMBIENT PROGRAMME 14

1 Introduction

1.1 Ambient Intelligence

Information technology has been evolving for more than three decades from the introduction of the first personal computer in the late 70's until the dominant World Wide Web paradigm in the early 00's. This continuous evolution affected the way users interact with personal computers and increased their expectations for innovative breakthrough technologies, causing the intrinsic potentials of the IT to steadily unveil. People nowadays are hooked on connectivity – they want access to information anytime from anywhere and the latest trends in IT industry indicates a path towards embedding communication facilities into devices of everyday use starting from the mobile phones to TV sets, while even more will definitely emerge.

Towards this objective, researchers developed novel concepts, techniques and tools to provide content-rich invisible computing. This led to the emergence of a novel domain in ICT (Information and Communication Technologies), opening up new horizons, namely Ambient Intelligence. Ambient Intelligence is a vision of the future information society stemming from the convergence of ubiquitous computing, ubiquitous communication and intelligent user-friendly interfaces.

Ambient Intelligence offers great opportunities to support social development, enrich everyday activities and dramatically change the way of life; for instance, on the road critical information can be delivered easily and in real-time, whereas at home an ambient environment delivers seamless, on-demand content in any room, while also facilitating interconnection between homes. Considering that ICT has been proven to play an important role in education by increasing students' access to information, enriching the learning environment, allowing students' active learning and collaboration and enhancing their motivation to learn [1], education could not oversee the promising potentials of Ambient Intelligence.

1.2 The Smart Classroom

The notion of smart classrooms became prevalent in the past decade [42]. Smart classroom is used as an umbrella term, implicating that classroom activities are enhanced with the use of pervasive and mobile computing, sensor networks, artificial intelligence, robotics, multimedia computing, middleware and agent-based software [11] to modernize students' experience and fully exploit the existing infrastructures (e.g., online resources of educational content). Following the rationale of augmented technology in the educational environments,

new means of interaction - such as interactive whiteboards, touch screens and tablet PCs - have gained popularity and have become a major tool in the educational process, allowing more natural interaction and restoring the past luster of the school when students were excited about their education. For the Smart Classroom to be acceptable to its users, it should be defined with educational concerns in mind, where the technology should enhance the quality of education without increasing its complexity or introducing technology-oriented burdens. In general, it should be reliable and controllable, but nevertheless adaptive to students' habits and changing contexts. Smart classrooms, via their technological enhancement, may support one or more of the following capabilities: video and audio capturing in classroom [33], automatic environment adaptation according to the context of use, such as lowering the lights for a presentation [12], lecture capturing enhanced with the instructor's annotations, information sharing between class members or even a tele-education experience similar to a real classroom experience [24]. The main objectives of the Smart Classroom are primarily to support the students during the learning process and facilitate its administration by the teachers. Therefore, the anticipated capabilities extend beyond simple automation of repetitive tasks and inclusion of e-learning services available in numerous platforms.

1.3 Objectives of the ClassMATE system

The ClassMATE system reported in this thesis aims to provide the following facilities in the context of the Ambient Intelligence classroom:

- For the Student
    - Direct access to unlimited educational content
    - Dedicated personal area to store educational material (e.g. submitted assignments, lecture notes, etc.)
    - Educational applications accessible not only during school hours, but at any time through the supported personal devices (PDAs / Smartphones, and Netbooks, etc.)
    - Personalized content and study guidelines delivery, semantically discovered according to each individual student's needs
    - Progress monitoring and detailed record keeping
    - Collaboration among classmates
    - Active participation in the teaching process
    - Automation of repetitive everyday tasks
    - Flexible workspace environments

- For the Teacher
    - Lecture preparation assistant
    - Statistics of class progress
    - Real-time student monitoring
    - Automation of everyday tasks

An example of Ambient Intelligence Classroom is currently under development in the context of the ICS-FORTH AmI Programme. The ClassMATE system, presented in this thesis, in collaboration with the PUPIL system [22] incarnates the initial concept into a functional prototype (Fig. 1), consisting of five different platform prototypes (artifacts): the AmIDesk, the SmartDesk, the AmIBoard, the SmartBoard and a common portable computer.

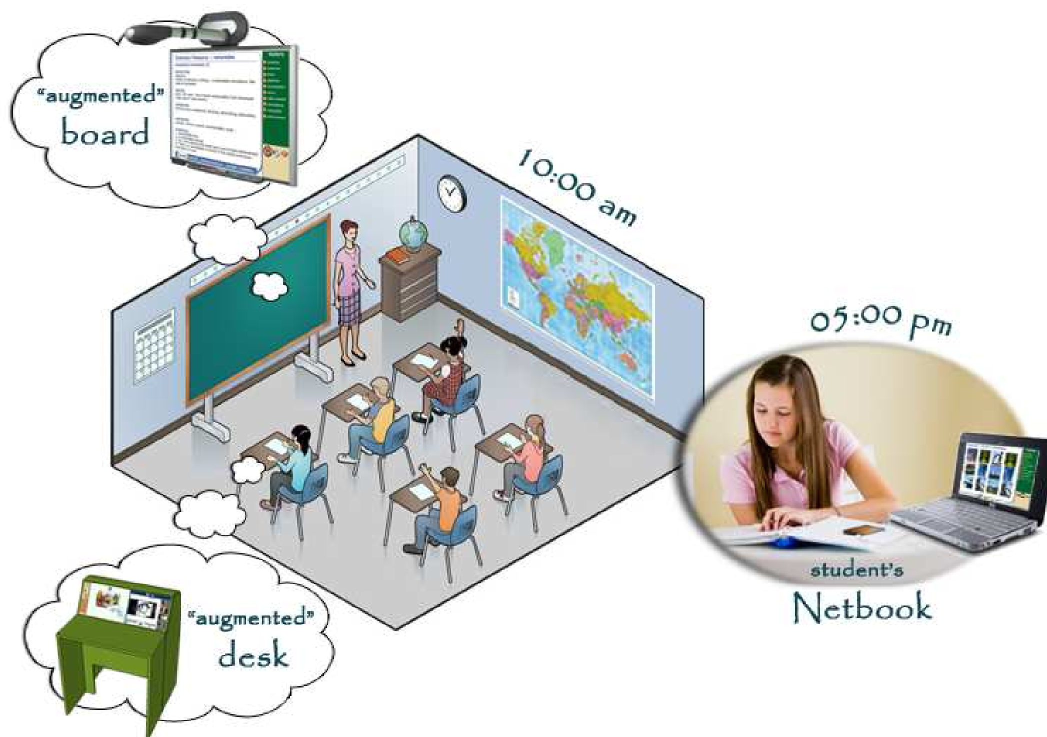


Figure 1: The Envisioned Smart Classroom

The PUPIL system acts as the front-end of the overall platform. In brief PUPIL: (i) promotes the design of usable educational applications though a library of "intelligent" widgets, (ii) equips classroom artifacts with flexible workspaces and enables application migration among them, (iii) support reusability of common interface patterns and minimize artifact specific design decisions and (iv) free designers from building the same interface for various platforms as the single version automatically transforms to the current context to ensure optimal display.

The ClassMATE system reported in this thesis constitutes the backbone infrastructure that aims to provide a set of "intelligent" facilities to enhance the educational process. The key feature that differentiates ClassMATE from similar architectures is the education-centric approach that has been adopted during its design. In more details, ClassMATE monitors the ambient environment and makes context-aware decisions to assist (i) the student in conducting learning activities, by simplifying everyday tasks and providing personalized content according to individual needs, and (ii) the teacher with administrative issues by automating common activities. Summarizing, ClassMATE aims to provide a robust and open ubiquitous computing framework suitable for a school environment that:

- provides a context aware classroom orchestration based on information coming from the ambient environment
- addresses heterogeneous interoperability of AmI services and devices
- facilitates synchronous and asynchronous communication
- supports user profiling and behavioral patterns discovery
- encapsulates content classification and support content discovery and filtering

1.4 Overview of ClassMATE

To achieve the above objectives, ClassMATE introduces various modules enclosed either in the (i) Ambient Environment Manager or in the (ii) Content Personalization Manager. The Ambient Environment Manager encloses the two modules that monitor the environment and enable contextual awareness the Context and the Device Manager. The Context Manager includes: (i) the Platform Expert that operate as an abstraction layer of that multiplatform environment that provides access to the wide-variety of platform-specific functions in a platform-independent manner, (ii) the ClassMATE Event System that defines a hierarchy with specialized event types forming the ClassMATE event type system and implements the essential mechanisms for their distribution, (iii) the Artifact Director that is a context aware module that orchestrates each artifact, (iv) the Class Orchestrator that controls every aspect of the classroom in a high-level, (v) the Application Launcher that bridges the ClassMATE with the PUPIL system by instructing the applications opening, (vi) the State Serialization Manager that manages application's state serialization and deserialization, and finally (vii) the Migration Processor that facilitates the application migration from the current local artifact to a remote node. The Device Manager includes: (i) the Multitouch Device Manager that enables multitouch interaction schemes and (ii) the Book Localizer that determines current context of use (e.g. currently studied course).

The Content Personalization Manager encloses the two modules charged with the delivery of personalized education content based on the current needs of the individual learner, the User Profile and the Data Space Manager. The User Profile collects personal data associated with a specific user (both static and dynamic) and the Data Space Manager which provides an abstraction layer between the applications and the physical storage layer and encapsulates a filtering mechanism for personalized content delivery based on user needs and preferences (available through the User Profile).

1.5 Structure of this thesis

The rest of the thesis is structured as follows:

- Chapter Two (2) presents a brief overview of related work
- Chapter Three (3) presents the ClassMATE approach towards delivering a ubiquitous computing framework for the AmI Classroom
- Chapter Four (4) describes the architecture and the technical details of the Ambient Environment Manager
- Chapter Five (5) presents the Educational Content Manager and in particular the content classification and the personalization mechanisms
- Chapter Six (6) describes in brief a real-life scenario supported by the ClassMATE system
- Chapter Seven (7) summarizes the conclusions of this thesis and outlines the future steps.

2 Related Work

2.1 Ambient Environments

As highlighted in [4] it is part of the ten-year vision of the Ministries of Education in Europe, in response to a changing information society, to promote ambient schooling, and create a schooling environment 'surrounding' the pupil in a non-intrusive way to address the issue of 'disconnection' between ICT use in and out of schools, with pupils being increasingly critical of the former. In short, the author points out that as fundamental issues related to the re-organization of learning is not addressed properly, the future of e-learning for schools in the Information Society is under question. Assche argues that schools have equipped some of their classrooms with computers, however their use remains limited, while through ambient schooling, via the appropriate use of advanced technologies, pupils will be supported as they continue to learn not only in formal institutions, but in the home, libraries, museums and the wider community as well. The envisioned system, the schoolGRID, can be seen as a further evolution of learning object brokerage systems including functionality related to: (i) learning content management and exchange, (ii) learning communities and (iii) school management. Finally, the envisioned system incorporates on-the-fly tailoring facilities with regard to the personal learning style, preferences, competencies, dynamic learner profile, or learning needs of the pupils.

Lin, Kratcoski and Swan, in [24], conducted a case study on a third grade class to explore the use of ubiquitous tools during a science unit on forces and motion. Their study documented the ways in which children could construct knowledge and create representations of their learning when afforded ready access to a variety of digital devices, and explored the implications of ubiquitous computing environments on student collaboration and situated learning. Situated learning asserts learning is connected to real situations in which knowledge is created and used. For that be achieved, a ubiquitous environment was selected to present knowledge in an authentic context and support social interaction and collaboration. The outcome of that study suggested that the use of ubiquitous computing tools within a situated learning approach facilitated the students' attainment of curricular content, technology skills, and collaboration skills.

Both [4] and [24] highlighted the need for a smart classroom environment. According to ISTAG [20], a 'key enabling technology' for a successful implementation of the AmI landscape is the presence of middleware systems that act as the main coordinator of the heterogeneous and distributed services of an ambient intelligent environment. Nowadays, a

plethora of such systems exist, providing the necessary functionality for the structured communication between the various AmI environments' components. As mentioned in [15], the essential requirements that an AmI middleware should address are: heterogeneity integration, synchronous and asynchronous communication, resilience, security and ease of use. Moreover, a successful AmI middleware should provide all the necessary monitoring and control facilities of the diverse ubiquitous computing artifacts which interoperate in an AmI environment, thus supporting a context aware orchestration strategy.

The AMIGO project ([21], [2]) developed a middleware that dynamically integrates heterogeneous systems to achieve interoperability between services and devices in intelligent home networks. The AMIGO architecture consists of a base middleware component, a programming and deployment framework and a series of legacy services named 'Intelligent User Services'. The base middleware component ensures the secure and robust interoperability of the heterogeneous service platforms that an AmI environment may host, providing also a generic mechanism for their semantic description of functional, non-functional and architectural features. The programming and deployment framework enables the developers to implement ad-hoc AMIGO-aware distributed services, allowing them to choose among two programming languages alternatives, .NET C# and Java. The Intelligent User Services on the one hand constitute the legacy services layer of the AMIGO architecture that provides users with basic functionality to interact with the intelligent environment. On the other hand, such services are responsible for compositing multiple information sources and disseminating context-related information. Finally, any available information is encoded into a user profile and exploited towards environment's adaptation based on the user's state and context changes.

Bandelloni and Patemo [5] suggested a system that provides users immersed in a multiplatform environment with the possibility of interacting with an application while freely moving from one device to another. Their work supported platform-aware runtime migration for Web applications that allowed users to change device and continue their interaction from the same point. A migration server takes into account the runtime state of an application and adapts its interface to best fit the target platform. For migration to be achieved, the user interface is encoded in an abstract interface description to facilitate transformation. In addition to total migration where the client interface migrates totally from a device to the other, they propose that partial migration and synergistic access can be achieved through their platform. In partial migration the client interface is divided into the

control and the presentation segments whilst the control segment remains on the current device and the presentation one migrates to the other device; in the synergistic access, named mixed migration, the client interface is split into several parts, concerning both control and presentation and different parts are distributed over two or more devices.

The Voyager development framework [32] supports the implementation of ambient dialogues, i.e., dynamically distributed user Interfaces, which exploit, on-the-fly, the wireless devices available at a given point in time. The primary motivation of Voyager is based on the vision that the future computing platforms will not constitute monolithic ‘‘all-power-in-one’’ devices, but will likely support open interconnectivity, enabling users to combine the facilities offered by distinct devices on-the-fly. Physically distributed devices may be either wearable or available within the ambient infrastructure (either stationary or mobile), and may be connected via a wireless communication link for easier deployment. Operationally, each such device will play a specific role by exposing different processing capabilities or functions, such as character display, pointing, graphics, audio playback, speech synthesis, storage, network access, etc. From the hardware point of view, such devices may be wrist watches, earphones, public displays, home appliances, office equipment, car electronics, sunglasses, ATMs, etc. The Voyager implementation focuses on device discovery and registry architecture, device-embedded software implementation, ambient dialogue style and corresponding software toolkit development, and a method for dynamic interface adaptation, ensuring dialogue state persistence.

2.2 Towards a technologically-augmented Classroom

In [7] Bravo, Hervas and Chariva propose a context-aware identification mechanism to support implicit user interaction. They argue that, as in recent years many research efforts aimed at obtaining simple and natural interaction with of computers, the same vision emerge for the Ubiquitous Computing, where the computer is distributed in a series of devices with reduced functionality, spread over the user’s environment and communicating wirelessly. The RFID technology is used to implicitly provide input to the system and to offer natural interaction, as a smart label carried by the student / teacher is the only requirement for the identification and contextual services acquisition. This contextual identification allows users to obtain services from the environment with ease (e.g., visualization of course’s presentations, proposed and solved assignments, etc.). The key feature of their work resides in the contextual awareness categorized in identity (which are his preferences),

location (is s/he standing near the board), time (which are the scheduled tasks) and task (what is s/he doing right now).

Breuer et al. [8] aimed to synthesize two lines of development that have been dealt with independently so far: (i) the development and evaluation of educational technologies to support problem-oriented and collaborative learning activities inside and outside of the classroom, and (ii) interaction design patterns as a means to document and generate design knowledge. They propose both a software framework to enhance classroom interaction through interactive whiteboards, multiple clients with pen-tablets and PDAs, and a basic layout of a pattern language for formal and informal learning environments. The proposed software orchestrates various hardware artifacts (e.g., Interactive Whiteboard application, access from multiple pen-tablets) in order to support collaboration, while the overall approach was illustrated using an interaction design pattern language for learning environments, aiming to document and optimize existing solutions and patterns, but also to employ them in order to generate new design knowledge.

The work conducted by Paredes et al. in [31] aimed to bring the Information and Communication Technologies into the traditional classroom. They argue that collaboration and Ubiquitous Computing paradigms would benefit to that direction if only they were put together into the educational environment; to demonstrate their objectives, they developed as a study case a system for language learning, in particular English as a Foreign Language (EFL). The AULA system was intended to improve communication abilities in a learning environment in order to achieve the necessary skills to develop a project in group by writing reports in a collaborative task. The learning task to be developed was writing a text (an essay, a report, a news article, etc.) in a collaborative way, since this is a usual activity in many contexts in real life. The system facilitates the structuring of the information resulting from the brainstorming process into the so-called aspects. Aspects and ideas are blocks of partial information, which constitute the initial document's framework on top of which the students contributed their work. Any contributions made are stored in a persistent way, thus supporting the student before, during and after the class.

Soh, Khandaker and Jiang proposed I-MINDS (Intelligent Multiagent Infrastructure for Distributed Systems in Education) [35], a system that provides a computer-supported collaborative learning (CSCL) infrastructure and environment for learners in synchronous learning and classroom management applications for instructors, for large classroom or distance education situations. Three agent types that provide educational-related facilities

existed: the teacher, the student and the group agent. The teacher agent facilitates instructor's interaction with students, classroom's management and performance monitoring. The student and the group monitoring agents supports a number collaborative learning mechanisms and both individual and group monitoring to evaluate performance. I-MINDS had been deployed and evaluated in a real-time environment, and the results have shown that such a system could be used to support student cooperative learning activities, and also as a testbed to collect instructional or pedagogical data for better understanding of student collaborative learning.

In Yau et al. [43] a Smart Classroom is proposed to increase the level and the quality of collaboration between college students and the instructor, since group formation to solve problems or develop projects are typical tasks in such environments. Every student is equipped with a situation-aware PDA, while the PDAs dynamically form mobile ad hoc networks for group meetings. Each PDA monitors its situation (locations of PDAs, noise, light, and mobility) and uses situations to trigger communication activity among the students and the instructor for group discussion and automatic distribution of presentation materials. In addition to educational material exchange, the PDAs support various educational-related tasks, such as electronic submission of exercises or questions, exams preparation, distribution and collection, etc. Finally, the proposed middleware is claimed to effectively address situation-awareness and ad hoc group communication for pervasive computing, by providing development and runtime support to the application software.

Baton [23] is the heart of the Smart Classroom [34] developed by Lin et al. It is a service management system to explicitly resolve the particular issues stemming from smart spaces while coordinating agents (delegating smart things in smart spaces). Baton is designed as a complement to coordination approaches in multi-agent systems with a focus on mechanisms for service discovery, service composition, request arbitration and dependency maintenance. Services are described in the OWL-S language, which makes the processes of service discovery and composition more accurate and efficient, while request collisions are modeled as linear programming problems that facilitate easier resolution and minimize service dependencies. Finally, the process of fulfilling a request is handled as a transaction, and a two-phase commit algorithm is utilized to assure its atomicity. The Smart Classroom [34] that is built on top of Baton mainly focuses on tele-education and distance learning, where every participant (remote or local) has the same view, thus the teacher can instruct the remote students just like teaching face-to-face in a conventional classroom.

2.3 Adaptive Hypermedia

In addition to “intelligent” objects and context-aware systems, the domain of education highly benefits from content filtering mechanisms that aim to deliver content tailored to the needs of the current learner. As Brusilovsky and Millan state [9], various adaptive web systems incorporate user models to adapt the systems’ behavior to individual users; the user model represents the information about a user and is essential to support the adaptation functionality of the systems. According to the authors adaptive web systems had investigated a range of approaches to user modeling, and the majority of them use feature-based approach to represent and model information about the users, while the once stereotype-based approach has lost dominance. The most popular features modeled and used by adaptive web systems are user knowledge, interests, goals, background, individual traits, and context of work, while each individual adaptive system typically uses a subset of this list. They conclude, among others, that convergence has begun to blur the boundaries between different classes of adaptive web systems, and that an effective adaptation learning algorithm would be able to process each user’s interactive behavior information and simultaneously update the structure of the model.

Heilman et al. proposed an intelligent tutoring system called REAP [16] that provided reader-specific lexical practice for improved reading comprehension. Towards such goal, REAP offered individualized practice to students by presenting authentic and appropriate reading materials selected automatically from the web. To address the various challenges emerged from using authentic material (e.g., technical documentation, sensitive topics, use of slang) recommendations by ELI (English Learning Institute) were adopted and appropriate filtering mechanisms were developed. The REAP system is claimed to satisfy a number of criteria in order to gain acceptance into the classroom at the English Language Institute of the University of Pittsburgh.

In [10], Conlan et al. proposed a multi-model approach to the dynamic composition and delivery of personalized learning utilizing reusable learning objects. The aim was to enhance the educational impact of eLearning courses, while still optimizing the return on investment, by facilitating the personalization and repurposing of learning objects across multiple related courses. Considering that courses typically differ in various aspects (ethos, learning goals,) and learners have different motivations, prior knowledge and learning styles, the adopted approach foresee a clear separation of content, learner and narrative models, and offers an adaptive metadata driven engine that composes, at runtime, tailored educational experiences across a single content base to the learner’s requirements.

Vassileva and Bontchev in [37] argue that modern adaptive hypermedia systems try to select content that best fits to the model of a given learner, based on various forms of system adaptation through mechanisms that rely on setting weights for content pages. They proposed a self-adaptive navigation mechanism based on concepts used for the definition of a polymorphic learner model (e.g., learning style, goals, prior knowledge, etc). These concepts are used for indexing the working paths towards content pages but not the pages themselves, while tests that measure users' satisfaction at certain control points dynamically update the weight of navigation path, thus supporting the adaptation engine's decisions regarding the most suitable path for a given learner model.

Nowadays, we witness a steadily increasing research interest in smart environments; everyday things become "intelligent" modules and the surrounding environment incorporates context-aware software to facilitate interaction with the users. The domain of education observes closely these technological trends with two mainstream approaches aiming to develop the smart classroom. The first mainly focus on incarnating an intelligent environment by integrating technologically advanced objects in the conventional classroom (e.g. interactive whiteboards, portable computers and mobile devices) [7], [8], incorporating software solutions that automatically distribute electronically the available educational material (e.g., presentation, exercises, grades, etc.) [[43][35][23]] and facilitating collaboration and flexible use from various terminals [ref migration]. The latter approach aims to support distance learning either online (i.e., tele-education) or offline (i.e., e-learning). As presented in [43], the main objective of tele-education approaches is to facilitate remote interaction and leaning as if the learner was physically present in the classroom. Concerning the offline learning, the e-learning platforms studied diverge from the current practices where content is structured in uncorrelated courses by introducing content correlation and real time progress monitoring that facilitate personalization to individual learners' needs, based on the actual learner's needs instead of high-level theoretic models.

The role of intelligent ubiquitous technology in the classroom and in education in general is still far from being maturely understood, and systematic approaches to supporting students and the teachers throughout the educational process are necessary. ClassMATE aims to combine the best of both worlds, smart environments and e-learning platforms, by introducing a pervasive ecosystem that assists in a non-obstructive way the students during learning activities both at school and at home. Moreover, through automating common

teachers' activities (e.g., material distribution, homework collection, progress monitoring), ClassMATE permits the teacher to undistractedly focus on the teaching process. ClassMATE, similarly to [2], [43], [24], is built on top of a middleware infrastructure to facilitate dynamic service discovery, supports collaborative tasks [8], [31], [35], stateful application migration [5], service personalization [7] and personalization of semantically discovered content to each individual learner's needs. Its main advantage though is that every decision made and action taken is driven by contextual information (who, when, where, what) towards facilitating end user interaction (student or teacher).

3 The ClassMATE Approach

ClassMATE aims to bridge the gap between the ambient environment and the educational applications in a transparent manner. It constitutes the backbone of the ambient classroom that enables applications to exploit the information provided by the environment, and enhances the educational process by automating time-consuming everyday tasks such as submission deadline notification, student's activity and progress monitoring, and progress report. For that to be achieved, ClassMATE is based on a modular architecture. Additionally, it supports various artifacts in the envisioned classroom. For instance, the Smart classroom intended for the ICS-FORTH Ambient Programme consists of five different platform prototypes (artifacts): the AmIDesk [3], the SmartDesk, the AmIBoard, the SmartBoard and a common portable computer.

Artifact	Characteristics	Resolution	Screen Diagonal	Intended Use
AmIDesk	Vision-based touch-enabled device	1600x600	27''	student's desk
SmartDesk	Touch screen device	1440x900	19''	student's desk
AmIBoard	Vision-based touch-enabled device	1920x1200	81''	classroom board
SmartBoard	Touch sensitive interactive whiteboard	1024x768	77''	classroom board
Netbook	Common Netbook	1024x600	10.1''	both classroom and home

Table 1: The Smart Classroom intended for the ICS-FORTH Ambient Programme

The ClassMATE architecture addresses the needs for a robust and open ubiquitous computing framework in the school environment, as well as fundamental issues such as heterogeneous interoperability of AmI services, synchronous and asynchronous communication, resilience, security, context aware orchestration and ease of use in the intelligent classroom of the future. The key feature of the ClassMATE's architecture, that differentiates it from similar architectures, is the education-centric approach that has been adopted during its design. In more details, ClassMATE aims to provide a set of "intelligent" facilities to enhance the educational process. Therefore, the offered services were defined taking into consideration the needs of students and teachers both during school-hours and when studying at home. In particular, ClassMATE facilitates student's learning activities by simplifying everyday tasks such as submission deadline notification, update of educational material, and by providing personalized content that fits the specific needs of every individual learner. On the other hand, ClassMATE assists the teaching process by

automating common teachers' activities (e.g., material distribution, homework collection, progress monitoring), thus permitting the teacher to better focus on the educational process

3.1 The ClassMATE Architecture

The ClassMATE's core consists of four major components layered in parallel: the Device Manager, the Data Space, the User Profile and the Security Manager, glued together via the fifth major component, the Context Manager. A number of Utility modules, placed in parallel, offer auxiliary services and functionality. These five major components also define the "hooks" where additional functional modules could be integrated into the system to extend the available functionality.

The **Context Manager** is the orchestration component of the ClassMATE's architecture. It monitors the ambient environment and makes context-aware decisions.. In more details, the Context Manager is responsible for making the decisions for every process workflow in the classroom's environment, and controlling the operation and collaboration of ClassMATE's services and applications to address users' needs at any specific time frame. To this purpose, the Context Manager applies appropriate reasoning strategies to user-, service-, application-related data, in the classroom environment. Besides this general orchestration provided by the Context Manager, every AmI artifact in the classroom operates under the orchestration of a local Artifact Director, which is responsible for its robust operation. The Artifact Director, at any time, keeps track of what is currently running (applications or services) on the artifact, and according to the Context Manager's directions they initiate, stop or suspend the processes running on the artifact.

The **Device Manager** offers a generic mechanism for heterogeneous devices manipulation by any ClassMATE enabled application. Every AmI artifact (e.g., interactive board, smart desk, etc.) that belongs to the classroom environment accommodates a local Device Manager agent that handles input/output devices and supports their interaction with any application in the ClassMATE cloud. Both remote and local devices are supported transparently by the system, as the interaction is orchestrated by the Context Manager and transported through the Events Layer.

The role of the **Data Space** is threefold: a) it implements a centralized content repository, providing transparent content access and management by any ClassMATE application and service, as if it was a local resource, b) it encapsulates a content classification mechanism, based on IEEE's LOM specification [18], providing the necessary content-related rationale to data mining procedures, and c) it encapsulates a sophisticated filtering mechanism for

personalized content delivery. For the latter to be accomplished, the Data Space strongly collaborates with the User Profile to collect the essential static or dynamic user characteristics.

The **User Profile** implements the classroom's users (students and teachers) behavior monitoring and evaluation, in order to provide user related metadata to the ClassMATE's services and applications. According to the IEEE's Learning Technology Systems Architecture (LTSA) [17], as illustrated Fig. 2, the User Profile represents a learners' record repository, which keeps track of every individual student's learning status and behavior data. Additionally, the User Profile accommodates the knowledge learning resources library of students' behavior patterns, dynamically gathered via their activity monitoring. Data gathered by the User Profile service, through an iterative monitoring and evaluation process, constitutes the main feedback for the Context Manager, so that a learner's centric rational is applied for content delivery and interaction control, thus providing adaptation to individual student's needs.

The **Security Manager** is responsible for the authorization management of the intelligent classroom's stakeholders (users and applications). It is based on a set of dynamically updated access lists, which define the rules that a user or an application must follow. The Security service is in continuous communication with the Context Manager, updating its access lists according to the current needs of the educational process. For instance, when the teacher requires the complete students' attention on the whiteboard, the Context Manager advises the Security service to suspend any interaction with other classroom's AmI artifacts (e.g., smart desks), thus students will not be distracted by the interaction with other devices.

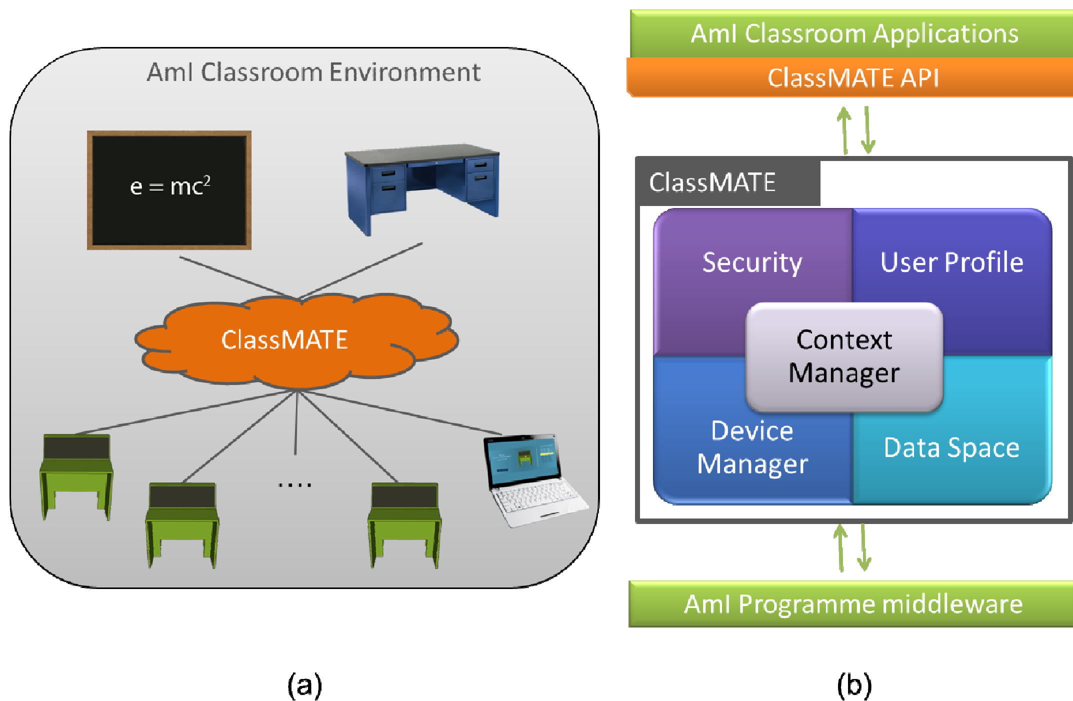


Figure 2: The Envisioned Architecture of the Smart Classroom System

In terms of intercommunication, ClassMATE relies on a generic services interoperability platform, named FAMINE (FORTH's AMI Network Environment), which has been implemented in the context of the ICS-FORTH AmI Programme. FAMINE provides the necessary functionality for the intercommunication and interoperability of heterogeneous services hosted in an AmI Environment. It encapsulates mechanisms for service discovery, event driven communication, remote procedure calls, etc., supporting a plethora of programming languages and frameworks, i.e., .NET languages family, Java, Active Script, ANCI C++, etc.

4 Ambient Environment Management

The envisioned Classroom is surrounded by an intelligent environment that ClassMATE aims to fully exploit through the Context and the Device Manager. The Content Manager encapsulates mechanisms to identify environment's characteristics and facilities, discover and manage the available services and orchestrate the applications running on the contained artifacts. The Device Manager aims to offer a generic mechanism for heterogeneous devices manipulation by any ClassMATE-enabled application; thus in collaboration with the Content Manager identifies the active devices and makes them available for use.

4.1 The Context Manager

The Context Manager consists of seven modules, as depicted in Fig 3, which interoperate to achieve context-awareness. The Platform Expert operate provides access to the wide-variety of platform-specific functions in a platform-independent manner, the ClassMATE Event System implements the essential mechanisms for event distribution, the Artifact Director orchestrates each artifact, the Class Orchestrator controls every aspect of the classroom the Application Launcher instructs the applications opening, the State Serialization Manager manages application's state serialization and deserialization, and finally the Migration Processor facilitates the application migration.

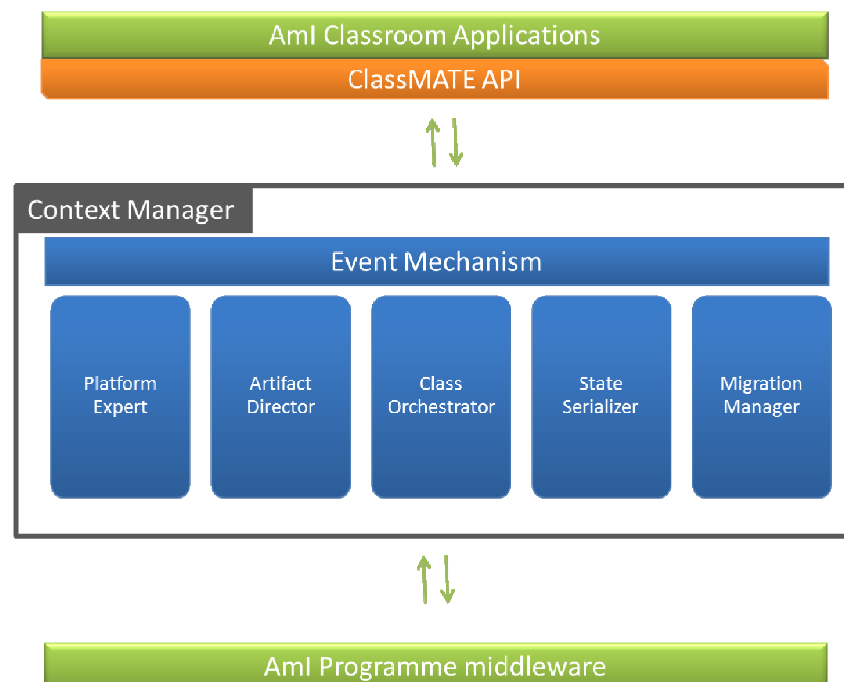


Figure 3: Context Manager Architecture Diagram

4.1.1 Platform Expert

In the envisioned technology-augmented classroom various platforms may exist. Due to the diversity of services provided by each artifact, the ClassMATE system was designed to operate as an abstraction layer that provides access to the wide-variety of platform-specific functions in a platform-independent manner; for instance, the Smart classroom intended for the FORTH Ambient Programme, consists of five different platform artifacts. Therefore, the Platform Expert module was introduced. The Platform Expert's concept is based on similar well-established abstraction layers incorporated in the majority of the modern operating systems and was implemented following the Singleton design pattern. The Singleton pattern [14] ensures that a class has only one instance and that the instance is easily accessible. A global variable makes an object accessible, but it doesn't avoid instantiating multiple objects. A better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and can provide a way to access the instance.

Every application depends on a well-defined set of services to operate properly. If the host platform does not implement the complete set, then the application would either execute poorly or would have to dynamically adapt its logic, using defensive programming techniques, to address such lack. Nevertheless, if the host platform offers supplementary services to substitute the missing ones, then the application would work properly as long as the core functionality remains the same.

Based on the above observation, ClassMATE adopts an Interface-based approach to ensure that every platform (i.e., ClassMATE enabled device) will support a minimum set of ClassMATE's core services. For each core service an API is defined that describes precisely its operations, while a concrete implementation is provided by each specific platform to ensure portability; these classes that implement the same API constitute a service family. An application programming interface (API) is an interface implemented by a software program which enables it to interact with other software. It facilitates interaction between different software programs similar to the way the user interface facilitates interaction between humans and computers. An API is implemented by applications, libraries, and operating systems to determine their vocabularies and calling conventions, and is used to access their services. It may include specifications for routines, data structures, object classes, and protocols used to communicate between the consumer and the implementer of the API.

Following the Abstract data types (ADT) programming paradigm, every application operates independently of the platform that is running on, since it does not have to know the concrete instance that offers a particular service, but only use the appropriate API exposed by that service family (as depicted in Fig. 4). ADTs are purely theoretical entities, used (among other things) to simplify the description of abstract algorithms, to classify and evaluate data structures, and to formally describe the type systems of programming languages. However, an ADT may be implemented by specific data types or data structures, in many ways and in many programming languages; or described in a formal specification language. ADTs are often implemented as modules: the module's interface declares procedures that correspond to the ADT operations, sometimes with comments that describe the constraints. This information hiding strategy allows the implementation of the module to be changed without disturbing the client programs. The notion of abstract data types is related to the concept of data abstraction, important in object-oriented programming and design by contract methodologies for software development. The following example illustrates the advantages of the Interface-based approach: any application designed for touch interaction that uses the ClassMATE's Touch Interface would operate flawlessly both when running on the vision-based touch-sensitive AmIDesk [3] and the touch screen of the SmartDesk.

Another additional advantage that derives from the interface-based approach is that despite the system's distributed nature, the locality of the required services does not affect the applications' logic. One service could run locally on the same host, or remotely on another host, or even inside the ambient environment, and the interested application will access the exposed operations through the ClassMATE-exposed interface, as if the concrete service provider was a simple source-level object. The Platform Expert, in cooperation with the ClassMATE core, encapsulates the necessary logic to transparently handle the intercommunication needs between the local and the remote node(s). An example scenario, enabled by ClassMATE's distributed services, concerns the ClassBook Reader application, which displays an electronic version of a physical course book augmented with hotspot areas that the student can select to search for relevant content (e.g. an image to see relevant multimedia) or launch an exercise application (e.g., a multiple choice exercise). It uses the book localization service to determine which page should be displayed. When running on the AmI- and Smart- Desk artifacts of the technologically-augmented classroom, the in-artifact (local) service identify the page on the physical course book using the front-facing camera and notify the reader application, while when running on a student's laptop at home

a remote book localization service can determine the appropriate page range, even the exact page, based on tomorrow's lecture schedule. In both cases, the Classbook Reader application displays the appropriate page without being aware of the concrete instance that defined it or its locality.

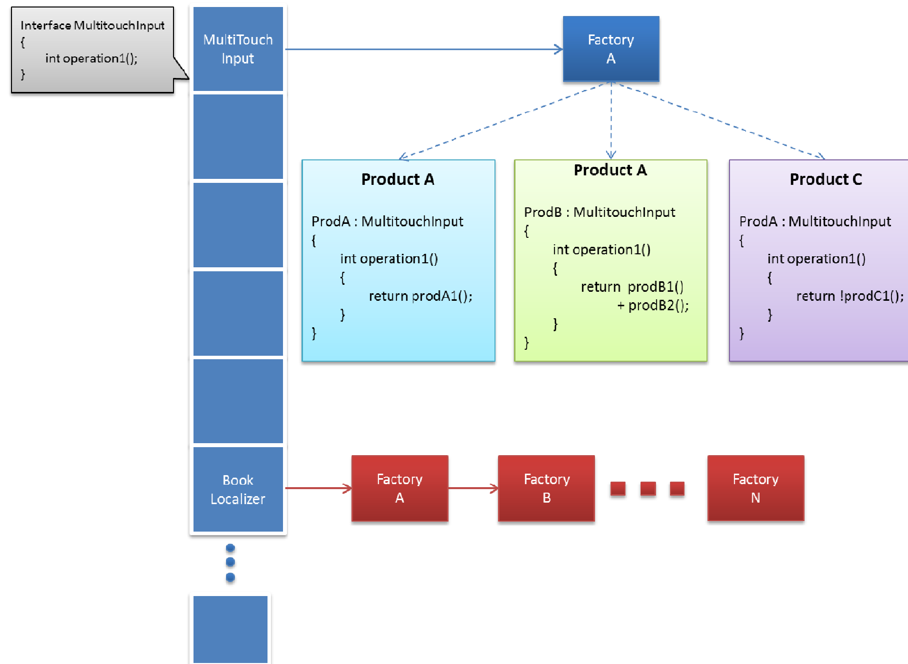


Figure 4: Interface-based Abstract Service Factory approach

The interface-based approach for each service family satisfies the requirement of application's portability. The modules that support dynamic service installation and platform configuration through external files, according to the requirement of service modularity, will be described in the next sections.

```
public List<T> resolveService<T>()
{
        List<T>  list = new List<T>();
        SrvFactoryEntry entry = services[typeof(T).FullName];

        foreach (SrvInfo si in entry.PlatformServices)
        {
                object[] createParams = { si.Name,
                                          si.Locality,
                                          si.RemoteCxtName };

                Object o = null;
                entry.FactType.InvokeMember("createProduct",
                                            BindingFlags.Default
                                            |BindingFlags.InvokeMethod,
                                            null,
                                            entry.getFactoryObject(),
                                            createParams);
                list.Add((T)o);
        }

        return list;

}
```

Figure 5: Platform Expert's method to concrete service objects

4.1.1.1 Service Factory Interface

To be portable across different platforms, an application should not hard-code its services for a particular platform. Instantiating platform-specific classes of services throughout the application discourages later modifications. For instance, explicitly declaring the service "S1", having in mind a particular platform "P1" (e.g., the SmartDesk artifact) that provides "S1", will immediately constitute the application stationery as when launched in another platform "P2" which provides "S2" as an alternative to "S1" the application will not work properly. This issue can be addressed by defining a services family "SF1" that defines the interface "SF1_Interface" which exposes the necessary operations, and every artifact should provide a concrete implementation of that interface. The application developer, at the source level, will request a concrete object that implements "SF1_Interface" and the ClassMATE core will instantiate the appropriate object for. For that to be achieved the Abstract Factory design pattern [14] is applied; the Service Factory interface declares the createProduct operation, which when invoked returns a new service object that belongs to that family (Fig. 5, Fig. 6). Subsequently, clients do no hard-code the desired services; instead they call the createProduct operation to obtain service instances. Thus clients remain independent of the prevailing platform as they are not aware of the concrete classes they are using.

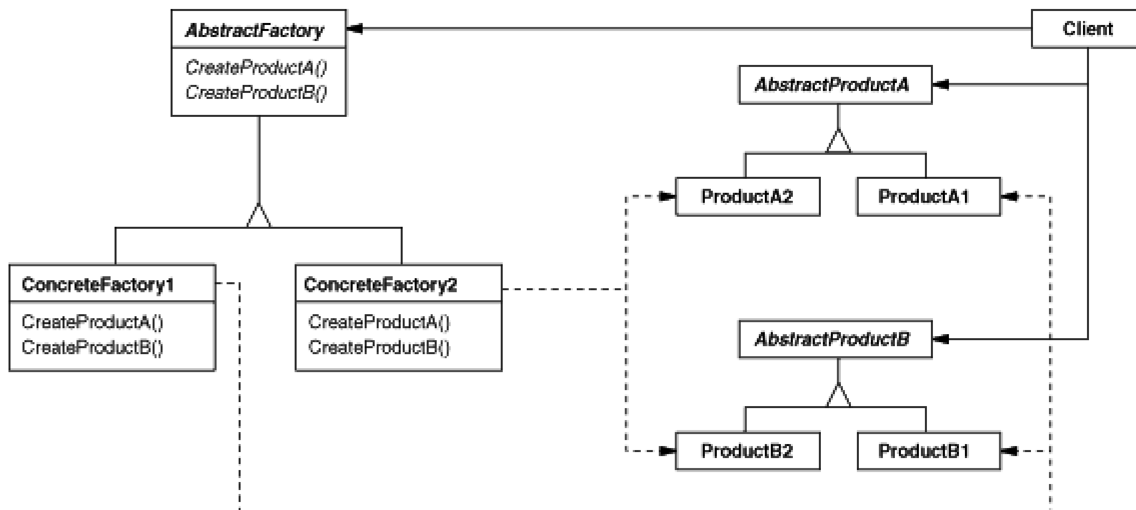


Figure 6: Abstract Factory Design Pattern

Every services' family that requests integration with the ClassMATE system should implement the ServiceFactoryInterface and provide a concrete implementation of the createProduct method to return a new service object for every platform. The createProduct method when invoked accepts three parameter arguments: Name, Locality and RemoteContextName. Name indicates which service should be instantiated, Locality determines the context where the desired service will run (i.e., locally or remotely), and finally, RemoteContextName is applicable only for remote services and indicates the node that hosts the desired service.

4.1.1.2 Service Info & Factory Entries

The Service Info class is a utility class that stores data regarding the available services on the current platform (i.e., Name, Locality and RemoteContextName). During the platform's initialization, the Platform Expert module instantiates a new ServiceInfo object for each supported service contained in the configuration, populates it with the retrieved values and adds it to the available services list.

The ServiceFactoryEntry, as implied by its name, stores data regarding the factories of supported services. Each entry contains: the literal representation of the service's full name, the factory instance that will be used to create the concrete objects and a list of ServiceInfo elements enumerating the supported services of this service family in the current platform. The Platform Expert module populates the list of available factories when loading the current platform configuration. The service's name is encoded in the configuration file, but the factory instance is not explicitly defined. The class that implements the ServiceFactory interface is implicitly defined by the service's name, and the ClassMATE system

automatically loads that class using the .NET reflection mechanism and in particular the **Activator.CreateInstance** operation. The pattern used to discover the factory class is a proprietary protocol, where the factory class should be named as ServiceTypeXFactory (e.g., for a service named "ClassMATE.Core.ServiceA" the system will lookup for a class that implements the ServiceFactoryInterface named "ClassMATE.Core.ServiceAFactory").

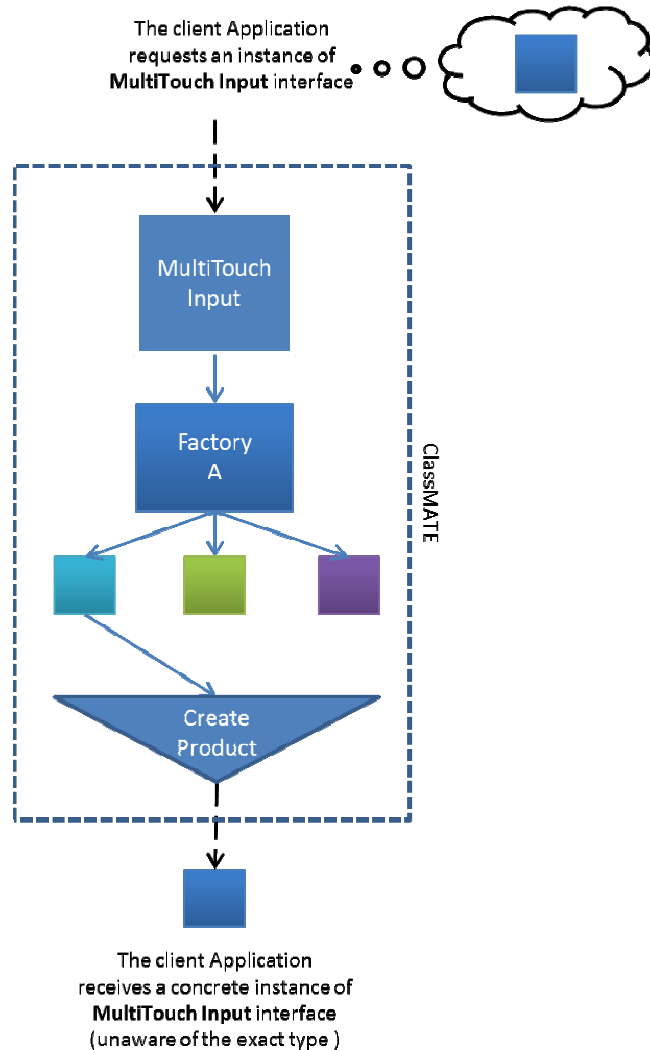


Figure 7: Process to instantiate a concrete service object

At runtime, the Service Factory Entry list is solely used by the Service Factory Registry during the service resolution process. Since the createProduct method is a key method of the ClassMATE's system, its performance should be optimal; thus, the frequently used Factory list is always kept in the main memory to ensure fast access time. When a client application requests a specific service type, the Service Factory Registry retrieves the respective factory entry, invokes the createProduct method using the contained ServiceInfo entries to supply the appropriate arguments, and finally collects all the concrete instances into a list that is

returned to the client application (Fig. 8). The objects contained in such list offer the service type requested by the application and supported on the current platform.

4.1.1.3 Service Factory Registry

The Service Factory Registry maintains a one-way associative map between a library and the service name(s) that it exposes, while a single library can expose more than one service types. The map is populated during platform's initialization with the services described in the platform's configuration. The key feature is that every family of services that exposes the same interface also offers a factory class that instantiate the concrete service objects. The full name of the exposed interface was selected to be the association key in that map, as the client applications only know the exposed interface type and they are completely unaware of the internal factory pattern used to create the concrete objects.

Taking into consideration client's objectives when resolving a service through the Service Factory Registry, the approach selected is not only straightforward to use but also minimizes errors taking advantage of the compiler's runtime checks. The "resolve" method is implemented as a Generic method, having as a method-specific type argument the interface type that the client would like to use. The type argument purpose is twofold: on the one hand it implicitly provides the lookup key, thus the caller (client) does not have to specify anything else, and on the other hand it eliminates the need for explicit cast of the returned values (Fig. 9).

```
interface TouchInputIface
{
        event CustomDelegate evtDelegate;
        void operation1(int param);
}

public class A
{
     //
     //some code
     //

    List<TouchInputIface> list = null;
    int param = 0;

    list = PlatformExpert.Instance().resolveService<TouchInputIface>();
    foreach (TouchInputIface mi in list)
    {
        mi.operation1 (param1++);
        mi.evtDelegate += mymethod;
    }

    //
    //some code
    //
}
```

Figure 8: Sample usage of the service resolution mechanism

When the “resolve” method is invoked with a type parameter T, the Service Factory Registry extracts the literal representation of that type T and resolves the Service Factory Entry that exposes that service. Using the .NET reflection mechanism, the createProduct method of the respective factory instance is invoked to create the appropriate objects. Since more than one providers might offer the same service T, the Service Factory Registry stores each individual object in a temporal list, and upon completion returns it to the caller (client).

Another feature that is natively supported by the Service Factory Registry in cooperation with the Service Factory Interface is distributed service’s locality. The platfom’s configuration encodes location-related information (locality type and remote host) and the service-specific factory instance with respect to these attributes will create the necessary products. Both a local and a remote provider might as well offer the same service T but the client will never tell the difference between them.

Portability was one of the key objectives when designing the service resolution method. Since each Service Platform Registry will use the respective configuration to load the available services and the client will not have to modify anything, portability is ensured. Every application that wished to use a particular service is not aware of the concrete instance that offers it, instead it is only aware of the interface that describes the functionality of that service. Finally, the only special case that needs exceptional handling is when no suitable service is available and the client must implement a failsafe mechanism to address that issue; this is actually part of the client’s application logic.

4.1.1.4 Configuration files (Platform.xml, Global Service Definitions.xml)

The Global Service Definitions configuration file holds the global list of the services available in the ClassMATE system independently of the platform, along with the library files that contain them. The data are encoded in an XML format and the library file is shared among the classroom artifacts; however, it is locally stored to minimize network traffic and optimize loading time. This external configuration facilitates the introduction of a new service, as by simply adding a new entry in the configuration the service will be automatically loaded in the ClassMATE system. When the new service becomes available for the applications to use, the only prerequisite is to obtain the interface that exposes its functionality. The structure of the Global Service Configuration file is as follows:

```
<ClassmateServices>
        <Service>
            <Type> The Service’s Interface full name goes here </Type>
```

```
                <Lib> Path to the service's libraryThe library's name goes here (including the .dll
extension) </Lib>
        </Service>
        <Service>
                More service definitions go here
        </Service>
</ClassmateServices>
```

The Platform configuration file is also encoded in XML format, and is used by the Platform Expert to determine which services are supported in the current platform (artifact). Apart from the services, it also contains general artifact information like name, screen resolution, etc., useful both by the ClassMATE core and by various ClassMATE-enabled modules (e.g., the Window Manager, the ICS Widget Library, etc.). This file is unique to each artifact, as the supported services and artifacts characteristics differ.

The key feature though is not the configuration of the supported services per se, but the definition of service families that follow the factory design pattern and offer a class to transparently create the alternative concrete implementations of a service T interface.

```
<Platform>
        <Info>
                <Name> Artifact's Name goes here </Name>
                <Resolution>
                        <Width> Horizontal Resolution (e.g. 1600) </Width>
                        <Height> Vertical Resolution (e.g. 800) </Height>
                </Resolution>
                Other characteristics go here
        </Info>
        <Type>
            <Name>
                The Service's Interface full name (as defined in the Global Service
Definitions)
            </Name>
            <Service>
                <Name> IdentifierThe name as defined in the Global Service Definitions
</Name>
                <Locality> LOCAL | REMOTE </Locality>
                <RemoteCxtName> The name of the remote node </RemoteCxtName>
            </Service>
            <Service>
                More service definitions go here
            </Service>
        </Type>
        <Type>
                More service type definitions go here
        </Type>
```

```
</Platform>
```

The Platform Configuration contains a complete description of the platform and being external facilitates the maintenance and upgrade process. The removal of an unsupported service can be achieved by simply removing the appropriate service element, while the addition of a new one under an existing service family is accomplished by simply adding the new child element in the appropriate section. On the other hand, the addition of an entirely new service family (interface type) is a two-step process. The first step is to declare the service in the Global Services Definitions, and the second step is to add a new service type element in the supported platforms' configuration that internally contains the concrete classes implementing that service.

4.1.2 ClassMATE Events

Coupling was first defined as "*the measure of the strength of association established by a connection from one module to another*" by Stevens, Myers and Constantine [13]. As the definition implies, coupling is inevitable in almost any software system, however it can be significantly minimized through careful system design and confined among the simplest components while the complex ones remain completely decoupled [13].

Two major coupling types exist: static and dynamic. In static coupling, if class A is coupled to class B, then class B should be available during class's A compilation for that to be successful. Moreover, if class B changes, then class A should also change to be compilable again. In dynamic coupling, if class A is coupled to class B, then class B should only be available during runtime, while class A individually compiles successfully. Since class B does not have to be present at compile time, any changes do not break the compilation of class A.

Among these types of coupling, the static one appears to be more problematic, in the sense that compile-time dependencies are essential. However, these dependencies make static coupling safer than dynamic, as the compiler makes the necessary type checking. On the other hand, dynamic coupling does not have any compile-time dependencies but any coupling related errors would only show up during runtime, thus hindering the debugging process.

Apart from the aforementioned coupling types, three orthogonal coupling flavors can be identified: Logic, Type and Signature coupling. Logic coupling is the most abstract and the least desirable of the three, as it means that two classes share information or make

assumption about each other (i.e., class A contains an algorithm that is related to an algorithm contained in class B or classes A and B contains a literal value that is used for the same conceptual purpose). Type coupling is the most recognizable form, and arises when a class A directly uses a type defined in another class B. Finally, Signature coupling occurs only at run time and has the potential to decouple classes from each other (i.e., C++ function pointers or C# delegates, as long as the method's signature is compatible then the caller does not know the receiving end of the call the callee).

In general, in terms of development cost, system scalability and maintenance, dynamic coupling accompanied by Signature coupling is an optimal choice, especially in a fully dynamic system such as ClassMATE.

The issue arises therefore to identify a programming paradigm / tool / methodology that can offer the desired features. Such a programming paradigm is event-based programming. An event is a detectable condition that can trigger a notification, while a notification is an event-triggered signal sent to a run-time–defined recipient. A software system is said to be event-based if its parts interact primarily using event notifications [13]. An event-based system not only reduces the overall complexity, making design, development, testing and maintenance easier, but most importantly reduces the coupling of the system as the modules do not have to be aware of each other but only to "listen" and correspond to the appropriate events. Since extensibility is one of the main ClassMATE objectives, and since high coupling discourages extensibility, it was designed and implemented as an event-based system to reduce coupling to the minimum.

An event is the detection of a condition that sends a notification ("fires the event") to the interested parties to trigger their reaction. The node that detects the condition is known as the event publisher, the event source or simply the sender, while the receiving end of the event is the event subscriber or listener or event handler or simply the receiver. From the above becomes clear that for an event to be "fired" at least one subscriber is required. The process of establishing a valid set of receiver is called subscription or registration [13]. Finally, during registration a well-known list of publishers is necessary for the receivers in order to register themselves appropriately.

ClassMATE's event mechanism is implemented on top of .Net built-in Event Mechanism and extends it to offer the necessary functionality. Following the .NET Framework approach, stating that if the event handler requires state information, the application must derive a

class from the EventArgs class to hold the data, a hierarchy with specialized event types derived from general ones was defined, forming the ClassMATE event type system [13]. Subsequently, a set of core event types, named ClassMATE commands, were defined: AppCmd, MigrateCmd, MimeCmd, UpdateCmd. Their exact objectives and internal mechanics will be described in more details later on.

During receiver registration, as mentioned above, the list of the publishers should be known. However, in ClassMATE this is not possible, since the list of publishers could be modified at any time. As a result, the need emerged for a dynamic and scalable solution. A variant of the Proxy design pattern [14] based on events was selected. The pattern's intention "*Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically*" [14] fits perfectly the needs of ClassMATE. The Proxy Design Pattern is used when an object, called the Proxy, is interested in knowing about events that occur in another object, called the subject. The Proxy tells the subject which events it is interested in. When the subject detects one of these events, it sends an event notification to the Proxy, often by calling one of the Proxy's methods. A subject can theoretically have any number of Proxies.

Utilizing the Proxy pattern, any interested party does not need to know the exact producer of an event, but only register a callback method (by delegating) to a specific event type. When an event of this type occurs, the Proxy pattern ensures that every registered callback will be invoked.

4.1.2.1 Base Event Args

Event-related data make ClassMATE events useful, and since the .NET EventArgs class does not contain event data, ClassMATE defined its own base class from which every ClassMATE event will derive from, the BaseEventArgs. This class obviously derives from the .NET EventArgs class to ensure seamless integration with the .NET framework, and introduces two quite important attributes, common to all ClassMATE events, the event Sender and the event identifier. Since multiple event handlers can be triggered by the same event, each handler determines whether an event was intentionally received and should be handled or it should be discarded by checking if this handler is interested to events coming from the particular Sender.

4.1.2.2 Abstract Event producer

An Event Producer could be considered as a special-purpose service that exposes a certain public API and fires (produces) events. Setting aside the event type, every event producer must offer the same functionality. That API exposes the methods that facilitate (i) new subscribers' addition, (ii) existing subscribers' removal, and (iii) invocation of the currently registered delegates.

To ensure that all event producers will implement that fundamental API, the AbstractEventProducer class was defined to accumulate those common tasks. It is implemented as Generic class to support any Event type that derived from BaseCmdEventArgs and encapsulate operations that are not specific to a particular data type, and declared as abstract to permit the declaration of the abstract method "isLocal" with no implementation that should be overridden by any derived class. Every ClassMATE-aware Event Producer should derive from this class and must provide implementation for all abstract methods. The "isLocal" method differentiates a Local from a Remote Event producer, and is mainly utilized by the Event Registry during event broadcast process. Finally, each Event Producer can enhance the base class with event-specific functionality, as long as the main interface remains unchanged.

Finally, the key feature of the Abstract Event Producer is the EventHandler delegate that it contains, which will be called upon the occurrence of some "event". The delegate can have one or more associated methods that will be called when your code indicates that the event has occurred. Since direct access to this member is limited, the OnEvent method was defined that takes the Custom Typed Event as a parameter and distributes it through that delegate to the subscribers (Fig. 10).

```
public class EventRegistry
{
      //
      //some code
      //

     private List<AbstractEvtProducer<T>>    producersList;

     public void sendEvent(object sender, T t)
     {
          foreach (AbstractEvtProducer<T> item in producersList)
          {
               if(item.isLocal())
                     item.OnEvent(sender, t);
               else
                     //conditional code
          }
     }
     //
     //some code
     //
}
```

Figure 9: Generic Send Event method

4.1.2.3 Event Proxy

ClassMATE, in addition to dynamic service registration, supports application and service migration from one node to another at run time. Migration refers either to the UI or the application logic or both. Concerning the event handling mechanism, even though the above cases suffer from different problems, the main common difficulty is that every event producer(s) and receiver(s) change during migration. As a result, the application/service should be aware of the migration process and keep track of all the event producers that it uses, so as to update them if they change their content after initialization (during execution). Pure event producers do not have to know anything about context or migration, since they only produce events and the receivers are responsible for successfully receiving them (not such a good practice). However, event producers that are dependent on other event producers, so they are actual event receivers, should follow the same approach.

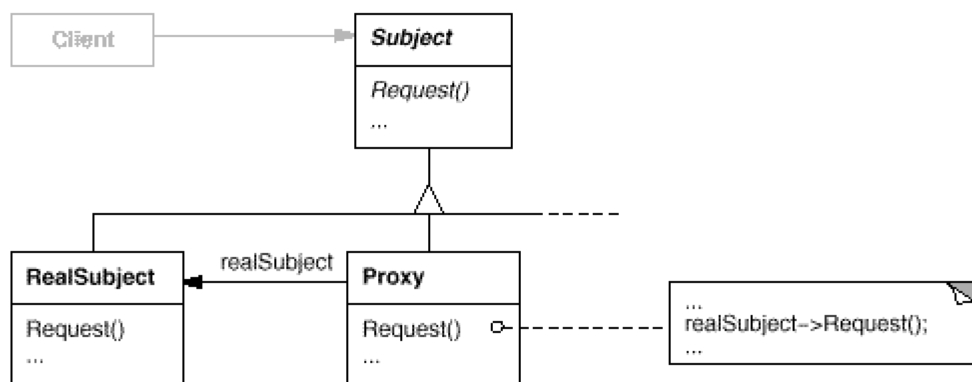


Figure 10: The Proxy Design Pattern

This is not compatible with the modular nature and ease of use of ClassMATE. Since ClassMATE is responsible for launching / suspending services in any ClassMATE node, then clearly it should be responsible for a transparent migration process. Migration if examined closely amounts to suspending the application / service at some node, save its state, transfer the state to the new location and resume the same application / service with the saved state at another remote node. The state transition will be examined in more details in the StateSerialization API.

The state itself characterizes the application, however the rest of the services / application present in the classroom ecosystem do not need to know anything about the state, as they are only interested in the event produced. For handlers to be installed, the producer should be implicitly known as an AbstractEventProducer of the event type A. Even so, when the

producer instance alters, the abstraction is invalid. The new AbstractEventProducer should be registered and the handlers re-subscribed. To automate and transparently handle this process, the EventProxy class was defined to minimize the necessary boiler plate code.

The EventProxy (Fig. 11) was implemented as a Generic class that allows the definition of type-safe data structures, without committing to actual data types, and facilitate reuse of data processing algorithms without duplicating type-specific code. Internally it maintains two lists, the handlers' delegates and the producers list. The handlers list contains all the event delegates registered for a certain type of event and that must be invoked when the event occur. Apparently, this list is not used for invoking the delegates, but for keeping track of the installed handlers, so as to re-subscribe them if a migration occurs. The producers list contains all the AbstractEventProducers, registered in the ClassMATE, that produce events of type T. ClassMATE supports dynamic service registration through the PlatformExpert and ServiceRegistry modules. To enable integration with ClassMATE, every event type T must be accompanied by a module that implements the ServiceFactoryInterface and creates the respective AbstractEventProducer for that event type.

When the migration process is complete, the Event Proxy is notified to update its' invalidated list of producers and re-subscribe the handlers to the new producers, so as to be notified when an event occurs. This process is completely transparent to the handlers since they would never know the exact producer so they will also not know that the producer has changed. They are only aware of a particular event type and they have only offered one method to be invoked when an event of that type occurred. The process of adding a new delegate is accomplished through the EventRegistry module that will be described next.

The Event Proxy also contains a SendEvent method that broadcast the event to the registered subscribes. For that to be achieved, Event Proxy iterates the list of current AbstractEventProducers and invokes their OnEvent method (as defined in the AbstractEventProducer base class), and as a result the event distributes. It is important to note that at before the invocation any migrated producers had been re-initialized and as a result the list is updated (Fig. 12).

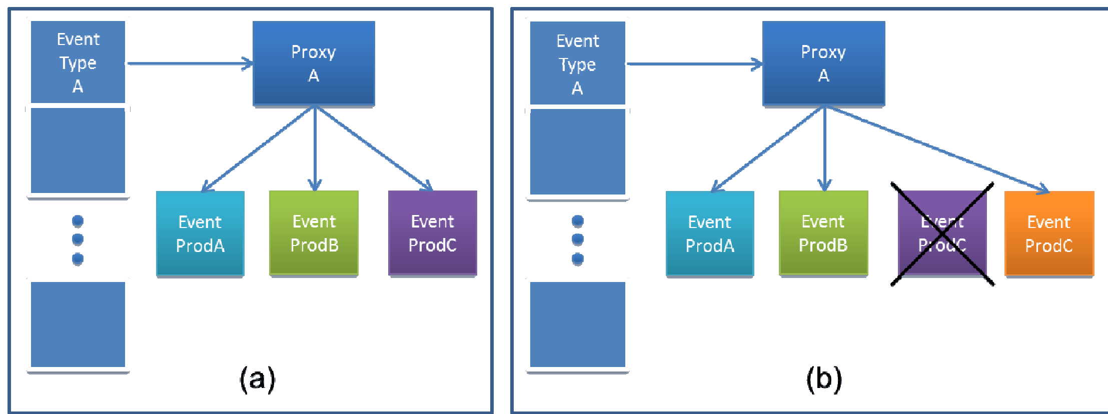


Figure 11: The Event Proxy Rationale

4.1.2.4 Event Registry

The EventRegistry module is the sole entry point for the various applications and services to the ClassMATE's event mechanism. Both the handlers' subscription and events' distribution processes are accomplished through that point, and considering that a unique instance of it should exist in the same artifact, the Singleton design pattern [14] is used for its implementation.

The subscription process is performed by the AddHandler and RemoveHandler exposed to the Event Registry API. Every ClassMATE-enabled application or service that desires notifications for events should subscribe the relevant delegate to be invoked by the producer when the event condition is satisfied. A delegate is a type that references a method and is similar to a C++ function pointer. Once a delegate is assigned a method, it behaves exactly like that method. The delegate method can be used like any other method, with parameters and a return value. Any method that matches the delegate's signature, which consists of the return type and parameters, can be assigned to the delegate. This makes it possible to programmatically change method calls, and also plug new code into existing classes. As long as the delegate's signature is known, it can be assigned its own delegated method.

As the above suggests, an event handler class should implement a method that matches the delegate signature for the desired event. Despite that such a programming practice would not compromise scalability -as each event type will define its own signature- it would perplex maintainability. If a single delegate signature changes, then all the relevant handlers should update their implementation to comply with the new specifications. ClassMATE, aiming to enforce uniformity, utilizes the default EventHandler delegate offered as a built-in facility of the Microsoft's .Net Framework. The default EventHandler is a predefined

delegate that specifically represents an event handler method for an event that does not generate data. Since ClassMATE's events hold data, the Generic version of that delegate is used, the EventHandler<TEventArgs> delegate class, where the Generic type parameter is substituted by the appropriate subclass of the Base Command Event Arguments.

The standard signature of an event handler delegate defines a method that does not return a value, whose first parameter is of type Object and refers to the instance that raises the event, and whose second parameter is derived from type EventArgs and holds the event data. If the event does not generate event data, the second parameter is simply an instance of EventArgs. Otherwise, the second parameter is a custom type derived from EventArgs and supplies any fields or properties needed to hold the event data.

Despite the fact that the Event Registy does not use Generics at all, the AddHandler and RemoveHandler methods were implemented using method-specific Generic type parameters [26]. The different event types are stored in a Dispatch Table that maps an event type with an Event Proxy. When any of the Add or Remove Handler methods is invoked specialized with a particular event type, the event type is the key that indexes the appropriate Event Proxy that has to either subscribe or unsubscribe the provided delegate from its active subscribes list. The event distribution mechanism works in a similar manner. When the SendEvent method is invoked with the specialized event type, the Event Registry locates the appropriate Event Proxy and delegates the distribution process to it by invoking the Proxy's SendEvent method that notifies event's subscribers (Fig. 13). The aforementioned process highlights the importance of the Event Proxy class, as neither the subscribers nor the Event Registry are concerned with any migration-related activity, because the migration logic is encapsulated in each Event Proxy instance.

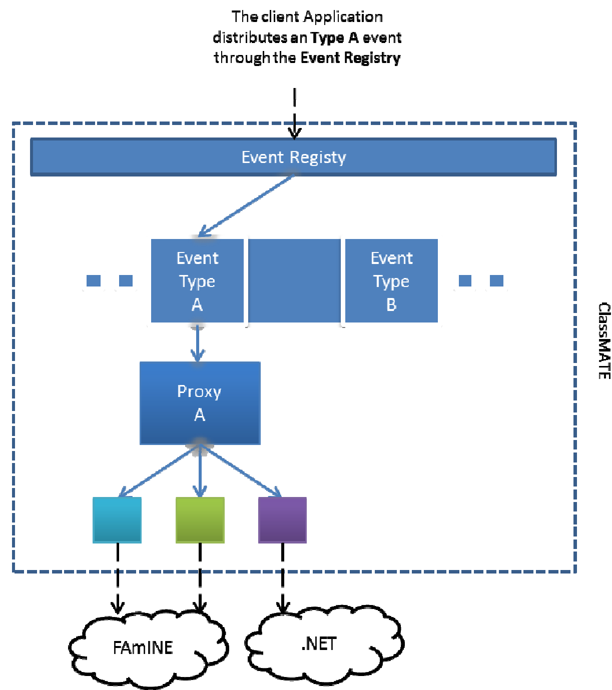


Figure 12: Event Distribution Mechanism

4.1.2.5 ClassMATE Message Events

The ClassMATE Message Event is a primitive event type that is exchanged between either local or remote nodes. It derives from the BaseEventArguments and extends it by adding the Receiver, the RemoteContextName and the Message attribute. The Receiver denotes the intended recipient of that message, the RemoteContextName is used by the FAMiNE infrastructure to facilitate the artifacts communication and dispatch the event to the proper node and finally the Message attribute contains the actual message that must be exchanged. For the Message Event to be successfully dispatched a conditional delivery is necessary; thus the EventRegistry checks the RemoteContextName, and if it is empty, then the event should be dispatched locally, otherwise it should be forwarded to the FAMiNE infrastructure for delivery.

4.1.2.6 ClassMATE Commands

4.1.2.6.1 Base and RemoteBase CommandEventArgs

The ClassMATE Command Events are divided into two major categories: the intra-artifact or local and the inter-artifact or remote events. The intra-artifact events “travel” within the same artifact since they are raised and handled by different threads of the same local process; consider the Multimedia Application, used to display multimedia content (i.e., images and videos) that receives an event from the SmartDesk ClassBook Reader Application. The inter-artifact events, on the other hand, “travel” between the various

classroom artifacts (e.g., from the AmIDesk Multimedia Application to the SmartBoard Multimedia Application, etc.) since they are raised from a thread of the local process and handled by a thread of a remote process in another artifact. The following scenarios illustrate an intra- and an inter- artifact event. An intra-artifact event occurs when the BookLocalizer service of the AmIDesk detects the current physical book page and instructs (by raising a local event) the AmIDesk Window Manager to display that page in the Classroom Book application. On the other hand, an inter-artifact event could occur when the student selects the migration of the Multiple Choice Exercise Application from the AmIDesk to the SmartBoard, and the ClassMATE instance of the AmIDesk communicates with the remote Window Manager of the SmartBoard to initiate and execute the migration process. ClassMATE in order to support both intra- and inter- artifact events introduced the BaseCommandEventArgument and the RemoteBaseCommandEventArgument classes respectively.

The BaseCommandEventArgument is primarily a ClassMATE event, thus it derives from the BaseEventArguments and extends it by adding the Receiver attribute. Since multiple event handlers can be triggered by the same event, each handler determines whether an event was intentionally received and should be handled or it should be discarded by checking the Receiver attribute.

The RemoteBaseCommandEventArgs extends the BaseCommandEventArguments by adding the LocalContextName and the RemoteContextName attributes. The RemoteContextName is used by the FAMiNE infrastructure to facilitate the artifacts communication and dispatch the event to the proper node, while the LocalContextName is used by the remote handler to cross-check the validity of the received command. The following example highlights the usage of these attributes. A student instructs the migration of the local Multimedia Application to the SmartBoard without prior teacher's permission. The RemoteBaseCommandEventArguments attributes populates as follows: "StudentDeskId" as the LocalContextName and "SmartBoard" as the RemoteContextName. Upon receipt, the SmartBoard's handler checks if the LocalContextName is permitted to execute the migration command through the Security Manager. If the desk identified by the "StudentDeskId" does not have the necessary privileges, the command will be discarded and no action will be taken.

4.1.2.6.2 Command Types and Objectives

Based on ClassMATE's requirements, the possible commands were categorized under the following domains (Fig. 14): Application, Update, Mime and Migrate. This categorization's role is twofold. On the one hand, from a conceptual point of view, such a command hierarchy disambiguates the purpose of each type and facilitates the overall design process, as every application designer has to deal with the a subset of system events only. On the other hand, from a technical point of view, this command type system based on subclassing is preferred over a more naïve approach where all the available commands form a complex union type and every instance declares its type. Moreover, every handler is interested in less than two events, thus it does not have to know redundant information regarding the other types and to check during runtime if the event is of the correct type. Finally, using different event types, ClassMATE eliminates flooding of the communication channel due to broadcasting every event to every subscriber (even those that are not interested in that particular event, not known at that time), but it implements a more sophisticated mechanism, where only the interested parties get notified (Fig. 15). The objective, the locality and the rationale of the various ClassMATE command types are described in more details in the next sections.

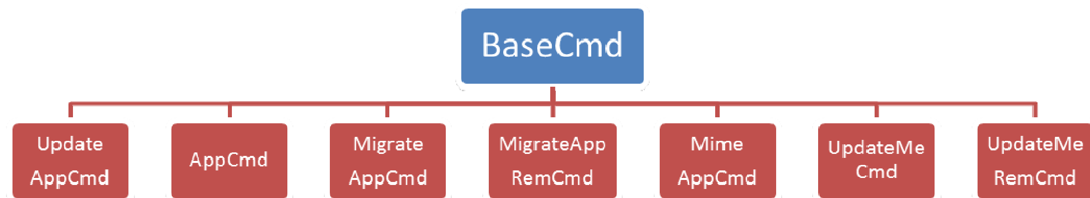
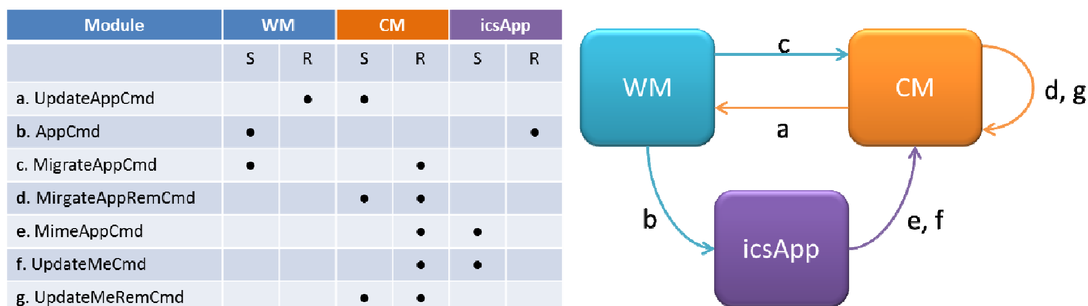


Figure 13: ClassMATE Command Type Hierarchy



Module	WM		CM		icsApp	
	S	R	S	R	S	R
a. UpdateAppCmd		•	•			
b. AppCmd	•					•
c. MigrateAppCmd	•			•		
d. MirgateAppRemCmd			•	•		
e. MimeAppCmd				•	•	
f. UpdateMeCmd				•	•	
g. UpdateMeRemCmd			•	•		

Figure 14: Sender and Receive Map of Command Events

4.1.2.6.2.1 Application Command

The application command, as implied by its name, aims to change the internal state of an application. The receiver is always a ClassMATE-enabled application and the sender is always the only component aware of any application instances, the Window Manager [22]. Since both parties reside locally in the same artifact, the Application Command derives from the BaseCommandEventArgs and introduces two additional parameters: the ApplicationName and the Argument.

An application command though is not issued by the Window Manager, but by the ClassMATE core. Such commands are mainly triggered by ambient environment factors and the core encapsulates them into expressive packages. Since the core does not have direct access to the applications, the Window Manager becomes the middleman for their delivery. However, the Window Manager might receive an application command because it is a ClassMATE-enabled application too, as long as it has registered the appropriate delegate (i.e., disable desk interaction as the teacher requires full attention to the board).

Multiple handlers may receive the same event simultaneously. The ApplicationName attribute is used to determine whether this command refers to the current handler and should be executed, or it should be discarded without modifying the handler's state. The argument encodes the actual command for execution. To address the encoding issue, a straightforward but extensible application-independent protocol was defined, including the most commonly used commands. The built-in commands are:

- **OPEN--FilePath**: Open the file indicated by the provided complete path
- **STATE--StateId**: Restore the state indicated by the state identifier (used during migration)
- **TERMINATE--Apps**, where Apps contains a list of comma-separated application names or the keyword ALL: Terminate the denoted applications
- **DISABLE--Apps**, where Apps contains a list of comma-separated application names or the keyword ALL: Disable the user interaction with the denoted applications
- **FOCUS--AppName**: Bring the denoted application to the front

Multiple applications can accept and decode the same command, but the interpretation and reaction will be application-specific. For instance, both the Multimedia Viewer and the Classbook Reader applications can decode the command "Open--FileName.xml", but each one will react appropriately. Since the ClassMATE core is the only issuer of every application command, any protocol modifications (i.e., a new ambient event triggers an already defined

action) or additions (i.e., a new ambient event introduces a new command) affect only the core's command generator module and the respective handlers' delegate method.

An indicative example that illustrates the usage of the application command type is the following: when a student opens the physical book to a particular page, the ClassMATE core identifies that page and launches the Classbook Reader application to display its electronic copy.

4.1.2.6.2.2 Update Application Command

The ClassMATE core uses the Window Manager as the middleman to distribute commands to the applications. The Update Application Command wraps a list of Application commands in a single package which the Window Manager later propagates to the receivers in the form of single Application Commands. The Update Application Command resides in the local artifact as the route of this event is always from the core to the Window Manager. Thus, it extends the BaseCommandEventArgs class and appends the dynamic list of Application Commands attribute.

To ensure that all the contained commands are delivered successfully to their recipients with zero losses, when an Update Application Command is received, the Window Manager examines every Application Command, launches the application pointed by the AppName attribute if inactive, and when notified that the application launched successfully, propagates the command for execution.

Returning to the aforementioned book example, from the point of view of the Window Manager, it can be formulated as follows: the ClassMATE core creates an Application Command, packages it into an Update Application Command and propagates it to the Window Manager. If the Classbook Reader application is not active, the Window Manager launches it and then propagates the command; otherwise, it simply propagates the command. ClassMATE in cooperation with the Window Manager (through the Update Application Commands) controls the applications and performs the overall classroom's orchestration.

4.1.2.6.2.3 UpdateMe and UpdateMeRemote Application Commands

During migration the state of an application is transferred from the local to a remote artifact, but the "new" application executes in an isolated standalone mode. Nevertheless, many cases exist in an ambient environment where the parent application should be able to control the migrated one, and vice versa. Instead of re-implementing a remote desktop protocol, ClassMATE offers a more sophisticated solution where only the necessary data are

transmitted to synchronize the remote state. For that to be achieved, the parent application, being aware that a migrated instance exists, digests any local events and transmits them when necessary. However, since no direct communication channel exists between them, the only available route is through the ClassMATE core, which encapsulates the transmission process.

The UpdateMe Application Command is exchanged between an application instance and the ClassMATE core, thus it derives from the BaseCommandEventArgs and packages two additional attributes, the AppName and the Argument. The UpdateMe and the Application Command seemingly appear the same in terms of enclosed data; however, the involved parties differ and most importantly, the Argument contained in an UpdateMe command is encoded in a proprietary, application-specific, protocol which only that particular application can handle. An application is able to use one of the predefined system protocols (e.g., the Application Command Protocol described above) to transfer messages to its remote instance as long as the protocol is suitable for its needs. When the ClassMATE core receives such a command, it firstly consults the Security Manager and the Classroom Orchestrator to validate sender's privileges and if that requirement is met, the command is wrapped into an UpdateMeRemote Application Command and marked as ready for transmission.

The UpdateMeRemote Application Command is the carrier of every UpdateMe command from one artifact to another. The transmission channel is a combination of .NET Events and FAMiNE proprietary format, while the necessary marshaling and unmarshaling processes are completely transparent to the receiver. Every UpdateMeRemote Application Command derives from the RemoteBaseCommandEventArgs, since the local and remote context attributes define the involved network nodes, and adds a single attribute, the UpdateMe command to be transmitted.

Upon receipt, the remote ClassMATE instance maps the UpdateMe Application Command into an Application Command. No sophisticated logic is needed, as the command attributes were appropriately filled by the parent application which initially generated it. The translation into an Application Command ensures that no additional handlers should be registered for that particular command type, since the already registered Application Command handler will receive and handle it by incorporating the appropriate logic. The generated Application Command is eventually delivered to the target application wrapped in an Update Application Command handled by the respective Window Manager.

The following example illustrates the use of the UpdateMe and UpdateMeRemote Application Commands. The teacher asks one of the students to continue his Multiple Choice Exercise at the AmIBoard. From that point forward, every action on the desk artifact (i.e., multiple question answer) is locally handled, and only the digested outcome is packaged and transmitted by the desk's ClassMATE instance to the board's ClassMATE instance.

4.1.2.6.2.4 Mime Command

Through the ClassMATE core, the ambient classroom may launch various educational applications. However, every application can launch other applications either directly or indirectly, to enhance the educational process and facilitate students' study through their collaboration. In shell-based approaches, well-established practices to address application intercommunication, the parent application launches a virtual shell and executes a set of commands with the appropriate initialization parameters to launch another application. The shell-based approach is particularly suited for systems that host a great variety of applications (i.e., a complete OS), however, in the ClassMATE's case, it would most likely compromise its clearness, as every application instead of complying with a common scheme, would arbitrarily define its own proprietary scheme and the other applications should incorporate specific logic for each one. Besides, ClassMATE's educational objectives require a preprocessing step to authorize on the one hand the application launch and on the other hand ensure personalized content delivery based on context-related reasoning. To highlight the added value of the above, consider the following example: the student selects to launch the Multimedia Application in order to play a specific video, but ClassMATE collects also other educational material (images, words, exercises, etc.) related to that particular video and proposes them to the student.

For that to be achieved an intermediate layer was introduced, that orchestrates the launch process and relieves the application from handling the reasoning process, i.e., communicate with the respective modules and provide the relevant data to facilitate mining. The most appropriate host of such module is the ClassMATE core, as it not only orchestrates the overall classroom, but is also in direct communication with other key modules such as the DataSpace and the Security Manager. ClassMATE uses a variant of the Mailcap invocation scheme [6] that modern Operating Systems use to associate specific applications with specific mime types, and when an application has to launch a new application it should only notify the core and supply the appropriate mime.

An Internet media type, originally called a MIME type after MIME (Multipurpose Internet Mail Extensions) and sometimes a Content-type after the name of a header in several protocols whose value is such a type, is a two-part identifier for file formats on the Internet. MIME is short for Multipurpose Internet Mail Extensions and it was defined in 1992 by the Internet Engineering Task Force (IETF). MIME is a specification for formatting non-ASCII messages so that they can be sent over the Internet. Many e-mail clients now support MIME, which enables them to send and receive graphics, audio, and video files via the Internet mail system. There are many predefined MIME types, such as GIF graphics files and PostScript files. It is also possible to define your own MIME types.

The application communicates with ClassMATE through a special command type, the Mime Command. This command is exchanged between local modules, thus it derives from the BaseCommandEventArgs and two essential attributes to fulfill its objective: the Mime type and Resource Identifier. The Mime type belongs to the commonly known mime types and the identifier is used as reference for searching relative content.

The key feature is that the ClassMATE core supplies the necessary data (i.e., mime types and identifiers) for every interactive resource and the application simply packages them in a Mime Command when triggered by the user. ClassMATE when it receives a MIME Command, initially resolves the application that can handle that mime type (i.e., png images) and then requests from the DataSpace module to search for related content. Upon successful discovery, the DataSpace stores the locations of the discovered data in a file and returns its path to ClassMATE. The generated file is structured in a mime-specific format which can be read by the applications able to handle that particular mime type. The ClassMATE core firstly packages the application's name and the file path in an Application Command, then wraps the Application Command in an Update Application Command and finally dispatches it to the Window Manager to arrange application's launch. Through a Mime Command an application can not only launch another application, but also dynamically modify its content if already launched.

An indicative scenario that illustrates the described process is the following: the student interacts with the Classbook Reader and selects an image contained in the course book asking for relative content. The Classbook Reader constructs a Mime Command, populates the Mime type attribute with the image's type value (e.g., "image/png") and the ResourceIdentifier with the URI of that particular image, and sends the command to the ClassMATE. Upon receipt, ClassMATE resolves the Multimedia Application and asks from the

DataSpace to discover related content. As soon as the relative content is successfully gathered, the ClassMATE notifies the Window Manager to launch the Multimedia Application and dictate its content population based on the discovered data. Ultimately, the student will be able to interact with the Multimedia Application and browse those images.

4.1.2.6.2.5 Migrate and MigrateRemote Commands

Many educational methods are based on collaborative learning. For that to be achieved in a traditional classroom, the teacher should firstly spend some time copying the relevant data at the blackboard and then start lecturing. On the other hand, students do not care for the actual lecture but they are trying to copy the contents of the board to their notebooks to study them later. Moreover, traditional means discourage collaboration in interactive media (i.e., videos, images, games etc.) as their rendering to a blackboard is an extremely complex task if not impossible. In the technology-augmented classroom, native support is offered to the concept of collaborative education with minimum software overhead and no additional hardware cost. Finally, since the content is stored in a redistributable digital format, the students do not have to copy anything, but can concentrate on the lecture.

The Window Manager [22], in close cooperation with ClassMATE, makes the whole process completely transparent to the rest of the system, using Migrate Commands. A Migrate Commands due to its local exchange derives from the BaseCommandEventArgs and the two added attributes are: the ApplicationName, MigrationContext and the State Identifier. The ApplicationName apparently defines the application that should migrate, the MigrationContext defines the remote node where the application should launch at, and finally the StateId is used by the remote application, when launched, to restore its state through the StateManager module.

For application migration to be achieved, the application's current state must be transferred and resumed to the other side. The StateManager maintains a map structure indexed by the StateId which associates an application with its saved state data and, in cooperation with the DataSpace, transfers and restores that state in the remote node. The migration process will be described in more details in later on.

Every user can transfer his work anywhere in the classroom (usually at the classboard) with a single gesture through the Pie Menu [22]. The Window Manager identifies the gesture, resolves the focused application and initiates the migration process. Initially, the Window Manager requests from the StateManager to store the application's state and return its unique id and then instantiates, populates and propagates a Migrate Command to the

ClassMATE. The local ClassMATE instance, upon receiving the Migration Command, confirms authorization privileges with the Security Manager and the Global Classroom Orchestrator. If successfully authorized, the saved state (indexed by the stateId) is copied into a temporal network repository for retrieval by the remote ClassMATE instance. ClassMATE has no control over the state data, but it simply manages the state transfer from one node to the other. Finally, a RemoteMigrate Command is instantiated and dispatched, through FAMiNE, to the appropriate remote node as indicated by the MigrationContext attribute.

Similar to the UpdateMeRemote Command, the MigrateRemote Command is the carrier of every Migrate Command from one artifact to another. Every MigrateRemote Command derives from the RemoteBaseCommandEventArgs since the local and remote context attributes define the involved network nodes, and adds a single attribute, the Update Application Command that holds an Application Command populated appropriately with data digested from the Migrate Command (i.e., application's name and STATE--StateId).

Upon receipt, the local ClassMATE core lays the groundwork for a successful migration. For that to be achieved, the ClassMATE core copies locally the state data stored at the temporal network repository pointed by the "StateId" encoded in the contained Application Command. As soon as the copy process completes, the Update Application Command is forwarded without any further processing to the local Window Manager to launch the relevant application. On successful initialization, the application will receive an Application Command that instructs its state restoration and will use the supplied identifier to load the appropriate one from the StateManager module.

The aforementioned example, where the teacher asks one of his students to continue his Multiple Choice Exercise at the AmIBoard, presupposes that the migration process for the Multiple Choice Exercise Application is successfully completed and that the application migrates with its state from the student's AmIDesk to the AmIBoard.

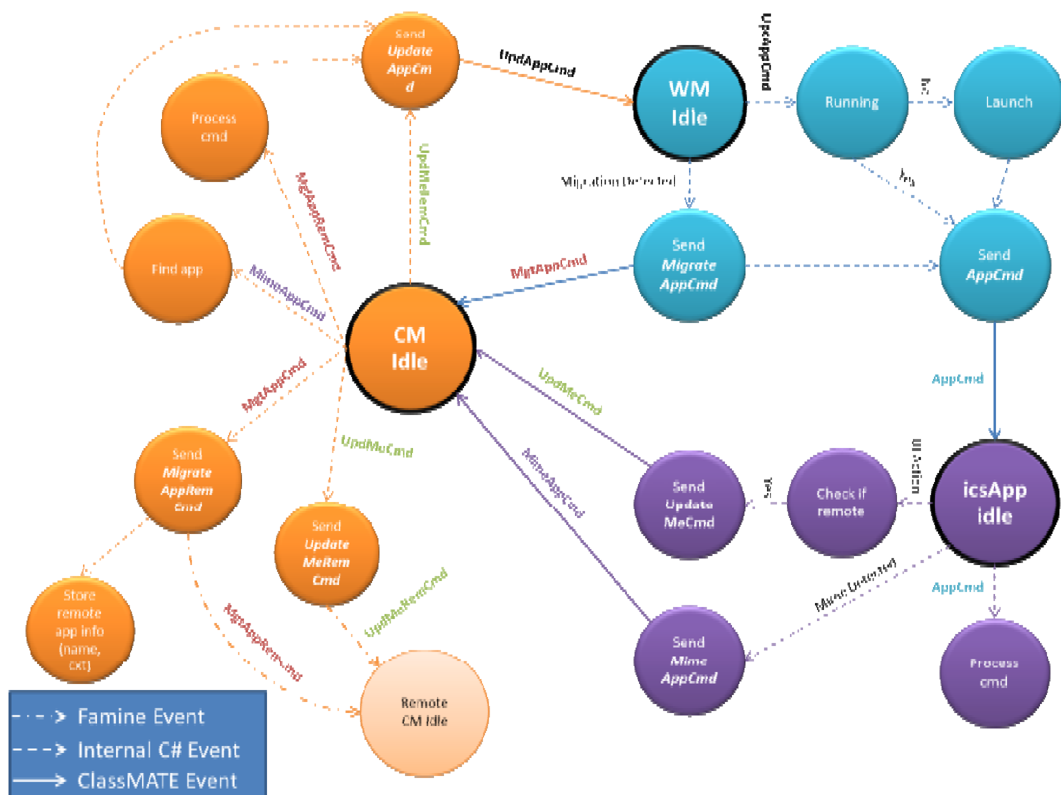


Figure 15: ClassMATE Command journey

4.1.3 Artifact Director

The Artifact Director is the context aware module that orchestrates each artifact. Since only a sole instance of such module should exist per artifact, the Singleton design pattern was used for its implementation. The decisions made and action taken can be either artifact-oriented, independent of the rest of the classroom, or global guidelines coming from the Class Orchestrator. In addition to the Class Orchestrator, the director is in direct communication with the other ClassMATE core modules (i.e., Security Manager, Data Space, Device Manager, etc.). The communication with the Class Orchestrator ensures that situations concerning the overall classroom ecosystem (e.g. the latest teacher's directives or environmental stimulations) and not only the artifact, for which the specific director is responsible, will be taken into consideration when managing the operation workflow and the collaboration among the available services (i.e., initiation, suspension or termination).

Context awareness is the key feature that facilitates the "smart" decision making process, since the Artifact Director and Class Orchestrator should take into account current context attributes, both static and dynamic, to make the respective decision. "Smart" decisions based on static attributes could include the termination of applications unrelated with the current course or automatic homework submission, while decisions based on dynamic

attributes might include access restriction to the help system (i.e., the Dictionary application) during examinations or disabling interaction with every application when the teacher requires the students' full attention to the board. The low-level subsystems should communicate with the Security Manager and the Global Orchestrator to make these decisions.

In addition to being the artifacts' orchestrator, the Artifact Director constitutes a bridge between the internal native FAMiNE services, such as the BookLocalizer or the MigrationManager, and the ClassMATE-enabled applications. Any events fired by a native FAMiNE service are first wrapped into ClassMATE events and then distributed to the rest of the system.

The migration and remote synchronization processes are handled by the Artifact Director in cooperation with the State Manager and the DataSpace module. When notified by the local Window Manager that an application should migrate, the Director saves the current state of that application in a temporal network repository and notifies the remote Director about that location. The remote director copies that state locally and notifies the Window Manager to launch the indicated application and use the copied data to restore its state. A similar approach is followed during the synchronization process, however for that to be achieved every Director maintains a map that associates every migrated application with a remote node name to dispatch the appropriate commands.

Regarding the Artifact Director's workload, it is engaged in heavy tasks either during initialization where communication channels with the rest core components should be established or when an event occurs and the director should handle it. The director's action to events can be marked as either proactive, when the Artifact Director tries to take some precaution measures (i.e., disable user interaction for the artifact when a test examination is about to begin), or reactive when the Artifact Director responds to a stimulating event (i.e., when the student initiates the migration process).

The Artifact Director is also responsible for handling any ClassMATE Commands sent to the ClassMATE core by the various applications or services. Thus, the set of handler operations that implement the necessary logic for each Command type are registered during the Director's initialization. The complex commands, like the MIME or Migrate commands, are delegated to internal command-specific sub-modules following a modular approach where

the logic is distributed into several concrete modules, facilitating scalability and code readability.

The Artifact Director though could "suggest" some more-interesting applications (i.e., pending exercises close to deadline, or applications with related content).

Finally, the Artifact Director through monitoring the overall artifact logs every student's action and facilitates the user profiling process. The student-related data are maintained by the Data Space component, and in particular the User Profile module, and used to facilitate the data mining process. The exploitation of these data will be described in more details in chapter five.

4.1.3.1 Mime Command Handler

The mime Command Handler is an internal subsystem of the Artifact Director that handles all the MIME commands by launching the suitable applications with the appropriate data. The invocation of the appropriate operation is achieved through a dispatch table that associates the known mime types with delegate methods to handle them. Every delegate method takes a single argument, the MimeCommandEventArgs (as received by the issuer application) which contains all the necessary information. A dispatch table is a table of pointers to functions or methods. The use of such a table is a common technique when implementing late binding in object-oriented programming.

The Mime Command Handler, being an internal part of the Artifact Director, has direct access to the various ClassMATE core components (i.e., Data Space, Security Manager, etc.) to accomplish its tasks. When a Mime command is received, the Mime Command handler based on the supplied mime type invokes the appropriate delegate to handle it. The delegate will firstly resolve the most suitable application, and then request relevant content from the DataSpace component based on the MIME command's supplied argument. When the mining process completes, the Mime Command Handler will notify the Launcher to schedule the initiation of the resolved application to display the discovered content described in the automatically generated resource file.

The application's selection is accomplished using a special purposed map, the Mime Map. It is important to note that the Mime Command Handler does not take any precautions regarding the application's ability to load and interpret the resource file, as it is taken for granted that an application that can handle a particular type is also aware of the file's format containing the data to be displayed.

4.1.4 Class Orchestrator

The Class Orchestrator is the head of the classroom; it can be compared to the CEO of the classroom as it controls every aspect of the system. The control is performed in a high level and the Artifact Directors are responsible to apply its directions. For the decisions to be made, the Class Orchestrator monitors the environment and reacts to events of common interest. Orchestrator's decisions can affect either an individual or a group of artifacts; for instance, if the Orchestrator realizes that an examination is about to start, an instruction should be distributed declaring that every assistive application should suspend during examination time. On the contrary, if the Orchestrator realizes that the teacher yielded the floor to a student, then an instruction should be sent directly to the artifact that hosts that particular student, declaring that this student is authorized to interact with the class board.

The environmental monitoring and the communication needs between the Class Orchestrator and the Artifact Directors is accomplished directly through the FAMiNE middleware. In addition to the environment monitoring, the Class Orchestrator utilizes the class timetable and the detailed course schedule to infer decisions regarding global actions (e.g., exercises delivery or submission, examination date and time), while the Artifact Directors feed the Orchestrator with data regarding individual students (e.g., ongoing assignment score).

A modular approach was used for the Class Orchestrator implementation; the Interface-based mechanism is used to resolve the available services that monitor the environment, while the Security Manager is a special-purposed module which the Orchestrator uses to delegate access-related requests for handling.

4.1.4.1 Security Manager

Every attempt to launch an application or migrate from the local to a remote node is authenticated by the ClassMATE's authentication and authorization module, the Security Manager. Nevertheless, the decisions regarding any access rights are made by the Class Orchestrator as it performs classroom administration and monitoring, however they are no longer disseminated by the Orchestrator as the Security Manager intercepts and handles any access-related requests.

The Security Manager utilizes both static data from the Course-Applications registry and the dynamic decisions made by the Class Orchestrator as part of the context monitoring (e.g., disable hints applications during examinations) respectively. To optimize that process the

Security Manager maintains a local cache of previously made decisions, thus not having to query Orchestrator all the time and simultaneously "listens" for events coming from the Orchestrator to alter these decisions. The Course-Applications Registry is a configuration file that associates a particular course with a list of applications related to it. That registry is used during school hours to ensure that the students will always interact with course-related applications without being occupied with other courses or even worse wasting time entertaining applications. During break or out-of-school-hours, the students are permitted to interact with any of the installed applications.

An LDAP [40] approach is used to accommodate multiple rights lists, as for every application and artifact two lists exist, the groups' and the users' list, containing the authorized user groups and individual users respectively. In addition to these detailed lists, a few wildcard flags are used to handle special cases. Special cases occur due to context-related triggers and affect the access to an application or even an artifact. For example, consider the following cases, where access should be disabled on the one hand to the Multimedia Application when solving an exercise to avoid distraction, and on the other hand to the entire artifact to draw student's attention to the board. In the first case the MultimediaAccess flag turns from "LIST" to "NONE", where LIST denotes that the appropriate list contains the authorized users and NONE disables access completely; in the second case, the DeskAccess turns to "FALSE" to completely disable interaction with the entire Desk artifact. These flags are prioritized during the decision making process to take advantage of compiler's short-circuit evaluation and optimize the overall performance.

4.1.5 Application Launcher

The Application Launcher is the core module that bridges ClassMATE with the PUPIL system [22], by instructing application opening (Fig. 17). The Launcher cannot actually launch an application as every ClassMATE-enabled application is hosted inside the PUPIL's environment, in particular the artifact's Window Manager, but it generates the appropriate commands (i.e., Update Application Command) that when handled by the Window Manager, will eventually result in application(s) launching.

An application can be launched either directly by the Artifact Director as a response to a native FAMiNE (context-oriented) event, or indirectly by its Mime Handler delegate when handling a Mime Command fired by an application. In both cases, for an application to be launched the ClassMATE should incorporate mechanisms to both resolve from the installed

applications the preferred one(s), either by name or by mime type association, and ensure that the essential security-related requirements are met.

Figure 16: MIME Command Handling Process

Regarding its implementation, three main reasons led to the selection of the Singleton [14] design pattern: (i) facilitate control monitoring by issuing every single launch command from the same object, (ii) ensure that all the commands will be issued in order, and (iii) simplify the Launcher's invocation process by either Artifact Director or the Mime Command Handler.

4.1.5.1 Application Registry

To simplify the installation process of new applications and ensure the system's scalability, an external application configuration was introduced. For an application to be successfully installed, the respective entry must be present in the applications configuration. The application entry stores information regarding the application's name and icon, the loader class that constitutes the entry point to the constructor and the path to the binary file that contains the executable code (DLL library).

```
<ClassmateApplications>
        <Application>
            <Name> The Service's Interface full name </Name>
            <Loader> Loader's full name </Loader >
            <Icon> Path to the application's icon </Icon>
            <Lib> Path to the application's library (including the .dll extension) </Lib>
            <Mime> Mime Type </Mime>
            <RelatedCourses>
                <Course>CourseName1</Course>
                <Course>CourseName2</Course>
            </RelatedCourses>
        </ Application>
        <Application>
                More application definitions go here
        </Application>
</ClassmateApplications>
```

The configuration is loaded at start-up and the contained entries are converted into a map where every installed application is listed. The repetitive loading ensures that the map will

always be up-to-date. Whenever an application must be launched, this map provides the necessary parameters to the Application Command's issuer to generate a valid command, suitable for the Window Manager. In most cases the application registry is not used individually, but in combination with the Mimetype – Application Registry that will be described later.

4.1.5.2 Mimetype - Application Map

In addition to the installed applications registry, an associative dictionary exists to map the system's available mime types with the applications able to handle them. Apparently, every mapped application must belong to the currently installed applications.

Two alternative approaches exist regarding the storage of the mime types: either in the Applications' Configuration file or in an external configuration file. If mime types are stored in every application's configuration, a conflict might arise at runtime as more than one application could be able to handle the same mime type. On the contrary, if they are stored in an external configuration file, the conflict issue would be resolved; however, support for multiple applications handling the same mime type would be eliminated as well. A hybrid solution was the most preferable, as it combines the best of both worlds. Every application declares at installation time the mime types it can handle, and the Mime Map would store only the "preferred" one (as most modern Operating Systems do). Whenever a conflict occurs, context-aware reasoning is applied to select the application that can both handle that particular mime type and belong to the current course-related application list as well. If no match is found, then the user is prompted to select the preferred application from a list of available choices. Eventually, every pair entry associates a Mime type with a particular application. The Mime Map is generated during start-up and facilitates dynamic mime configuration by reflecting any changes made before use. At runtime the Mime Map is used by the Mime Command Handler to resolve and schedule the launch of the application that will display the discovered data.

```
<MimeMap>
    <MimePair>
        <MimeTypeName> Mime Type </MimeTypeName>
        <ApplicationName>
            Application's Name (as defined in ClassmateApplications)
        </ ApplicationName >
    </MimePair>
    <MimePair>
            More pair definitions go here
    </MimePair>
```

```
</MimeMap>
```

4.1.6 State Serialization

In computer programming, an application's state is essentially a snapshot of the measure of various conditions in the system. In the ClassMATE system, the concept of program's state is the building block used to realize fundamental ambient services: (i) suspend the application's state based on context-related conditions (e.g., suspend the Dictionary Application when taking an essay exam) and restore it later, (ii) store a temporal snapshot of the application's state and use it as a bookmark to jump back to that state (e.g., pin an interesting image displayed in the Multimedia Application to the Clipboard [22] to simplify return) and (iii) empower migration to another context by saving the application's current state and restore it at a remote node.

For that to be achieved in all three cases, the application's state should be stored in an efficient and easily programmable manner. In computer science, in the context of data storage and transmission, serialization is the process of encoding objects and the objects reachable from them, while protecting their private and transient data, into a stream of bytes so that it can be stored in a file or memory, supporting the complementary reconstruction of the object graph from that stream. Serialization among others provides a more convenient method of persisting objects than writing their properties to a text file on disk, and re-assembling them by reading this back in [30]. The only precondition is that if a serialized object contains internally other objects, then that object should also be serializable. All the mainstream programming languages offer a built in serialization mechanism in the form of an API, though any class may implement its own external encoding format and become solely responsible for its proper serialization and deserialization. ClassMATE is based on such approach, and in order to simplify the integration process offers its own proprietary Serialization Interface built on top of the native .NET Serialization Interface, which every ClassMATE-enabled application should implement to save/restore its state in a uniform manner.

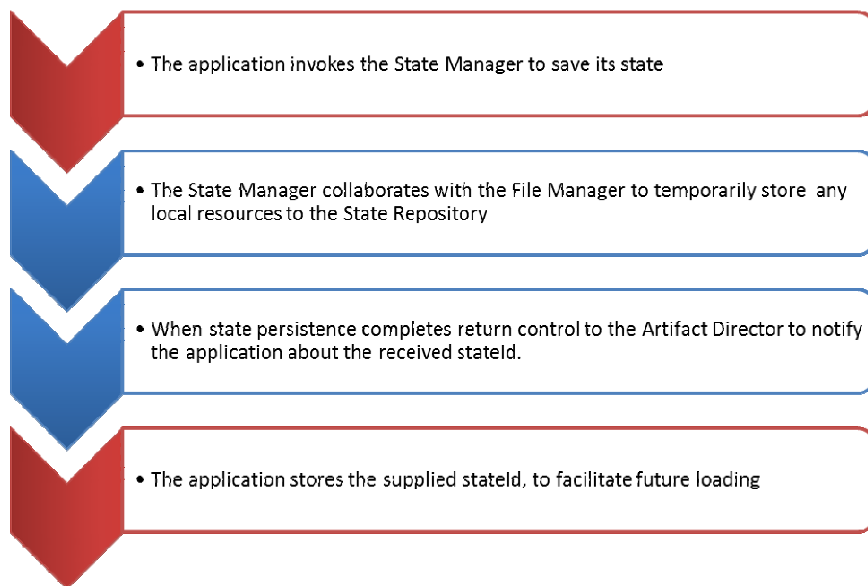


Figure 17: Application's State Serialization Process

Considering that every state object is application-dependent, an abstract approach was adopted during implementation to ensure universal application. The Serialization API defines two simple methods: (i) the SaveState and (ii) the LoadState, both implemented using Generics. The SaveState method simply serializes the supplied arguments regardless of their types, while the LoadState restores the application's state using the supplied arguments. In the latter case, the supplied types should match the desired ones for the restoration process to be successful (e.g., if application A attempts to restore its state using data intended for application B, then the process would result either in a corrupted memory stack with invalid data or in a runtime memory exception).

4.1.6.1 Resource Format Pair

An application may use multiple sources to get data from; in the ClassMATE system though, these sources correspond to local data files. These files can be encoded either in a proprietary application-specific format, as they were created during installation to hold configuration properties or during runtime to store application related data (e.g., history, user preferences, etc.), or in one of the predefined system formats as they were automatically generated at runtime by ClassMATE components (e.g., DataSpace). During the serialization process the first are handled as simple binary files while the latter require special treatment. This information regarding file variations are preserved in the persistent layer thought the ResourceFormatPair class which associates an existing file with a resource reference format (described in chapter five).

Serialization is a bidirectional process towards and from the file system. During serialization phase every data file, application-specific or not, is treated as a simple binary file and is stored in the appropriate location. During the deserialization phase though, the application-generated files are treated differently from the system-generated ones. In the first case, the system being unaware of the proprietary format cannot customize the data contained in those files, thus it simply loads them from the persistent layer to the main memory; in the latter case, a format-specific process is invoked to revalidate the contained data. As an example, consider the case where the Multimedia Application migrates from an AmIDesk to the AmIBoard; the system-generated file that describes the images to be displayed include desk-specific file paths, thus the Multimedia-specific delegate must replace them with valid board-specific paths.

4.1.6.2 State Class

The state of each application is determined by a number of attributes maintained by that particular application. In such cases, serialization is achieved by storing the state-related data in an auxiliary class and then save that class the file system; when loaded back the extracted data are used for state's restoration. ClassMATE's Serialization mechanism extends the aforementioned process in order to operate independently of any internal structures by introducing an abstract utility class, the State class, that package the state-related data. The State class implements the .NET Serialization interface to be (de)serializable and maintains the following attributes: (i) the name of the application whose state is encoded in the current instance, (ii) the AppData, a parameterized type object to store the application's auxiliary class, and (iii) a collection of ResourceFormatPairs where the external data files used by the application are described. For the state's (de)serialization to be successful, all the contained objects should also implement the .NET Serialization interface. The ClassMATE Serialization mechanism does not modify at any point the state object as the application is solely responsible for its valid population; ClassMATE only cares about its intact transfer to the storage repository.

The State object is instantiated by the application when the SaveState method is invoked, populated with the appropriate data and forwarded to the StateManager to associate it with a unique identifier and write it to the file system. Likewise, when the LoadState is invoked, a state object is supplied as the argument encoding the state data and the application uses it to restore its state; to facilitate that process, the State class provides a parameterized method that returns the AppData object casted in the application's specific type.

4.1.6.3 State Manager

The State Manager contains the low-level routines that communicate with the file system during the serialization and deserialization process respectively. To ensure that a single instance will handle any serialization-related request the Singleton [14] design pattern was used for its implementation.

For Serialization to be achieved, the State Manager uses both the built-in .NET Serialization components and the ClassMATE's File Manager. The .NET components (i.e., FileStream and BinaryFormatters [25]) expose operations to read and write binary files, while the File Manager provides the file descriptors by transparently querying the classroom's data repository. The State Manager is a stateless component as instead of maintaining any data in its memory, it stores every state object in a binary format in the repository. In the repository a particular segment is reserved to host the state objects, and a string-based protocol is used as the indexing scheme. Every serialized state object generates its own folder and is identified by and unique id that includes: (i) the application's name, (ii) the artifact's identifier, and (iii) a timestamp of the creation date. The protocol not only ensures that the same application can serialize (and deserialize) its' state multiple times, but also eliminates overlaps between states created from the same application on different artifacts.

The StateManager public exposes the SaveState and the LoadState that encapsulate the low-level procedure calls to write and load data from the file system. The SaveState (Fig. 18) takes as input arguments the state object that must be serialized and a flag that indicates the Serialization type (Suspension, Pin, Migration) and returns as an output result the unique StateId to be used during deserialization (similar to a ticket given at the cloakroom when checking a coat). If Serialization type is either Suspension or Migration, then the state's ResourceFormatPair collection is iterated to locate and copy the external files in the state's folder in the repository. The LoadState receives the StateId as a single input argument and upon completion returns the retrieved state object. Its workflow can be decomposed in two phases: the deserialization and data validation phase. During the deserialization phase, the StateId points the binary data to be deserialized as a state object in the main memory, while the second phase ensures that the data contained in external files generated by system components will be valid. During the validation phase, the state's ResourceFormatPair collection is iterated, and for each file whose associated format type belongs to the system-generated types, the appropriate delegate is invoked to revalidate its contents. The

revalidation process will be described in more details in the “Resource Description Format” section; in a nutshell it includes from simple string replacements to even binary files copy.

4.1.7 Initialization Process

The ClassMATE’s modular architecture assumes that during system’s initialization, a series of actions will be taken to load and prepare the essential modules before use. Considering that the ClassMATE systems targets systems of diverse scales (e.g., from a single notebook to a large scale distributed AmI environment), the Platform Expert module should be launched first to identify the current platform, resolve the respective services, and then launch the Artifact Director to undertake artifact’s control. ClassMATE’s initialization process resembles the one followed by every computer, where the system’s Bios (Platform Expert) is initially launched to setup the environment, and then the OS kernel (Artifact Director) takes over control.

The initialization process (Fig. 19) of the Artifact Director module consists of three phases: (i) events installation, (ii) local services resolution, and (iii) environment’s notification. During events installation, the Artifact Director immediately initiates the ClassMATE’s event system and registers the Command handler delegates to start listening for events addressing the ClassMATE core, including events from other core component (e.g., DataSpace, DeviceManager, etc.) and the Class Orchestrator. During the second phase, the Artifact Director, based on the current platform’s configuration, resolves the artifact-specific services that provide context-related data (e.g., book localization, student’s presence, etc). Finally, during the last phase, the Artifact Director notifies the Class Orchestrator, and the ambient environment in general, that the current artifact was successfully initialized and is henceforth fully functional.

Upon successful initialization, the control is passed back to the caller, to continue in ClassMATE’s case to the Window Manager, to continue with its normal workflow.

Figure 18: Platform initialization process

4.1.8 Migration Process

Application's migration is the result of collaborative work between both local and remote components, and can be decomposed into the foundation and the realization phase (Fig. 20). During the foundation phase, the local Window Manager identifies the appropriate user action on the desk and initiates the migration process by requesting from the application on the foreground to save its state through the State Manager. When save is completed, the application returns the unique StateId received by the State Manager to the Local Window Manager, who creates a Migrate Command with the application's name, the StateId and the remote context and forwards it to the Artifact Director. When received by the Artifact Director invokes a special-purposed internal module, the Migration Command Handler, to extract the necessary information from the received Migration Command and generate an Application Command with a "STATE--StateId" argument, denoting that the remote application will have restore its state. The foundation completes by the transmission of the generated command to the remote node.

The realization phase starts when the remote Artifact Director receives that command, and engages by modules running on the remote node only. The Artifact Director after successfully validating that the command was received intentionally, the contained data are not corrupted and the Security Manager approves migration, requests from the State Manager to transfer locally and revalidate the data pointed by the StateId (the exact process was described in detail in the State Manager section).

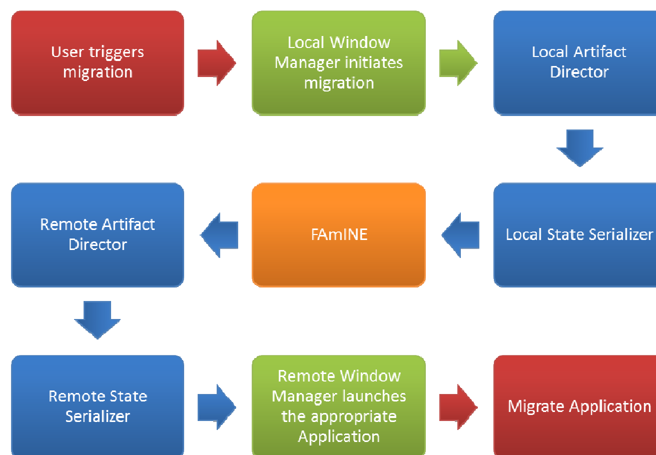


Figure 19: Application Migration Process

Finally, the Artifact Director notifies the Window Manager to launch the appropriate application and forward the Application Command that enforces the application to restore its saved state and successfully complete migration.

4.2 Device Manager

The Device Manager offers a generic mechanism for the manipulation of heterogeneous devices, by any ClassMATE-enabled application. For that to be achieved, the same Interface-based approach used by the Platform Expert is adopted as well, where every device exposes its functionality as a service API, completely dissociated from the hardware layer; an artifact that lacks a particular device can emulate its functionality through software by implementing the appropriate interface. Therefore, the Device Manager is the sole extension point where new devices can be added, whereas any ClassMATE-enabled applications request the appropriate service API from the Platform Expert.

Every artifact accommodates a Device Manager instance which handles the input / output devices and supports their interaction with any application in the ClassMATE cloud. During initialization, the Device Manager uses the Service Factory pattern to resolve the services of the current platform (the factory instance is responsible for instantiating the appropriate objects); hence, both remote and local devices are transparently supported by the system, as the interaction is orchestrated by the Class Orchestrator and the communication needs are handled by the ClassMATE's Events Layer.

4.2.1 Towards a universal Multitouch solution

The latest trends in human-computer interaction indicate a turn towards multitouch interaction schemes, especially after the launch of Apple's Iphone and other several multitouch-capable tablets and screens. Moreover, the computer's vision domain contributes towards the same track, supporting to reproduce multitouch interaction through vision. The great variation between the protocols used by hardware vendors with those used by software-based solutions prevents the establishment of a commonly acceptable, yet scalable Multitouch API. The latter has changed with the advent of Microsoft Windows 7 and the Windows Touch technology [29] (Fig. 21). Multitouch functionality has been incorporated as an integral part in the operating system's core and full support was added to the application development tools [27]; hence applications can take full advantage of the native multitouch support by using native APIs. ClassMATE introduces an extensible mechanism where any hardware- or software- based multitouch system can be transparently installed, with no modifications either to the application or to the Windowing System.

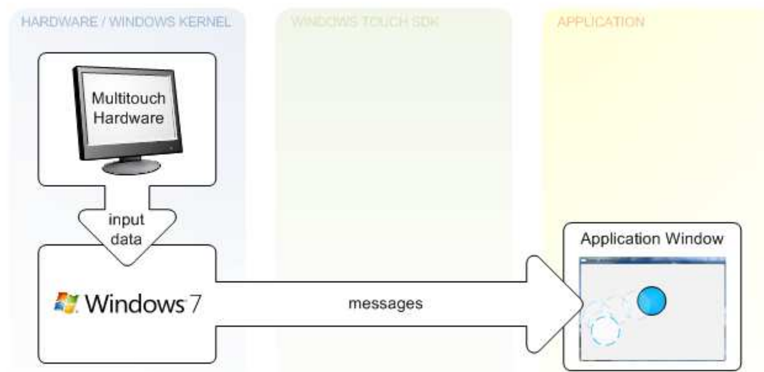


Figure 20: Windows 7 sends messages from multitouch hardware to an application

The introduction of multitouch interaction established new interaction schemes like the Multitouch Manipulation. Manipulations can be considered as a superset of gestures. The difference between manipulations and gestures is best demonstrated through a simple example. The user can expand an object and at the same time move it using manipulations; with gestures, only one at a time can be performed. This ability to manipulate an object in real time makes applications more intuitive to users by enabling a more realistic experience. The Manipulation APIs are used to simplify transformation operations on objects for touch-enabled applications. Manipulations are performed in Windows 7 through the manipulations COM object [28]; without that built-in mechanism, every developer should keep track of active touch points, calculate numerous metrics and, finally, manually apply the appropriate transformations.

Manipulations are transparently calculated by each WPF components private manipulation processor using Windows Touch Messages generated by the driver of the touch-capable device, as depicted in the following Fig. 22.

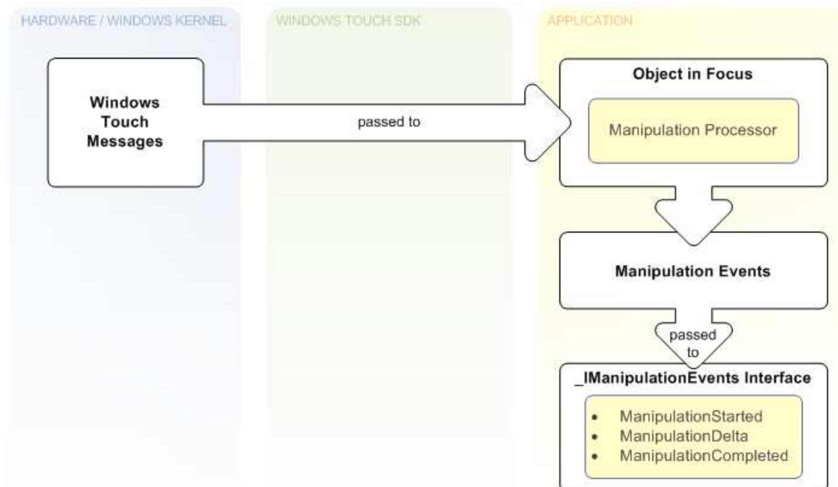


Figure 21: Windows 7 Manipulation Overview

In ClassMATE though, the built-in processors cannot directly translate and use ClassMATE's Touch events to calculate manipulations. To address that, the ManipulationEnabled API was introduced to avoid low level driver programming; every application that is manipulation-enabled should implement it and provide an operation that returns its internal manipulation processor to the ClassMATE system to inject custom code that can recognize and handle the custom events. In addition to the ManipulationEnabled API, a number of supplementary classes, required by the Windows Touch mechanism, were introduced to provide the necessary functionality.

- The VisionBasedTouchDevice emulates in software a physical custom touch device and encapsulates touch-related information: (i) the position of the touch, (ii) the exact time at which the event occurred, and (iii) a flag denoting its type, (i.e., TouchDown when a finger touches the screen for the first time, TouchMove when a finder is moved over the screen without losing contact, and TouchUp when a finger is drawn away from the screen).
- The TouchInputInterface is the publicly exposed interface that provides Touch functionality. Its' key feature is the provided event hook, where applications can register their own delegates to be invoked when a touch event occurs; the same technique is following for native WPF touch events as well. The event mechanism used is the one built-in .NET framework, so as to optimize performance and offer great user experience.

- Finally, the TouchInputHandler is the Touch system core, as it communicates directly with the vision system. Its main objective is to translate vision events to Touch events, which when received by the WPF framework generate the same effect as if they were generated by a physical touch device. The handling process includes: (i) the translation of the coordinate system, as the one used by vision differs from that used by the screen, and (ii) the determination of the event type (up, down move). For that to be achieved, the handler maintains a dictionary of the currently active contacts, and upon change, identifies the newly added contacts, the contacts that moved, and those that do not exist anymore and generates the appropriate touch events.

4.2.2 Book Localizer

The Book Localizer is a local module that resides on every artifact and is charged with identifying the current book page that the ClassBook Reader Application should display (Fig. 23). In the technologically-augmented desk [3], the front-facing camera is utilized to identify the currently open page and notify the Artifact Director to launch, if necessary, and update the ClassBook Reader appropriately. For the remaining artifacts, that module is emulated through software as they lack the necessary hardware. In that case the exact page is selected by combining data from the class timetable retrieved by the Class Orchestrator, the obligations of the course (i.e., ongoing tasks and assignment deadlines) also from the Class Orchestrator and finally current student's profile (already prepared assignments, pending tasks) by the User Profile.

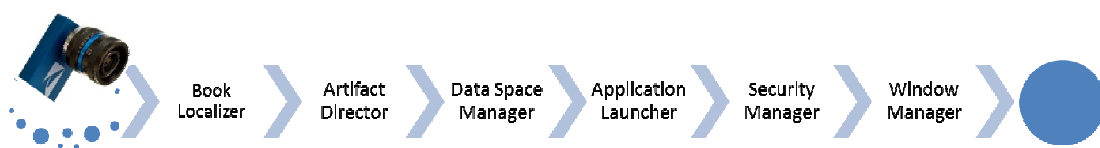


Figure 22: Physical Course book Localization Process

5 Content Personalization

One of the key features of the ClassMATE system is the delivery of personalized education content based on the current needs of the individual learner. For that to be achieved various modules collaborate. On the one hand, the User Profile provides the user-related parameters for the content personalization process. In addition to the "common" static personal data (e.g., name, surname, grade, scores, etc.), dynamic data are collected at runtime though interaction monitoring and encoded into behavioral models that facilitate the adaptation of the filtering process. On the other hand, the educational content is enhanced with metadata that convey information about its educational attributes and the taxonomies under which is classified, whilst a sophisticated content discovery mechanism utilizes the available metadata entries to semantically identify educational content suitable for the current context of use (e.g., course) and the current student. Finally, the content personalization mechanism is built in a modular way to facilitate: (i) content addition, (ii) introduction of new classifications schemes or modification of existing ones, and (iii) query adjustments.

5.1 User Profile

The User profile is a collection of personal data associated to a specific user; therefore a profile refers to the explicit digital representation of a person's identity and characteristics. The information contained in the profile can be exploited by systems taking into account the persons' characteristics and preferences, for instance by adaptive hypermedia systems, to personalize the human computer interaction. In ClassMATE the user profile is not a passive structure, as in various computer applications where it simply identifies the valid users of the system, but is rather an active component that evolves through time, and bridges the ambient environment with the Data management layer. The User Profile's main objectives resemble those of the IMS Learner Information Package [19], where the data model that describes the characteristics of a learner can be used for:

- Recording and managing learning-related history, goals, and accomplishments
- Engaging a learner in a learning experience
- Discovering learning opportunities for learners

In the ClassMATE system two discrete profiles exist: (i) the teacher's and (ii) the student's profile. The teacher's profile is not yet fully exploited for educational purposed and mainly acts solely as a passive personal data repository. The student's profile on the other hand, is "fully" utilized by the system both to personalize the content discovery process to the

particular educational needs of each individual student, and log commonly used interaction patterns in order to define a student's behavioral profile. The behavioral characteristics and the analysis process will be described in more details in the next section. The student's profile is divided into four segments: (i) personal data, (ii) student record, (iii) user preferences, and (iv) behavioral attributes.

- The personal data, as implied by their name, include personal information such as full name, date of birth, e-mail address, home address, etc.
- The student record includes detailed grades (oral and written examinations, projects, etc.) and activity list (pending exercises, scheduled examinations, etc.) for the ongoing courses, and a complete history of the past years' records.
- In the user preferences section, the user's customization options [22] are stored (e.g., desktop background, windows skins, color themes, etc.)
- Finally, the behavioral attributes section accommodates the knowledge resources library of students' behavior patterns, dynamically gathered via their activity monitoring.

To better understand the behavioral attributes used by the ClassMATE system, consider that the available educational content is structured under thematic areas (e.g., mathematics, physics, linguistics, etc.) and every single educational exercise is marked with a type (e.g., multiple-choice exercise, fill-in-the-gap exercise, free-text exercise, etc.) and a difficulty tag (i.e., easy, normal, hard).

The behavioral attributes are categorized into course-specific and general. The first category refers to metrics about a student's attitude towards course-specific activity, while the latter includes accumulated metrics regarding all the student activities. The complete list of those attributes can be found below:

General Behavioral Attributes:

- ratio of successfully answered exercises, per exercise type, per difficulty level, and correlated, e.g., 75% correct answers on multiple choice exercises, 80% on easy/medium questions, and 20% on hard
- amount of hints asked per exercise type, per difficulty level, and correlated, e.g., 15 hints asked on hard exercises; 12 on free-text exercises and the remaining 3 on medium
- favorite and disliked difficulty level(s), e.g., {favorite: medium, hard}, {disliked: easy}

- favorite and disliked exercise type(s), e.g., {favorite: multiple choice, image-to-image matching}, {disliked: free-text}
- favorite and disliked course(s), e.g., {favorite: mathematics, physics}, {disliked: linguistics}

Course-specific Behavioral Attributes (for instance in the context of mathematics):

- favorite exercise type(s), e.g., multiple-choice regarding theorems
- favorite difficulty level(s), e.g., medium or hard
- favorite and disliked topics, e.g., differential equations, trigonometry

Activity related metrics not only assist the learners by providing personalized content focused on their weak thematic areas, but also improve the educational process by reporting class's activity to the teacher; hence, the teacher is able to modify the course schedule, dictate exercises that the students prefer and augment course syllabus with assistive material in order to bridge any knowledge gaps.

The student profile data depending on their modification rate are categorized into those that never change (static), those that gradually change (semi-dynamic), and those that continuously change driven by user interaction (fully-dynamic). Static data are manually defined once and never change (e.g., full name, date of birth, etc.), semi-dynamic data are automatically generated by the ClassMATE core according to environmental triggers (e.g., the announcement of course's upcoming schedule or final examination's grades). Finally, regarding the fully-dynamic attributes (e.g., preferred exercise type, preferred difficulty level, etc.), standard learning styles are used for their initialization (only for the freshman students), while the Activity Monitor module dynamically updates them at-runtime through monitoring user interaction. These data gathered by the User Profile service, through an iterative monitoring and evaluation process, constitutes the main feedback for the Context Manager, so that a learner's centric rational is applied for content delivery and interaction control, thus providing adaptation to individual student's needs. For efficient monitoring to be achieved, both the ClassMATE-enabled applications and the ClassMATE core are obliged to notify the Activity Monitor of interesting events (e.g., the student discarded an exercise, the student successfully solved a mathematic problem, the student asked for additional information regarding some topic, etc.). When notified, the Activity Monitor correlates the data from the received event with contextual information (e.g., current course/topic based

on the timetable, current exercise difficulty level, etc.) and updates the student profile respectively.

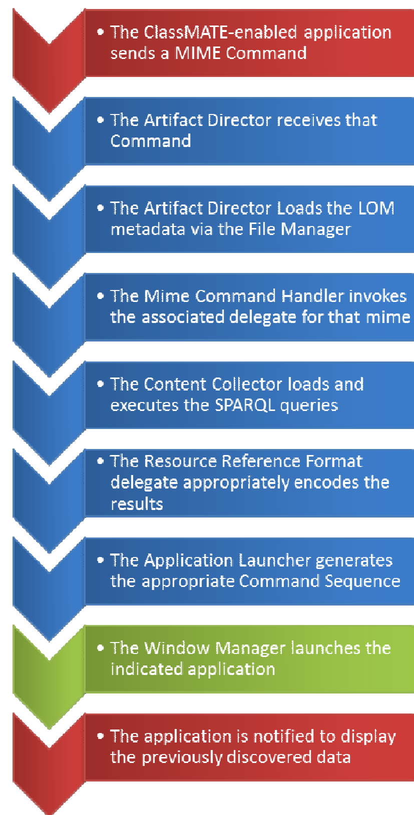


Figure 23: Automatic Content Discovery process

The following scenario illustrates the overall process (Fig. 24). In the course of Mathematics, a student has to solve three exercises, namely one exercise and two problems of varying difficulty regarding the Pythagorean Theorem. The system randomly proposes the hard problem first, but the student is not confident enough for its solution and decides to skip it. The theory exercise is proposed secondly, but the student postpones it for a later session. Finally, the simple problem is proposed and the student solves it correctly. The pending hard problem is once again proposed and the student solves it correctly using two of the available hints. The above scenario yields a pattern, in the context of Mathematics, where the user prefers problems over theory exercises, and in particular simple over hard problems. The system is now aware of that pattern and in the future, in the context of Mathematics and for that particular user, it will firstly propose the problems and then the theory exercises; and if the user asks for supplementary content the system will prioritize simple instead of hard problems. However, the system will periodically try to advance the difficulty level and monitor and re-evaluate the student's learning behavior pattern once again. Alternative patterns based on the aforementioned scenario that could alter the delivery order could be:

either a student who prefers practicing in theory exercises and then move to problems, or a student who prefers solving hard challenging problems.

5.2 DataSpace

The DataSpace provides an abstraction layer between the applications and the physical storage layer. This added layer not only encapsulates the implementation details, but also makes available the following key facilities: (i) a single reference point to content repositories providing transparent content access and management, (ii) a content classification mechanism providing the necessary content-related rationale to data mining procedures, and (iii) a sophisticated filtering mechanism for personalized content delivery based on user needs and preferences.

The Data Space strongly collaborates both with the User Profile and the ClassMATE core to collect the essential static or dynamic user- and context- characteristics to enhance the decision process. Therefore, the ClassMATE-enabled applications are transformed from “fat” clients who independently provide rich functionality, to “thin” clients with limited functionality concentrated solely on providing a graphical user interface, as the ClassMATE core and the DataSpace deals with the content remaining functionality.

5.2.1 Related technologies overview

The DataSpace implementation has adopted numerous well-established data management standards and mining techniques particularly suited for e-learning platforms; a brief overview of these technologies is provided in the following section.

5.2.1.1 Learning Object Metadata (LOM)

The Learning Object Metadata (LOM) [18] specifies a conceptual data schema that defines the structure and specifies the data elements of a metadata instance for a learning object. A learning object is defined as any entity -digital or non-digital- that may be used for learning, education or training. A Metadata instance for a learning object describes relevant characteristics of the learning object to which it applies. Such characteristics may be grouped in general, life cycle, meta-metadata, educational, technical, educational, rights, relation, annotation, and classification categories.

LOM is intended to be referenced by other standards that define the implementation descriptions of the data schema, so that a metadata instance for a learning object can be used by a learning technology system to manage, locate, evaluate or exchange learning objects, while it does not define how a learning technology system represents or uses a

metadata instance for a learning object. Its purpose is to facilitate search, evaluation, acquisition, and use of learning objects, for instance by learners or instructors or automated software processes. This multi-part standard also facilitates the sharing and exchange of learning objects, by enabling the development of catalogs and inventories while taking into account the diversity of cultural and linguistic contexts in which the learning objects and their metadata are reused.

5.2.1.2 Sparql

RDF is a directed, labeled graph data format for representing information in the Web. RDF is often used to represent, among other things, personal information, social networks, metadata about digital artifacts, as well as to provide a means of integration over disparate sources of information. The SPARQL [24] query language for RDF was designed to include triple patterns, conjunctions, disjunctions, and optional patterns in queries, and return an XML document format for representing their results.

Most forms of SPARQL query contain a set of triple patterns called a basic graph pattern. Triple patterns are like RDF triples [38], except that each of the subject, predicate and object may be a variable. A basic graph pattern matches a subgraph of the RDF data when RDF terms from that subgraph may be substituted for the variables and the result is RDF graph equivalent to the subgraph.

The example below shows a simple SPARQL query to find the title of a book from the given data graph. The query consists of two parts: the SELECT clause identifies the variables to appear in the query results, and the WHERE clause provides the basic graph pattern to match against the data graph. The basic graph pattern in this example consists of a single triple pattern with a single variable (?title) in the object position.

Data	<http://csd.uoc.gr/books/book1> <http://purl.org/dc/elements/1.1/title> "SPARQL Tutorial" .

Query	SELECT ?title
	WHERE
	{
	< http://csd.uoc.gr/books/book1> <http://purl.org/dc/elements/1.1/title> ?title .
	}

Result	"SPARQL Tutorial"

5.2.1.3 SemWeb

SemWeb.NET [36] is a Semantic Web/RDF library written in C# for Mono or Microsoft's .NET. The library can be used for reading and writing RDF (XML, N3), keeping RDF in persistent

storage (memory, MySQL, etc.), querying persistent storage via simple graph matching and SPARQL, and making SPARQL queries to remote endpoints. Limited RDFS and general-purpose inferencing is also possible. The SemWeb's API is straightforward and flexible. The library has no particular tools for OWL schemas. It operates at the level of RDF triples only.

The library's facilities used in ClassMATE are listed below:

- RDF/XML: Reading and writing RDF/XML (including XMP). The reader is streaming, which means the entire document doesn't ever need to be loaded into memory.
- Notation 3: Reading and writing NTriples, Turtle, and most of Notation 3
- SQL DB-backed persistent storage for MySQL, combined with the extended Select operation to query many things at once (much faster than making individual calls to the underlying database)
- The available in-memory store
- RDFS Reasoning and rule-based reasoning based on the backward-chaining Euler engine, over any data.

5.2.2 Metadata

Metadata is loosely defined as data that describe other data. Metadata is a concept that applies mainly to electronically archived or presented data, and is used to provide a substantial amount of information about those elements (e.g., definition, structure, administrative directives, etc.). Metadata is structured according to a standardized concept using a well-defined metadata scheme, and the contained information could refer to:

- means of creation of the data
- purpose of the data,
- time and date of creation,
- creator or author of data,
- placement on a network (electronic form) where the data was created,
- etc.

For instance, a digital image may include metadata that describes the camera settings, how large the picture is, the color depth, the image resolution, when the image was created, and other data. A text document's metadata may contain information about the size of document is, the author, the date when the document was written, and a short summary of the document.

Among others, metadata can be used during content discovery to associate their data elements. The term metadata discovery refers to a process where automated tools discover the semantics of a data element in data sets and produce a set of mappings between the data source elements and a centralized metadata registry. Based on the matching algorithm used, the discovery process can be categorized as lexical (exact, synonym pattern), semantic and statistical matching [41].

In the context of ClassMATE, where automatic content discovery is a vital task, the employment of metadata could significantly improve results accuracy. In particular, the LOM scheme was selected to define the metadata structure, as the majority of its contained data (general, educational, relation and classification sections) fit the ClassMATE needs and requirements. thus making LOM an ideal choice. As a result, the mining and classification processes heavily engaged metadata-related logic in their implementation.

Metadata is data. As such, metadata can be stored and managed in a registry or a repository. LOM however does not provide a standardized solution concerning metadata storage. Subsequently, the storage / retrieval mechanism was implemented from scratch following the specification word by work without any derivations. The XML language was preferred as the implementation technology over other binary-based solutions, because it not only facilitates readability and modifiability through a simple text editor, but also ensures portability as every learning object can always be accompanied by its metadata. The metadata population will be described in more details late on; in short it is a semi-dynamic process where the system initializes a metadata entry during learning object's admittance using contextual information which the user can later refine and augment.

The LOM implementation in ClassMATE will be described in more details in the next section.

5.2.2.1 LOM Types

The LOM specification defines a set of custom data structures (Fig. 25) used throughout the hierarchy to ease implementation and facilitate maintenance. These structures include primitive datatypes, containers and complex data types which either extend or combine containers and primitives formulating composite structures. The entire collection of the LOM data structures, their attributes and relations is depicted in the figure:

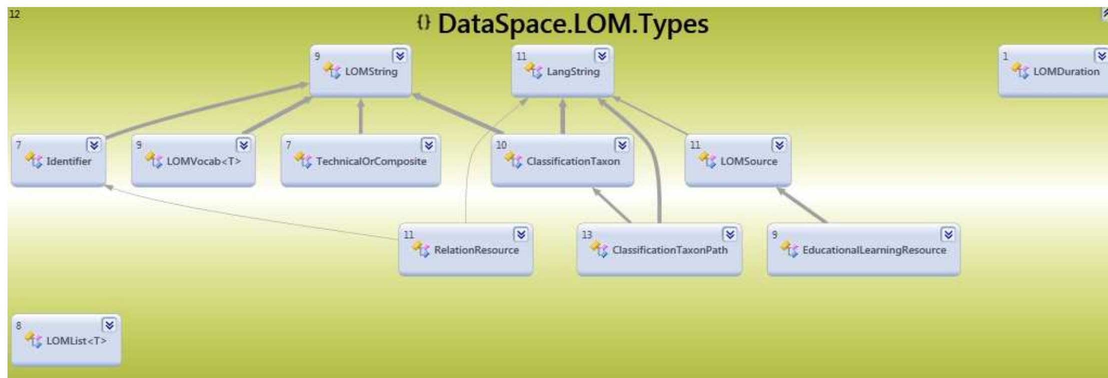


Figure 24: The LOM Datatype

The key functionality of every LOM structures is not limited to its contained attributes, but is encapsulated in the operations that implement the IEquatable and the XMLSerializable interfaces. The IEquatable interface is used for comparison purposes and every LOM element should implement its own algorithm; that algorithm should not only refer to the natural comparison method but implement a more sophisticated method where the natural ordering is combined with the stored values to determine the result. The XMLSerializable interface on the other hand, defines the appropriate methods that facilitate storage and retrieval from an XML file. During LOM storage, every LOM element is dictated to provide a string representation of its internal structure to be persisted in an XML file, while during loading, given a valid XML element, every LOM element should populate its contents with the supplied values.

5.2.2.2 LOM Metadata Structure & LOM Entry

The ClassMATE utilizes only a subset of the LOM specification (Fig. 26) during the mining and classification processes. This subset contains the following LOM sections: (i) general, (ii) technical, (iii) educational, (iv) relation, and (v) classification; every section defines a new class type composed by various LOM types and implements the IEquatable and XMLSerializable interfaces.

The general section groups the general information that describes a learning object as a whole, and is mainly used to identify the associated a learning object when necessary. The technical section describes the technical requirements and characteristics of a learning object, and is used during the mining process to filter the related content based on the MIME type. The educational section describes the key educational or pedagogic characteristics of a learning object, and its main objective is to personalize the content to fit the learner's needs before its delivery. The relation section describes the relationship between a learning object and other learning objects, if any, and is used during the mining

process to efficiently resolve other leaning objects already related to the current. Finally, the classification section describes where a learning object falls within a particular classification system, and is employed during mining and re-classification to resolve other learning objects that belong to the same taxonomy (e.g., siblings), and either return them or associate them with the current learning objects (by appropriately modifying their relation section).

The LOM Entry is a single aggregator that collects together these individual objects in a single class and exposes the appropriate operations to access them. In addition to these accessors methods, the LOM Entry also implements the operations defined by the IEquatable and XMLSerializable interfaces, but their implementation is straightforward as every request is delegated to the contained objects for execution.

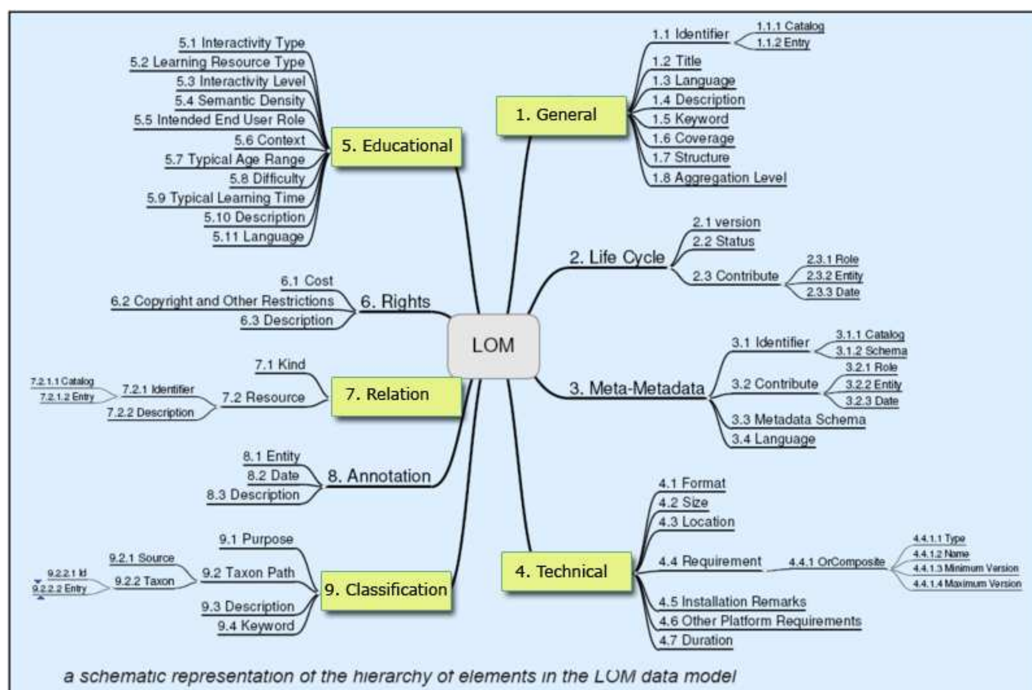


Figure 25: Learning Object Metadata (LOM) Specification

5.2.3 Resource Reference Format

The DataSpace module encapsulates a content classification and a sophisticated filtering mechanism for personalized content delivery. For the mining process to be successful, every data object available in the ClassMATE's repository should be classified and accompanied by the relevant metadata. However, the content delivery does not automatically ensure that the content is displayed properly by the application. An XML-based file, the resource reference file, is generated to hold references to the actual data resources discovered by the Data Collector.

The structure of that file varies according to the type of the discovered content. The DataSpace has defined a basic, yet extensible, collection of structures, namely Multimedia, HotSpots and Hints, that are sufficient for the needs of the currently handled content types. The Data Collector according to the type of the discovered content uses the respective delegate to transform the results of the mining process into the appropriate structure, and then store them into a resource reference file. Any ClassMATE-enabled application that wishes to present those data is solely responsible for parsing the resource reference file and loading the designated content. To optimize performance, the Data Collector, prior to generating the resource reference file, copies locally (in collaboration with the File Manager) any remote resources, hence the stored references point to artifact-specific locations. During migration though, these artifact-specific locations are no longer valid and should be replaced. The Data Collector's delegates are responsible for reloading the previously stored resource reference files and replacing the any invalid data.

The above observations led to the definition of the ResourceReference API, which defines two operations only: WriteToStream and RevalidateData. Every internal component that belongs to the DataSpace module and generates a resource reference file should implement that interface appropriately. The WriteToStream operation takes as input arguments the results of the mining process (as a sparql XML result set) and an output stream, and stores the results in the appropriate structure in the output stream. The RevalidateData operation takes as input arguments an inputstream that corresponds to a resource reference file (whose structure can be handled by the current delegate) and an output stream, replaces any invalid data contained in the input stream, and stores the updated version in the same structure in the output stream.

```
<ResourceFormats>
        <FormatPair>
            <MimeTypeName> Mime Type </MimeTypeName>
            <ResourceFormat> Format's Identifier </ResourceFormat>
            <ResourceFormatSchema> Format's Schema </ResourceFormatSchema>
        </FormatPair>
        <FormatPair>
                More pair definitions go here
        </FormatPair>
</ ResourceFormats >
```

5.2.3.1 HotSpot

The HotSpot format is exchanged among applications that display course's content as an image (e.g., the electronic version of a physical course book page) that contains interactive

spots which can trigger the launch of other applications. Such an example is the ClassBook Application that displays the electronic version of the currently open page of the physical book. The images and exercises displayed on any page are selectable, and when selected the relevant content discovery processed is triggered and the appropriate ClassMATE-enabled application is launched (e.g., the Multimedia Application is launched if an image is selected or the Multiple-Choice Exercise if an exercise is selected).

The structure of the resource reference file regarding Hotspots, as depicted below, contains the path to the image that should be displayed (as aforementioned the actual image file is copied locally to optimize loading time), and the list of hotspots available on that image. For every hotspot, the bounding points specify the area in which every user action should trigger a MIME command, and optionally designates the region that could be visually decorated (by the application) to attract the user's attention. In addition to the bounding points, the MIME type of the learning object contained in that area is defined (e.g., image/png) and the Command entry is populated with that learning object's URI (e.g. 6thGrade_EnglishCourseBook_Chapter3_Unit3_Lesson1_RollerCoasterImage). When the user triggers a mime event, the application instead of having to identify the selected object, simply packages the values of the MIME and the Command entries in a MIME Command and forwards it to the ClassMATE core for handling.

```
<HotSpotElements>
        <ImageSource> Value </ImageSource>
        <HotSpots>
          <HotSpotElement>
            <BoundingPoints>
              <Point>
                <X> Normalized X coordinate </X>
                <Y> Normalized Y coordinate </Y>
              </Point>
              <Point>
                More point definitions
              </Point>
            </BoundingPoints>
            <MIME> Mime type </MIME>
            <Command> Command (Learning object's URI) </Command>
          </HotSpotElement>
        </HotSpots>
        <HotSpots>
            More hotspots definitions
        </HotSpots>
</HotSpotElements>
```

The HotSpots module implements the ResourceReference API to facilitate its use by the DataSpace. The WriteToStream operation populates the various HotSpot entries by parsing the sparql results provided as input, whereas the RevalidateData operation loads the XML-based representation of a HotSpot structure, locates the image in the ClassMATE's repository using the value of the ImageSource tag, makes a local copy of it to the remote artifact and then replaces the value to point to the new location.

5.2.3.2 Multimedia

The Multimedia format is exchanged among applications that display multimedia content (e.g., images, videos and sound) such as the Multimedia Application. The structure of the Multimedia resource reference file contains three distinct sections,Images, Videos and Audio, whereas any multimedia reference file should contain at least one of these sections. Each section contains the list of learning objects that should be displayed, while for each object the path and URI are stored.

The Multimedia module implements the ResourceReference API. The WriteToStream operation generates a XML-based file with the appropriate format, while the RevalidateData operation for each path entry copies the data file from the ClassMATE repository to the remote artifact, and then updates the path value with the new artifact-specific location.

```
<Multimedia>
	<Images>
		<Image>
			<ImagePath> The path to the image file </ImagePath>
			<ImageURI> The unique identifier of this learning object </ImageURI>
		</Image>
		<Image>
			More image definitions
		</Image>
	</Images>
	<Videos>
		<Video>
			<VideoPath> The path to the video file </VideoPath>
			<VideoURI> The unique identifier of this learning object </VideoURI>
		</Video>
	</Videos>
	<Audio>
		<Audio>
			<AudioPath> The path to the audio file </AudioPath>
			<AudioURI> The unique identifier of this learning object </AudioURI>
		</Audio>
	</Audio>
</Multimedia>
```

5.2.3.3 Hint

The Hint format is used by the application that take advantage of ClassMATE's mining and personalization features to assist the student when solving an exercise. Hints are presented gradually while their content is adapted to fit the needs of each individual learner, for instance augment textual information with an explanatory image for visual learners. Every Hint resource reference file is structured in three sections that correspond to their displaying order. The first section provides the definition of the selected item (e.g., the black in a multiple choice exercise) in a textual representation accompanied by a collection of multimedia files (image, video and audio file) that describe the same definition in a multimodal manner. The second section provides a personalized collection of examples of use that include the available options (e.g., multiple-choice alternatives, available words for selection in a matching exercise). Finally, the third section contains a list of incorrect options that should be eliminated, leaving the student with a fewer choices between the correct answer and one or more incorrect ones.

The Hint module also implements the ResourceReference API. The WriteToStream operation generates a XML-based file conforming to the Hint format, while the RevalidateData operation only affects the first section where the contained path entries, if any, are updated with the new artifact-specific locations.

```
<Hints>
        <FirstHint>
                <Definition> A definition of the word to be filled-in the selected sentence </Definition>
                <ImagePath> A representative image </ImagePath>
                <VideoPath> A representative video </VideoPath>
                <AudioPath> A representative audio file </AudioPath>
        </FirstHint>
        <SecondHint>
                <Example> An example of use </Example>
                <Example>
                        More examples
                </Example>
        </SecondHint>
        <ThirdHint>
                <Eliminate> Word A </Eliminate>
                <Eliminate> Word B </Eliminate>
        </ThirdHint>
</Hints>
```

5.2.4 Content Classification and Personalized Delivery

The Data Space is not a simple content repository in terms of a local data holder, but in fact it integrates a sophisticated metadata repository, with references to the actual system's data.

5.2.4.1 Taxonomies Overview

Taxonomy is the practice and science of classification. A taxonomy, or taxonomic scheme, is a particular classification ("the taxonomy of ..."), arranged in a hierarchical structure. Typically, it is organized by supertype-subtype relationships; in such inheritance relationships, the subtype by definition inherits the properties, behaviors, and constraints as the supertype, plus one or more additional properties, behaviors, or constraints. The concept of Taxonomy is particularly suited to the ClassMATE needs, as it allows efficient classification and retrieval of the available content.

The Resource Description Framework (RDF) is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata data model, but it evolved into a general method for conceptual description or modeling of information. The RDF data model is similar to classic conceptual modeling approaches, such as Entity-Relationship or Class diagrams, as it is based upon the idea of making statements about resources in the form of subject-predicate-object expressions, while a collection of RDF statements intrinsically represents a labeled, directed multi-graph. These expressions are known as triples in RDF terminology, where the subject denotes the resource, and the predicate denotes traits or aspects of the resource and expresses a relationship between the subject and the object. RDF's simple data model and ability to model disparate, abstract concepts has also led to its increasing use in knowledge management applications for knowledge representation over the relational model and other ontological models. The ClassMATE system builds the knowledge base using the RDF technology and employs the SPARQL language as the inference channel both for content classification and discovery.

The DataSpace module implements a sophisticated content delivery mechanism to enhance the educational process and support the learners by providing access to related educational material that would otherwise require intensive manual effort to discover. Towards this end, the inclusion of taxonomies in the ClassMATE platform empowers the classification mechanism by introducing in addition to the basic content categorization the notion of the knowledge map between learning objects. The latter is achieved by the hierarchical structure imposed by taxonomies that implicitly connects objects together and increases the semantic coherence of the knowledge map. The introduction of taxonomies, combined with the SPARQL query language and the ClassMATE’s User Profile module, results in a powerful content retrieval mechanism that ensures personalized content delivery according to the learner’s needs. Moreover, the DataSpace module integrates a semi-automatic content

reclassification mechanism through which a newly inserted taxonomy can provide the essential rationale to reclassify the already available content.

The classification-related data are stored independently of the actual content. A LOM-based metadata file accompanies every learning object and any classification-related values are stored in that file. Since learning objects may belong to more than one taxonomy at the same time, the ClassMATE's knowledge base consists of complex directed graphs that should be used during content retrieval. The use externally stored metadata ensures that whenever a learning object is used as the search criterion all the essential information for the query building will be available through its LOM file (e.g., keywords, associated taxonomies). However, a discovery process that iterates the contents of the repository and parses every LOM file to find similarly classified objects is not efficient; thus the SemWeb library is used to store in a relational database (MySQL) the content relations to optimize performance (Fig. 27).

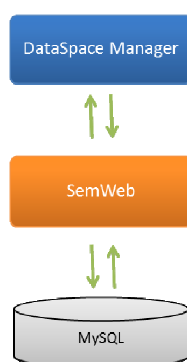


Figure 26: SemWeb Bridge between RDF data and a Relational Database

5.2.4.2 Taxonomies Installation

The RDF technology includes an extensible knowledge representation language, providing basic elements for the description, called RDF Schema (often abbreviated as RDFS). Since the taxonomy's structure is encapsulated in the contained concepts and their relationships, ClassMATE employs the RDFS technology towards taxonomies definition. A concept is modeled as an RDF class, and a relation between two concepts is modeled as a property between their respective classes; the taxonomy's hierarchy is implicitly defined though classes inheritance. Taxonomy outlines the pattern and the hooks where the actual data will be placed. The taxonomy data are represented as RDF instances, encoded in an external RDF file built on top of the taxonomy schema; that file's structure should be fully compliant with the schema specification. Data files do not enclose actual content, rather they hold the taxonomy schema populated with URIs pointing to the appropriate data. For example, an

instance entry of the Classbook taxonomy, defined based on course's structure (i.e., book, section, chapter, etc.), will contain the following values:

```
<Book>
        …
        <Page>
                <hasPageId> 37 </hasPageId>
                <hasImage>
                        6thGrade_EnglishCourseBook_Chapter3_Unit3_Lesson1_RollerCoasterImage
                </hasImage>
        </Page>
        ….
</Book>
```

The TaxonomyLoader is responsible for loading the taxonomy definition and data in the system. The LoadTaxonomy operation parses the RDF schema, collects the contained classes and properties and stores them in the TaxonomyRegistry maintained by the DataSpace. The LoadInstances operation utilizes the TaxonomyRegistry to resolve the appropriate taxonomy components (i.e., classes and properties) and uses them to insert the supplied taxonomy instances in the database. For that to be achieved, the appropriate RDF triples should be generated and fed to the SemWeb library to finalize insertion.

The implemented algorithm is a recursive process, which parses the data file. Whenever an entry is completely loaded, it is immediately inserted in the database. The insertion query as aforementioned is encoded as an RDF triple. Every class instance is either the subject or the object of that triple while every property corresponds to the predicate; however, when an RDF property starts from a class and points a literal value, then the object of that triple is an RDF Literal instance. RDF parsing is a top-down process. However, in TaxonomyLoader's case the "shift-reduce" alternative was implemented, a widely-known mechanism by RDP parsers, supported by an internal stack to "memorize" data during "shift" and facilitate their insertion during "reduce".

5.2.4.3 Content Collection Mechanism

The data gathering procedure is performed on demand by the Data Collector mechanism, which searches in a "transparent" way diverse sources (e.g., web, file system, etc.) in order to discover content related to a particular topic and present it through the appropriate application. The various search criteria necessary to the collection process (e.g., topic, related taxonomies, mime type, etc.) are determined by a particular learning object and the URI of that object is provided to the Data Collector to facilitate the extraction of the necessary information from the linked metadata file (accessible through the URI).

The LOM standard offers a great number of metadata structures to facilitate the content discovery mechanism. The general description and keyword fields are used for “simple” queries, while the educational fields (i.e., semantic density, difficulty and typical learning time) are used for advanced queries. Classification attributes are also used during discovery, through a modular mechanism for searching for relevant content; the information from the classification section facilitate the Data Collector to identify the taxonomies under which the current learning object is classified and use the relative queries to discover relevant content.

An information retrieval process begins when a query is entered into the system. Queries are formal statements of information needs, while a query does not uniquely identify a single object in the collection; instead, several objects may match the query, perhaps with different degrees of relevancy. The queries used in the ClassMATE platform are expressed in SPARQL to take full advantage of the semantic information provided by the RDF technology. Queries are created by experts already acquainted with the taxonomies and the interrelations of the contained concepts.

Every query retrieves data from a particular taxonomy only, whereas a taxonomy can have multiple queries assigned to. Conflicts are eliminated by assigning every query to a specific mime type which indicates the data type of the learning objects that will be eventually discovered. Queries associated with the same mime type form a query family. The Query Registry is used to collects all queries together to facilitate their maintenance. New queries can be easily added in the registry, either under a section that corresponds to an existing taxonomy, or under a new section introduced by a new taxonomy; in the latter case, the added section implies that the new taxonomy has been successfully inserted in the DataSpace. Every query includes some “blank” parameter placeholders that are dynamically filled before execution with values extracted from the leaning object’s metadata file and return the result set in an XML-based format. To facilitate federated queries, queries that correspond to the same MIME type should return the same XML-format, while in the majority of the cases only the URI of the discovered content provides adequate data for the content collector to further process. Figure 28 presents a sample query family, extracted from the actual system, that collects image files by calculating their relevance based on their hierarchical distance (i.e., different images that belong to the same hierarchical branch and share a common ancestor are treated as relevant).

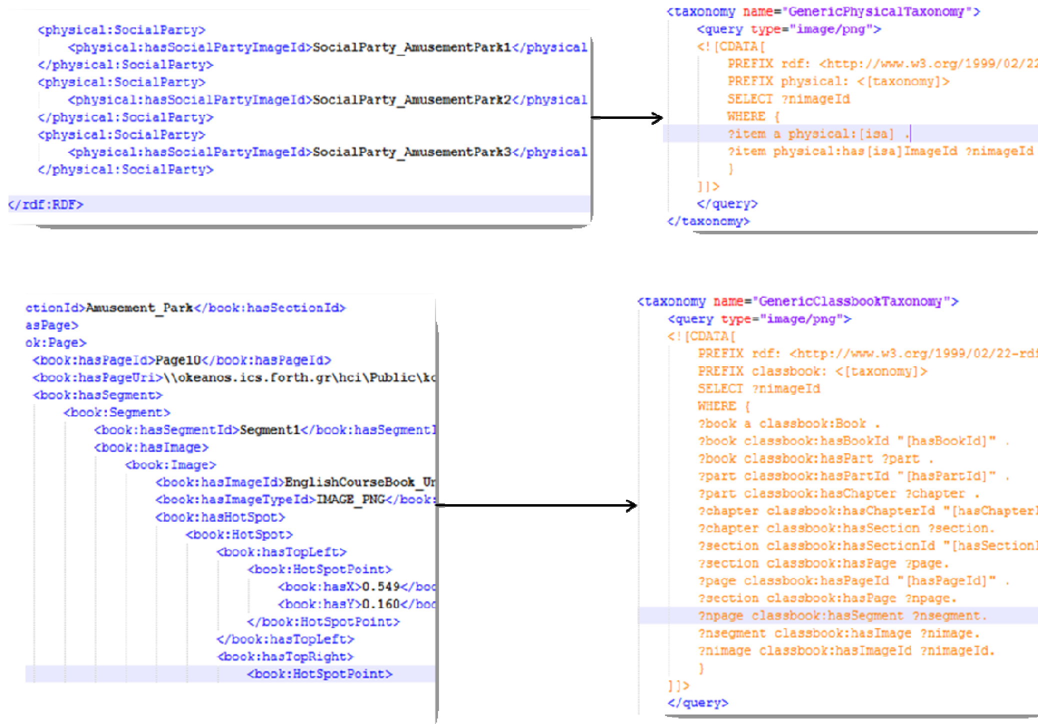


Figure 27: A sample query family that discovers related images

The data collection delegate resolves the appropriate queries from the Query Registry and populates their parameters with data gathered from the learning object's metadata file. The mining process is as follows:

1. Load the metadata of the learning object pointed by the supplied URI
2. Extract the taxonomies under which this learning object is classified
3. Initialize the list of acceptable MIME types with this learning object's type
   3.1. Augment the list of acceptable MIME with any explicitly defined types (e.g., the module that initiates the data collection mechanism is able to explicitly request the inclusion of specific mime types)
4. Request from the Query Registry those queries that match the specified criteria (i.e., taxonomies and mime types)
5. Populate the parameter placeholders for each query and execute it through the SemWeb proxy
6. Collect the results into an federate result set (categorized per mime type) and propagate them to the filtering and personalization module

The filtering process is divided into two distinct phases: the contextual filtering and the personalization. During the initial phase, contextual information is used to determine the

rule set, while during the second phase the remaining content is personalized according to the learner's profile. In more details, if the contextual filtering occurs during school-hours, the course-irrelevant learning objects will be discarded from the result set (i.e., in the context of foreign languages an image of a roller coaster could stimulate the discussion, while the kinetic energy equation is not important); vice versa, if the same scenario occurs during homework, then both learning objects will be returned as they are both equally useful.

The personalization phase begins immediately after the contextual filtering completes; during that phase the learner's profile eventually determines the content to be delivered. The behavior patterns stored in the User Profile clean up the remaining results by discarding those learning objects that do not meet the learner's needs; the LOM metadata are used to determine if a learning object is suited for the particular learner. An illustrative example of this process is the following: a learner wishes to study further on a topic and requests supplementary material in the form of multiple choice exercises that belong to that topic. The data collection process may return numerous exercises that match the search criteria; nevertheless, only the exercises whose difficulty level matches the preferred will be delivered, while the rest will be discarded. Any changes made in the User Profile have a direct impact on the personalization process, as the filtering operations evolve to include these updates.

At this point, the result set encloses the final list of learning objects (in the form of URIs) that should be displayed; however, the ClassMATE-enabled applications require additional information to display them properly. Therefore, the Data Collector instructs the Reference Resource delegates to generate the appropriate description files by extracting any additional information from the respective metadata files.

When the filtering and personalization process completes, the control is passed back to the Data Collector to notify the Artifact Director and eventually present the discovered content to the learner.

5.2.4.4 Content Classification

One of the key features of the ClassMATE's content delivery mechanism is the ability to reclassify content at runtime, thus altering the generated result set. Through that process the knowledge map evolves as new connections are added. The classification process is accomplished via RDF rationale that utilizes taxonomies (e.g., apply rules to select those

resources that derive from a specific base class or that share a specific property). Whenever a new taxonomy is imported, it should be accompanied by a rationale that describes how to classify the existing content under the specific taxonomy, and what should be inserted into the LOM classification section. The rationale is expressed as a set of SPARQL queries (similar to those used in the discovery process) that define the matching criteria. The learning objects that meet these specifications are added into the reclassification result set which update their classification sections to include the new taxonomy. For instance, consider the following scenario where the system currently has classified its content using the initial taxonomy and a new taxonomy regarding great Mathematicians was introduced. A sample rationale would be: classify all the "theorems" under the field of "Mathematics", using the name of the mathematician that proved them.

The reclassification process includes, in addition to the metadata update, a database update to store the latest semantic. To this purpose, a process similar to the file-based taxonomy installation is followed; the reclassification result set is used to dynamically generate a set of RDF triples which the SemWeb proxy imports to the database. Recalling the example from the "Taxonomy Installation" section, a potential reclassification of the book taxonomy would result in a list RDF triples in which the objects of the "hasPage" predicates will be automatically populated with URIs from the reclassification result set.

Finally, since the process of classification is a highly power-demanding task, it is conducted offline by the system, depending on its workload (e.g., overnight or during weekends).

5.2.5 Data Repository

Influenced by the emerging trend of online storage services where files are distributed in the cloud, and the wide acceptance of file sharing protocols, ClassMATE instead of binding to a particular solution, implements an open mechanism where data can be stored anywhere, and the File Manager fetches them when necessary. That way the available storage space explodes exponentially, the introduction of new data sources (e.g., online platforms that host educational material) requires minimum modifications, and the available classroom storage can be utilized as a local cache to decrease loading time.

5.2.5.1 File Manager

The File Manager is the entry point from which the ClassMATE core and the various ClassMATE-enabled applications gain access to educational content. Content is not directly loaded from its original location, but a local copy is made to optimize loading time and act as a cache for future requests. File Manager encapsulates the necessary mechanisms to locate

a particular file in the available repository(ies) and fetch it locally to the current artifact. The adopted approach facilitates the addition of any kind of repositories, local, networked, and distributed, without affecting the applications since the Manager API remains unaltered.

Finally, in addition to the educational content repositories, the File Manager provides access to locally stored dedicated repositories (e.g., state repository) to support fundamental ClassMATE activities such as state suspension/restoration and migration.

5.2.5.2 Content Population

The first time that a specific content is searched by the system, it becomes immediately available to the user who requested it, but it is stored in a pending state as unclassified learning content, which will not be available until it is fully classified. However, whenever new content is stored in pending state, it is actually partially classified by the system, according to its current context of use (e.g., studying course, chapter, etc.), which is known to the Context Manager, and then marked for approval by the teacher. The approval and possible enrichment of its metadata is accomplished offline, followed by a new classification round; upon its completion, the new content becomes available to the entire system as appropriate learning content.

6 Conclusions and Future Work

6.1 Summary

This thesis has presented the ClassMATE architecture, a pervasive computing infrastructure for education, focusing on fundamental issues that should be addressed in order for an AmI educational environment to be supported. The ClassMATE system: (i) enhances the classroom orchestration with context awareness, (ii) addresses heterogeneity in the AmI classroom through well-established software design patterns, and (iii) supports educational content classification and personalized delivery.

CLASSMate has been designed and developed taking into account the educational process which takes place in the classroom and beyond, and in particular the needs and requirements which emerge in the context of typical learning activities. In order to immediately respond and orchestrate the AmI artifacts available in the classroom (e.g., interactive boards, smart desk, etc.) to address the needs of students and teachers effectively and efficiently, ClassMATE introduced the Class Orchestrator and the Local Director modules that monitor the ambient environment and make context-aware decisions. Furthermore, the ClassMATE event type system was defined to satisfy the inter-communication needs between the core and the external applications stemmed by ClassMATE's distributed nature. Finally, the Device Manager offers a generic mechanism for heterogeneous devices manipulation, by encapsulating any platform-dependent operations into abstract APIs.

Concerning student management, ClassMATE provides the User Profile module that not only maintains student personal data, school records, etc., but also incorporates a learners' behavior knowledge library (updated at runtime through user monitoring) for the intelligent environment to provide educational content appropriately adapted to each user's actual learning needs.

The Data Space Manager facilitates content management. It supports actual data distribution in multiple repositories, as access is performed through high-level operations that encapsulate the necessary discovery logic. Moreover, a sophisticated content retrieval mechanism is incorporated that performs "intelligent" semantic queries over the available data to discover and fetch content tailored to the learning needs of the current student.

Finally, a collection of auxiliary mechanisms were implemented to facilitate the use of ClassMATE framework by application developers; indicative examples are the Event Registry,

the Platform Expert, the State Manager, the Query Registry and more.

In summary, the outcomes of the work presented in this thesis include:

- the CLASSMate architecture for the Ambient Intelligent classroom
- the Context Manager for context-aware orchestration of the classroom environment
- the Device Manager for abstracting heterogeneous devices into high-level APIs
- the Data Space Manager for adaptive content discovery and personalized delivery
- the User Profiler for managing user-related information
- a collection of auxiliary programming tools such as the Event and the Classification system.

6.2 Conclusion

In combination with the PUPIL system, which realizes the User Interface of AmI classroom infrastructure, the ClassMATE system empowers scenarios such as the following:

- the student points an image to the electronic version of the book, the Context Manager collaborates with the Data Space Manager to discover and retrieve relevant content and when complete, notifies the PUPIL system to launch the Multimedia application to display that content
- the student send an interesting image to the augmented board in order to discuss it with the entire class; for that to be achieved the Artifact Director collaborates with the Class Orchestrator and the State Manager to deploy the same application to the remote artifact
- the student decides to solve an exercise electronically, points the exercise to the electronic version of the book, the Context Manager in combination with the Data Space Manager determine the application associated with the exercise type and notifies PUPIL system to launch the electronic version of the exercise
- the student asks a hint for a specific exercise, , the Context Manager collaborates with the Data Space Manager to discover, retrieve and personalize the appropriate hints and when complete, notifies the PUPIL system to launch the hint application.

Overall, it can be claimed that this work constitutes a significant first step towards supporting the extensive use of AmI technologies in the context of the classroom and of the educational process in general, by facilitating the development of context-aware

applications through hiding the complexity deriving from their use on various artifacts and devices.

6.3 Future Work

Hereafter, additional steps should be taken to fully support the initial concept.

The next step of this work would be to augment the available content and the classification criteria and then conduct an exhaustive user-based evaluation in order to acquire additional useful feedback from end users regarding the robustness of the system and the matching accuracy of the discovered content. The results of this evaluation would lead to further improvements and extensions of the Data Space Manager in order to better meet the students' needs. Towards this end, ClassMATE can assist the teacher, the ClassMATE system by providing supplementary graphical tools that facilitate content insertion and semantic query editing. The insertion tool should include both a single and a batch mode, where a single or multiple educational elements would be automatically classified and stored in the classroom repository(ies). Moreover, a classification refinement process would facilitate the teacher in customizing a priori the classification process or manually correcting any misclassified content afterwards. The semantic query editor would also increase the added value of the Data Space Manager, especially if it simplifies the mapping process by utilizing already available taxonomies and the conceptual relations among them.

In order to support mobile devices with limited processing power and functionality, some ClassMATE modules should be ported to that particular platform(s).

Possible enhancements to the context-related system include the integration of additional ambient devices in the Device Manager and the extension of the Class Orchestrator to react to natural gestures (e.g., voice commands, gesture recognition, eye-tracking to determine the receiver of a command etc.)

Finally, some general ideas that could empower the ClassMATE system would be to extend the administration facilities to orchestrate the whole school and explore potential interconnections with other ambient school environments or learning material repositories to form a global school network.

7 Bibliography

[1] Abrami, P., Bernard, R., Wade, C., Schmid, R., Borokhovski, E., & Tamim, R. (2008). A Review of E-learning in Canada: A Rough Sketch of the Evidence, Gaps and Promising Directions.

[2] AMIGO. (2008). *Amigo: Ambient intelligence for the networked home environment*.

[3] Antona, M., Margetis, G., Ntoa, S., Leonidis, A., Korozi, M., Paparoulis, G., et al. (2010). Ambient Intelligence in the classroom: an augmented school desk. *Applied Human Factors and Ergonomics.*

[4] Assche, F. (2009). Towards Ambient Schooling.

[5] Bandelloni, R., & Paterno, F. (2004). Flexible Interface Migration. *Proceedings of the 9th international conference on Intelligent user interfaces.*

[6] Bell Communications Research, Inc. (1991). *mailcap - Linux man page*. Retrieved from http://linux.die.net/man/4/mailcap

[7] Bravo, J., Hervás, R., & Chavira, G. (2005). Ubiquitous Computing in the Classroom: An Approach through Identification Process. *Journal of Universal Computer Science* .

[8] Breuer, H., Baloian, N., Sousa, C., & Matsumoto, M. (2007). Interaction Design Patterns for Classroom Environments.

[9] Brusilovsky, P., & Millán, E. (2007). User Models for Adaptive Hypermedia and Adaptive Educational Systems.

[10] Conlan, O., Wade, V., Bruen, C., & Gargan, M. (2006). Multi-model, Metadata Driven Approach to Adaptive Hypermedia Services for Personalized eLearning.

[11] Cook, D. J., & Das, S. K. (2007). How smart are our environments? An updated look at the state of the art.

[12] Cooperstock, J. (2001). Classroom of the Future: Enhancing Education through Augmented reallity. *Proc. Conf. Human-Computer Interaction (HCI Int'l 2001)*, (pp. 688-692).

[13] Faison, T. (2006). *Event-based programming: taking events to the limit.* Apress.

[14] Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns : elements of reusable object-oriented software.*

[15] Georgalis, Y., Grammenos, D., & Stephanidis, C. (2009). Middleware for Ambient Intelligence Environments: Reviewing Requirements and Communication Technologies. *Proc. 13th International Conference on Human-Computer Interaction (HCI International 2009),* (pp. 168-177). San Diego.

[16] Heilman, M., Collins-Thompson, K., & Cal, J. (2006). Classroom Success of an Intelligent Tutoring System for Lexical Practice and Reading Comprehension.

[17] IEEE Learning Technology Standards Committee. (2001). Draft Standard for Learning Technology-Learning Technology Systems Architecture (LTSA). IEEE Computer Society, IEEE 1484.12.1-2002.

[18] IEEE LOM. (2002). Draft Standard for Learning Object Metadata. IEEE Learning Technology Standards Committee, IEEE 1484.12.1-2002.

[19] IMS Global Learning Consortium. (2010). *IMS Learner Information Package Specification*. Retrieved from http://www.imsglobal.org/profiles/

[20] IST Advisory Group. (n.d.). *Scenarios for Ambient Intelligence in 2010.* Retrieved from ftp://ftp.cordis.europa.eu/pub/ist/docs/ istagscenarios2010.pdf

[21] Janse, M., Vink, P., & Georgantas, N. (2008). Amigo Architecture: Service Oriented Architecture for Intelligent Future In-Home Networks. *Constructing Ambient Intelligence*, (pp. 371-378). Springer Berlin Heidelberg.

[22] Korozi, M. (2010 (Unpublished)). PUPIL- Pervasive UI develoPment for the ambIent cLassroom. Heraklion: Computer Science department - University of Crete.

[23] Li, J., & Shi, Y. (2005). Baton: A Service Management System for Coordinating Smart Things in Smart Spaces.

[24] Lin, Y., Kratcoski, A., & Swan, K. (2005). Situated Learning in a Ubiquitous Computing Classroom. *Journal of the Research Center for Educational Technology (RCET)* , 25-38.

[25] Microsoft. (2010). *BinaryFormatter Class*. Retrieved from http://msdn.microsoft.com/en-us/library/system.runtime.serialization.formatters.binary.binaryformatter.aspx

[26] Microsoft. (2010). *Generics (C# Programming Guide)*. Retrieved from http://msdn.microsoft.com/en-us/library/512aeb7t.aspx

[27] Microsoft. (2010). *Input Overview*. Retrieved from http://msdn.microsoft.com/en-us/library/ms754010.aspx

[28] Microsoft. (2010). *Manipulations*. Retrieved from http://msdn.microsoft.com/en-us/library/dd371574(VS.85).aspx

[29] Microsoft. (2010). *Windows Touch*. Retrieved from http://windows.microsoft.com/en-US/windows7/products/features/touch

[30] ORACLE. (2010). *Object Serialization*. Retrieved from http://download-llnw.oracle.com/javase/6/docs/technotes/guides/serialization/

[31] Paredes, M., & Ortega, M. (2002). A Ubiquitous Computing Environment for Language Learning.

[32] Savidis, A., & Stephanidis, C. (2005). Distributed Interface Bits: Dynamic Dialogue Composition from Ambient Computing Resources. *Personal and Ubiquitous Computing*, (pp. 142-168).

[33] Shi, Y., & Xie, E. A. (2003). The smart classroom: Merging technologies for seamless tele-education. *IEEE Pervasive Computing Magazine* .

[34] Shi, Y., Xie, W., & Xu, G. (2002). Smart Remote Classroom: Creating a Revolutionary Real-Time Interactive Distance Learning System.

[35] Soh, L.-K., Khandaker, N., & Jiang, H. (2008). I-MINDS: A Multiagent System for Intelligent Computer-Supported Collaborative Learning and Classroom Management. *International Journal of Artificial Intelligence in Education 18* , 119-151.

[36] Tauberer, J. (2010). *SemWeb.NET: Semantic Web/RDF Library for C#/.NET*. Retrieved from http://razor.occams.info/code/semweb/

[37] Vassileva, D., & Bontchev, B. (2006). Self Adaptive Hypermedia Navigation Based On Learner Model Characters.

[38] W3C. (2004). *Resource Description Framework (RDF)*. Retrieved from http://www.w3.org/RDF/

[39] W3C. (2008). *SPARQL Query Language for RDF*. Retrieved from http://www.w3.org/TR/rdf-sparql-query/

[40] Wikipedia. (2010). *LDAP*. Retrieved from http://en.wikipedia.org/wiki/LDAP

[41] Wikipedia. (2010). *Metadata discovery*. Retrieved from http://en.wikipedia.org/wiki/Metadata_discovery

[42] Xu, P., & Han, G. (2009). Towards Intelligent Interaction in Classroom. In *Universal Access in Human-Computer Interaction.*

[43] Yau, S. S., Gupta, S., & Karim, F. (2003). Smart Classroom: Enhancing Collaborative Learning Using Pervasive Computing Technology.

APPENDIX A

General Section

- the list of globally unique labels that identifies this learning object; both the catalog scheme and the value of the identifier for that scheme are required
- the given title given
- the human languages used in its content
- a short textual description
- a list of keywords the describe the related topic
- the time, culture, geography or region to which this learning object applies
- an enumeration describing its organization structure
  - **value space:** atomic, collection, networked, hierarchical, linear
- an enumeration describing its aggregation level
  - **value space:** raw data, collection of raw data (e.g. a lesson), collection of collections (e.g. course), set of collection (e.g. set of courses)

Technical Section

- the format of this learning object expressed as a MIME type
- the size in bytes required on a physical storage device
- the list of URIs used to access it
- the system requirements necessary for using it (e.g. browser, operating system, etc)
- a list of installation remarks
- a list of other software or hardware requirements
- the time that this learning object takes to be played at intended speed

Educational Section

- an enumeration describing its predominant mode of learning
  - **value space**: active, expositive, mixed
- an enumeration describing its specific type
  - **value space:** exercise, simulation, questionnaire, diagram, figure, graph, index, slide, table, narrative text, exam, experiment, problem statement, self-assessment, lecture
- an enumeration describing the degree of interactivity that characterizes it
  - **value space:** very low, low, medium, high, very high
- an enumeration describing the degree of its conciseness

    - **value space:** very low, low, medium, high, very high
- an enumeration describing the intended user(s) for which it was designed
    - **value space:** teacher, author, learner, manager
- an enumeration describing the environment within which the learning object will be used
    - **value space:** school, higher education, training, other
- the age of the typical intended user
- an enumeration describing the difficulty level
    - **value space:** very easy, easy, medium, difficult, very difficult
- the approximate or typical time it takes to work with or through the learning object
- a list of comments about its use
- the human language used by the typical intended user

Relation Section

- an enumeration describing the nature of the relationship between this learning object and the target learning object
    - **value space**: ispartof, haspart, isversionof, hasversion, isformatof, hasformat, references, isreferencedby, isbasedon, isbasisfor, requires, isrequiredby
- the target learning object that this relationship references; the structure is similar to the identifier under the general section

Classification Section

- an enumeration describing the purpose of classifying this learning object
    - **value space**: discipline, idea, prerequisite, educationalobjective, accessibility, restrictions, educationallevel, skilllevel, securitylevel, competency
- a taxonomic path in a specific classification system
- a description of the learning object relative to the purpose of the classification

  a list of keywords and phrases that describe the learning object relative to the purpose of the classification