Digital Rights Enforcement for Pervasive Computing Applications

Dominik Roman Christian Dahlem

A thesis submitted to the University of Dublin, Trinity College

in partial fulfillment of the requirements for the degree of

Master of Science

March 2005

Declaration

I, the undersigned, declare that this work has not previously been submitted to this or any other University, and that unless otherwise stated, it is entirely my own work.

Dominik Roman Christian Dahlem

Dated: 26th October 2005

Permission to Lend and/or Copy

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

Dominik Roman Christian Dahlem

Dated: 26th October 2005

Acknowledgements

Special thanks go to my supervisor Dr. Jim Dowling for his commitment, attention to detail, and who was at any time a great source of inspiration. The other main acknowledgements go to Ivana Dusparic for her valuable collaboration on the project and Elizabeth Daly who proof-read my thesis throughout the writing.

As well, I would like to express my thanks to the members of DSG for providing a creative atmosphere and an environment where unlimited academic freedom lead to the result of this thesis.

And finally, to my family and friends for their love and encouragement.

Dominik Roman Christian Dahlem
University of Dublin, Trinity College
March 2005

Contents

List of Figures

Listings

List of Tables

Abstract

Increasingly, application software is expanding from the desktop into mobile application environments, such as handset devices and embedded systems which are more limited in resources and volatile in their network connectivity. An integrated architecture that can protect intellectual property for both types of environments should offer the promise of reduced software maintenance costs. Software licensing is an existing mechanism by which specific license agreements are enforced between a software provider and the users of the software. Usually, the license terms are activated by a unique activation code delivered to the user. Digital Rights Management (DRM) is a more recent development that covers the description, identification, trading, protection, monitoring, and tracking of all forms of rights usage over both tangible and intangible assets. This includes the management of Rights Holder relationships with the help of special purpose rights expression languages (REL). Both, software licensing and DRM approaches have failed to address the new challenges posed by protecting intellectual property for mobile application software. This thesis, therefore, proposes a solution to merge the best of both approaches for the special case of application software rights enforcement. It is targeted to mobile computing platforms to meet the challenges in that area.

Existing distributed software licensing systems were originally designed for fixed network applications and typically assume the immediate availability of a network connection to verify and validate rights with a rights server. Yet, this approach is not feasible for mobile environments, because of occasional connected network characteristics. Moreover, software licensing systems do not implement an existing standard for the description of license terms which cause interoperability issues with asset management software. The focus of the DRM community to date has been on rights management for media content, which has left many issues unresolved for the specific case of rights management for application software. The difficulties of developing an all-encompassing DRM solution for the media industry has left standards-based work on the enforcement of rights for application software under-specified. This is mainly because the media industry requires a broad consensus of hardware and software manufacturers to implement an agreed standard, whereas application software does not require runtime support of the underlying hardware or any third party applications.

The existing rights expression languages supported by DRM systems lack the support for the explicit specification of application-level features. Existing usage-based restrictions on digital work usually include display, print, play, and execute permissions. Also, the assumption of immediate or constant

network connectivity to a rights management server cannot be made for the validation and enforcement of rights on a pervasive computing platform, because factors such as network unavailability have to be anticipated. Therefore, the introduction of more flexible rights models for occasionally connected mobile environments is required. This is achieved through the specification and implementation of novel rights models, such as audit-based and feature-based models.

The introduction of these flexible rights models poses new challenges to designing an enforcement architecture for pervasive environments. The enforcement architecture has to deal with resource constraints on mobile devices, such as limited memory and processor power, while at the same time provide an extensible set of APIs so that it can be adapted for different computing platforms.

This thesis proposes a solution to enforce and deliver application software rights implemented in a generic enforcement framework, based on an extended version of the Open Digital Rights Language (ODRL), called PARMA REL, that accounts for the characteristics of applications in pervasive environments. In particular, the architecture supports the enforcement of audit-based and feature-based rights models. While the architecture in this thesis has specific support for mobile environments, it has also been designed to operate in a fixed network environment. A further contribution of this thesis is to present a pervasive application rights enforcement framework which does not make any assumptions on the target platform by basing the design on the dependency inversion and Hollywood principles. The architecture is designed in a way that decouples functional and rights enforcement logic. It supports the association of rights with application-level features by leveraging aspect-oriented software engineering techniques to weave the enforcement as an orthogonal service into any existing J2ME, J2SE, or J2EE application. This makes it possible to restrict access to certain modules at runtime. Developer support is provided by tools to generate aspects based on the rights description and the target platform. Furthermore, a MDA-oriented development process is introduced to cover the generation and weaving of rights into the application in a non-intrusive manner.

Consequently, the rights models designed for pervasive computing environments combined with the flexible enforcement architecture enable the enforcement of rights of applications in new, sophisticated and standard-compliant ways. The enforcement architecture is evaluated with respect to the ability to adapt to different platforms, to operate in resource-constrained environments, and to guard against potential attacks. Also the execution and runtime overhead of the enforcement logic is evaluated and the architecture is compared with existing enforcement architectures. The enforcement architecture is implemented for two platforms, J2ME and J2SE.

Chapter 1

Introduction

This thesis presents a Digital Rights Enforcement Architecture for rights-enabled applications, independent of their target platform. It supports the enforcement of application rights specified in an extended version of Open Digital Rights Language (ODRL) [55], called PARMA. PARMA is designed to meet the requirements for pervasive environments by specifying rights models that function in mobile environments with intermittent network connectivity. This chapter motivates the work, introduces the concepts of pervasive rights enforcement, presents the scope and finally outlines a roadmap for the thesis.

1.1 Motivation

1.1.1 Pervasive Computing Platform

Wireless networks and mobile computing are increasingly becoming integrated into everyday life. Mobile devices such as embedded devices, Personal Digital Assistants (PDA) or mobile phones provide users with the ability to access relevant information or services anywhere and anytime. The field of mobile computing and its transparent integration into a ubiquitous network infrastructure is referred to as pervasive computing [107, 43].

The aim of pervasive computing is to enable mobile devices and services to easily inter-operate over both wired and wireless communication channels. The communication technology is completely hidden from the application and ideally applications choose the most appropriate, e.g., cheapest, option available to interact with remote services [21]. A parallel research effort in pervasive computing deals with improving the human computer interaction by reducing explicit and implicit user distraction. The goal of pervasive computing is the seamless integration of computing, information services, and the human user to provide the user with more effective ways of accomplishing their tasks [99].

The recent development of applications for mobile environments is mainly driven by two factors. Firstly, simple development platforms are starting to appear such as J2ME [12] with its optional APIs and secondly, hardware advancements such as CPU capacity, memory availability, network, and commu-

nication technology [54]. The movement to pervasive computing platforms has introduced a number of challenges. For example assumptions from fixed network environments, such as constant and immediate network connection may not be valid, because there may not be any network connection available, and the assumption that remote calls do not incur a charge is not always valid, as common communication technology, such as GPRS, introduces costs to the user. Also, mobile devices have limited memory and processing capabilities.

So from the systems viewpoint, pervasive computing platforms present the challenge of dealing with highly dynamic and volatile network environments and therefore have to be designed for failure. Notably, unanticipated short- or long-term network disconnections have to be transparently handled at the transport or application layers of the OSI protocol stack without leaving the whole application inoperable. From the viewpoint of the application domain, enforcement of digital rights must address the aforementioned characteristics of pervasive computing environments. Enforcement has to be designed with the ability to adapt to network availability. If a network connection becomes unavailable while in the process of updating the usage rights, the application has to gracefully respond to the user, yet, at the same time enforce the rights effectively.

1.1.2 Software Licensing and Digital Rights Management

Traditional software licensing systems, such as Macrovision's FlexLM [19], provide mechanisms by which specific license terms are enforced between the software provider and the user. A software provider delivers a unique activation key to the user to unlock the protection mechanism in the software. The activation key itself only is a weak protection guarantee for the software provider, because these keys can be illegally distributed to other users. Software providers, therefore, demand a network connection upon start up of the application to a license server to verify and validate the license terms based on the activation key. The assumption of establishing an on-demand network connection violates the objective for the design of a pervasive application. Also, existing software licensing platforms [19] and standards, such as XSLM [41] require the specification of usage rights in applications by providing language- and platform-specific bindings to their licensing architectures. These solutions require programmer knowledge of proprietary application programming interfaces (API) and do not provide the integration and flexibility required for a more ubiquitous software deployment. In contrast, this thesis seeks a standard-based way of protecting software using an extension to ODRL, called PARMA. ODRL is an open XML-based standard for representing usage rights.

Digital Rights Management technologies provide a set of mechanisms to create, manage, deliver, and enforce rights. But so far, the DRM community has been mostly concerned with viewing and managing digital content in a controlled fashion. To date, there has been no satisfactory solution to applying these techniques to software applications, mainly because efforts are guided towards an all-encompassing DRM system [58] that combine legal, business, and technological aspects. Rather than specifying a complete rights management standard, this thesis adopts a different approach to attempt to handle the specific

case of rights enforcement for software applications.

Despite the extensive ongoing research into DRM, the music, film, and print industry so far have not succeeded in presenting a working DRM solution, because every audio and video stream can be copied with little effort [83]. Many cryptographic techniques such as the Content Scrambling Systems (CSS) for DVDs have failed, because soon after release illegal decryption programs, such as deCSS, became available which circumvented the security measures. Additionally, incompatibilities between DVD drives and the DVD stream caused problems in viewing the content. So, even though hardware enforcement solutions are feasible, they are not very practical. All viewing and recording devices would have to be manufactured according to a broad industry consensus. So, every DRM scheme forced on to a user can be bypassed by recording the original decrypted content, while it is being played.

In contrast enforcing rights for software is easier to achieve, because the rights description, the enforcement component, and the original application can be bundled together without the need for broad consensus between hardware and software manufacturers. Hence, no hardware or software support for DRM is needed. However, cryptographic techniques to encrypt software applications before they are distributed to the user and decrypt them at runtime such as in [65] are problematic. Similar to the problem of encrypting media content the original decrypted software application can be retrieved with little effort [93].

One advantage of a self-contained software application with attached digital rights and an enforcement component is the possibility to exploit new distribution channels. For example, much software and digital content is currently accessed via P2P networks and technologies, such as Bluetooth enabled super distribution using handset devices. The current model adopted by companies such as Nokia, however, is to avoid the use of these channels and restrict the distribution of mobile application software. For example, existing DRM architectures, such as Nokia's [104], encourages the use of forward lock to prevent the unauthorized further distribution of applications. The forward lock mechanism is implemented in the operating system to avoid the super-distribution of certain files based on their extension. This model is delaying the adoption of such software, as users are unwilling to pay either the high cost of application download over GPRS networks or the full-cost rights for the application software [79]. However, the low cost of distribution of digital media and applications over Private Area Networks (PAN) such as Bluetooth contrasts sharply with the high cost of GPRS. Given a choice, users would prefer to acquire applications over a communication channel that is free instead of paying for distribution as well as the application. Default evaluation rights attached to a software distribution would create a business model of "try-before-buy" without imposing any costs on the user.

1.1.3 Requirements for the Enforcement Architecture

As a result of the characteristics of pervasive computing environments and the failure of existing DRM solutions to target mobile applications a list of non-functional requirements dictate the design of the enforcement architecture. These requirements cover three areas. First, the architecture has to be exten-

sible and adaptive, because no assumptions about the target platform can be made. The enforcement has to work in similar ways on different target platforms. Second, enforcement should be transparent to compliant users and application software developers of PARMA. The enforcement has to seamlessly integrate into the application and should not change the user perception of the application dynamics, for instance affecting user experience due to a high runtime overhead. Consequently, the enforcement has to deal with platform-specific resource constraints. From the developer's perspective the integration of the enforcement should be as transparent as possible, ideally without having knowledge of enforcement APIs. Finally, one of the main requirements is to mask network failure to account for the characteristics of pervasive computing environments. This can be achieved by supporting an audit-based rights model that is designed to operate in occasionally-connected network environments.

1.2 Contributions

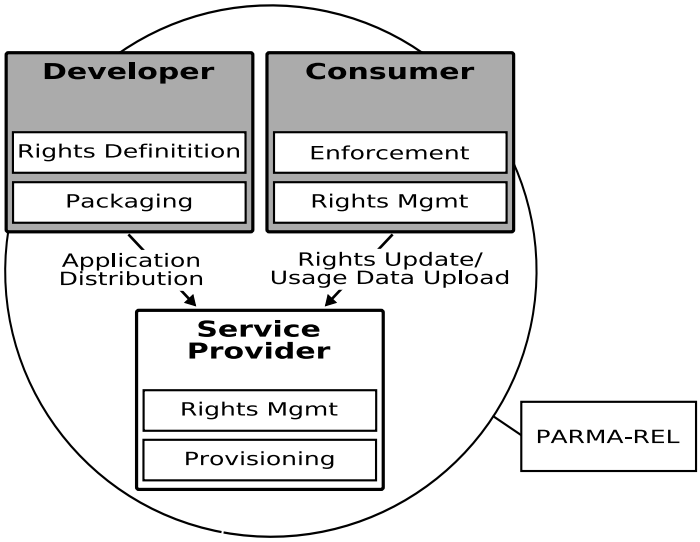

Figure 1.1: PARMA Architecture Overview

The PARMA enforcement architecture is part of an overall DRM architecture, called PARMA [55]. PARMA provides an integrated platform for the specification, generation, delivery, and management of application usage rights for pervasive computing environments. PARMA also specifies an REL which extends ODRL. PARMA REL [55, 56, 52] is developed to express flexible usage rights targeted towards mobile software applications. It extends ODRL REL in two ways by adding elements to the already existing usage models. First, it specifies an audit-based model to account for occasionally-connected network characteristics. Second, a feature-based model is introduced to restrict the execution of software application features. The application features are essentially designated methods that implement them.

As part of PARMA, the PARMA enforcement architecture implements a mechanism to enforce rights expressed with PARMA REL, including rights expressed in audit-based or feature-based models. The

implementation covers (marked grey in Figure 1.1, whereas the PARMA REL and the back-end server was developed by Ivana Dusparic):

- A container architecture (Section 1.2.1) to provide a platform-independent execution environment for the PARMA enforcement architecture that is also suitable for J2ME devices.
- The PARMA enforcement architecture (Section 1.2.2) based on the PARMA REL that was developed in a collaborative effort with Ivana Dusparic.
- And the generation and weaving of enforcement logic into existing applications (Section 1.2.3) using Aspect-Oriented Software Development techniques.

The following sections summarise the contributions of the enforcement architecture.

1.2.1 Container Architecture for Pervasive Environments

Often, enterprise software is faced with structural complexity. Mostly, this is due to the fact that individual components have dependencies to other components which realise access to services such as a persistent store, external resources or external business logic. This complexity arises when each component is responsible for resolving its dependencies. One approach to address this problem is by implementing the Inversion of Control (IoC) and Dependency Injection (DI) patterns. In this case, satisfying dependencies is externalised into a container. The container is responsible for resolving dependencies between its registered components and therefore removes this logic from each component. Consequently, the container can be configured to account for different platform-specific characteristics which is one of the requirements of the enforcement architecture in this thesis (Section 1.1.3). With the help of the container pattern, the enforcement framework could be designed to achieve a high degree of flexibility and adaptability, because of the support for dependency configuration for different platforms.

There are several implementations of generic container architectures that conform to the IoC principle, including Picocontainer [26], Spring Framework [31], and Hivemind [11]. However, due to limitations dictated by mobile computing environments, such as the lack of reflection in the MIDP APIs [109], these containers are not suitable for the enforcement architecture. In MIDP applications it is not possible to reflect on an interface of a component in order to find out which components it depends on. In existing systems, the container reflects on the interface, the constructor, or the setter methods depending on the type of Inversion of Control the container represents. The advantage of container architectures is that individual components can be exchanged and therefore provide more flexibility, e.g., different implementations of the rights enforcement architecture can be provided to account for different rights expression standards.

As a result of the limitations of existing container architectures for mobile environments, this thesis provides a simple, general purpose implementation of such a container which is independent of the computing platform. It implements a variant to the setter-based approach of injecting dependencies.

When a component is registered with the container it also has to declare any interfaces it depends on. At runtime the container is able to find the appropriate implementation and injects an object implementing this interface into the component.

Although this container architecture is a general purpose implementation, its only aim is to provide a platform-independent way of assembling the proposed enforcement architecture.

1.2.2 Pervasive Enforcement Architecture

Enforcing application rights usually is a computationally expensive task for several reasons. If the application rights are represented in XML, or in any other format, an object representation of the rights has to be loaded into memory at runtime to retrieve and validate its contractual terms. Also, traditional distributed software licensing applications require an on-demand network connection to a rights server to be able to validate the rights in order to ensure that it has not been tampered with. The pervasive enforcement architecture overcomes limitations of occasionally-connected networks by providing a metered approach to access usage rights according to an audit-based rights model introduced by PARMA.

To address memory constraints on mobile devices the rights file is deserialized into an object model with a pull parser approach [96]. Unlike the Document Object Model (DOM) [66] parsers, no full object representation of the XML file is built into memory in pull parsers. But rather the application is responsible for pulling information from an XML file. Therefore, the memory footprint of a pull parser can be much smaller and is more suitable for mobile applications.

From a systems viewpoint the pervasive enforcement architecture comprises a set of components which are assembled with the help of the container framework. The main component in this architecture is responsible for enforcing rights expressed in PARMA REL. Other components include the configuration to parse the rights into an object model, the recorder to meter application usage, and daemon services which provide incoming and outgoing communication channels to the enforcement architecture.

The only components which require platform-dependent implementations are the recorder and the daemon services. This thesis provides two implementations of the recorder. One which uses the Record Management Systems [63] for J2ME devices and another one which uses a simple file-based approach for J2SE.

1.2.3 Generating Rights Enforcement Logic using XSLT Stylesheets

Often the modularity structure of a system architecture comprised of enterprise applications starts to increase in complexity due to the interaction of too many orthogonal concerns imposed from a wide range of application requirements. One such requirement is the integration of some DRM enforcement into applications. Traditionally, the application has knowledge of enforcement APIs which increases the structural complexity of the application. Application logic becomes tangled with logic concerned with enforcing rights. Aspect-oriented programming is a development technique that is capable of composing

orthogonal design requirements late in the development life-cycle of application software [74]. In this thesis aspects are used to encapsulate rights enforcement logic, enabling any software application to be woven with its rights enforcement code and the enforcement architecture.

In order to modularise enforcement-specific logic in the proposed architecture and to provide integration transparency (Section 1.1.3) to the developer a model-driven architecture (MDA) [62, 75] approach is used. MDA envisions an approach to separate the specification of system functionality from the application logic of the underlying platform technology. This thesis follows this vision by providing a platform-independent viewpoint on the enforcement of rights realised by the PARMA REL (Platform-Independent Model - PIM), a platform-dependent viewpoint realised by aspects which address the specifics of the target platform (Platform-Specific Model - PSM), and a transformation which translates the PIM into the PSM depending on the chosen target platform.

Aspect-oriented software development (AOSD) techniques realise the implementation of the PSM by providing different aspects for different target platforms to account for the platform specific characteristics. AOSD is capable of weaving orthogonal services into functional units of code, such as objects. This thesis uses AspectJ [3, 74] to weave the enforcement architecture into an existing application. The aspect is generated using XSLT [40] stylesheets based on the REL specification of PARMA.

The stylesheets are specific to the target platform. Dynamic join point features of AspectJ can not be applied to J2ME, because their underlying implementation is based on the reflection API. This fact requires the generation of one pointcut and one advice per specified feature in the rights file. In contrast, J2SE allows for dynamic join point features which results in the generation of one pointcut and a maximum of three advices in total. Two stylesheets are provided, one for MIDP and one for J2SE applications.

1.3 Scope

The scope of this thesis is to present an architecture for rights enforcement in pervasive computing environments. The objectives are to meet the non-functional requirements defined in Section 1.1.3 and to support the flexible rights models provided by PARMA. Further, PARMA uses the concept of AOSD in order to integrate rights into application software in a non-intrusive way. This thesis provides an implementation of this concept with an emphasis on the support of multiple platforms.

Even though security relevant questions are considered to be important for the success of a rights enforcement framework, this thesis does not employ any security measures. Data and channel security and privacy issues are left for future work.

1.4 Roadmap

The structure of the remainder of the thesis is as follows: Chapter 2 presents a survey of background material and related research in the field. It highlights the achievements and limitations of existing work

in rights enforcement, as well as recent approaches to building pervasive applications. Chapter 3 introduces the design of the pervasive application rights enforcement architectures. Chapter 4 presents the implementation of the enforcement architecture in MIDP and J2SE. Chapter 5 evaluates the enforcement architecture with respect to the adaptability to different platforms, resource-constrained environments, and potential attacks. Chapter 6 presents conclusions and future work.

Chapter 2

Background and Related Work

This chapter establishes the foundation of the thesis. Its aim is to introduce background work on container architectures and Digital Rights Management (DRM). The special purpose Rights Expression Language (REL) ODRL is presented followed by an introduction of PARMA REL which this work builds on. Related work with an emphasis on protection mechanisms is discussed and a comparison of related works is provided that identifies limitations with existing enforcement schemes.

2.1 Container Architectures

2.1.1 Introduction

Often, enterprise software is faced with structural complexity. In most cases, this is due to the fact that several generations of software engineering, often with different techniques, went into the construction of such a system. Also, the likelihood of modules collaborating with a growing number of dependencies over time is high. In particular, if these modules are responsible for resolving their dependencies an ongoing maintenance challenge is posed to the development team. Container architectures attempt to solve this problem by providing a consistent infrastructure to look up services and manage their life-cycle.

2.1.2 Overview

Container Architectures became very prominent with the advent of component-oriented programming. While object-oriented programming constitutes a technology for software development, component-oriented programming is more a packaging and distribution technology. A component is a unit of functionality or service, which is self-contained and exposes well-formed interfaces. Communication between components takes place on these interfaces. Hence, there is a clean separation of interfaces as "black-box view" and the implementation as "white-box view". This separation ensures independence in their development, delivery, and installation [110]. It also has the beneficial consequence of pluggability, which means that calling logic is isolated from the implementation strategy. Szyperski defines a

component as:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition [103].

One of the underlying principles of container architectures is a software architecture pattern called the Dependency Inversion Principle (DIP). The DI Principle states that:

- High level components should not depend on low level components by establishing dependencies to abstract representations instead;
- And abstractions should not depend on details. Details should depend on abstractions [84].

DIP is also known as Inversion of Control (IoC) [72] or Hollywood Principle ("Don't call us we'll call you"). These patterns provide an infrastructure for managing registered components. Usually, these infrastructural services include life-cycle management, a way of obtaining references to instantiated components, a consistent way of configuring components, and the management of relationships between components.

2.1.3 Implementation Strategies of IoC

The Hollywood Principle refers to one of the two implementation strategies of IoC, dependency lookup and dependency injection [72]. Dependency lookup is an active mechanism by which the component provides a container callback method and a lookup context, such as in EJBs [92, 89] and Servlets [69]. The responsibility is left to the component to use container APIs to look up resources and collaborators.

Listing 2.1: Dependency Lookup

```
1  public class MyBusinessObject implements MyBusinessInterface
2  {
3      private DataSource ds;
4      private MyCollaborator collaborator;
5      private Integer myIntValue;
6
7
8      public MyBusinessObject() throws NamingException
9      {
10         Context ctx = null;
11
12         try {
13             ctx = new InitialContext();
14             ds = (DataSource) ctx.lookup("java:comp/env/dataSourceName");
```

```
15              collaborator = (MyCollaborator) ctx.lookup("java:comp/env/
                    myCollaboratorName");
16              myIntValue = (Integer) ctx.lookup("java:comp/env/myIntValueName");
17          } finally {
18              try {
19                  if (ctx != null) {
20                      ctx.close();
21                  }
22              } catch (NamingException ne) {
23              }
24          }
25      }
26
27      // business methods.
28  }
```

The class `MyBusinessObject` in Listing 2.1 shows an example of dependency lookup. With the help of Java Naming and Directory Interface (JNDI) [78] the object implementing the `DataSource`, the collaborator `MyCollaborator`, and the value of the private member `myIntValue` are externalised from this class. Despite it being implementation-independent, this class makes the assumption that JNDI is available, which is not the case outside an application server. Consequently, this class introduces additional complexity by being responsible for obtaining its dependencies. It is not strongly typed, because the `Object` returned from the lookup has to be type cast to the corresponding object.

These deficiencies are addressed by the second implementation strategy, dependency injection (DI). Compared to dependency lookup DI is typically preferable, because it does not depend on special container APIs, such as JNDI calls. Moreover, the container takes full responsibility of resolving dependencies and injecting them into the component. Three types of dependency injections are usually differentiated:

- Type 1 IoC (interface injection - Listing 2.2);
- Type 2 IoC (setter injection - Listing 2.3);
- And Type 3 IoC (constructor injection - Listing 2.4).

Listing 2.2: Interface Injection

```
1  interface InjectConfiguration
2  {
3      void injectConfiguration(Configuration configuration);
4  }
5
```

```
6
7 public class Enforcement implements InjectConfiguration
8 {
9     private Configuration config;
10
11     public void injectConfiguration(Configuration configuration)
12     {
13         this.config = configuration;
14     }
15 }
```

Listing 2.2 shows an example of interface injection. With this technique an interface `InjectConfiguration` has to be declared and used for the injection. Any class which is interested in using `Configuration` has to implement this interface.

Listing 2.3: Setter Injection

```
1 public class Enforcement
2 {
3     private Configuration config;
4
5     public void setConfiguration(Configuration configuration)
6     {
7         this.config = configuration;
8     }
9 }
```

Listing 2.3 is an implementation of the setter injection. After the instantiation of the `Enforcement` class, the `Configuration` object is set by the container calling the `setConfiguration()` method.

Listing 2.4: Constructor Injection

```
1 public class Enforcement implements InjectConfiguration
2 {
3     private Configuration config;
4
5     public Enforcement(Configuration configuration)
6     {
7         this.config = configuration;
8     }
9 }
```

In Listing 2.4 an implementation of constructor injection is shown. In existing systems all three DI types use reflection mechanisms to determine the dependent class and do not require extra registration steps with the container.

An IoC container and DIP exhibit some advantages when designing frameworks to avoid rigidity, fragility, and immobility [84]. Rigidity refers to the complexity involved in modifying a component, because it involves a cascade of changes in dependent components. A design is referred to as fragile, if small changes in one component causes problems in components which do not have a conceptual relationship with the one that was changed. Finally, immobility is the inability to use a component in another application.

2.2 Digital Rights Management

2.2.1 Introduction

The ease of distribution of digital work enabled by the Internet created the demand to protect the copyright of digital work, because recent technical advancements simplify the process of illegal copying. Traditional media, such as books, impose some barrier to unauthorized exploitation. It is time-consuming to copy and usually results in a loss in quality. Digital work does not fall into this category. In general, copying a video, a piece of music or software products do not require prohibitive amount of time or even does not require a technically skilled person.

Digital Rights Management envisions measures to prevent piracy of digital works or at least to increase the effort it involves to infringe on copyrights. A DRM System (DRMS) acts as a hardware or software safeguard to determine whether a user's access is granted or blocked. The evolution of DRM systems so far, can be divided into three generations [70]. The first generation DRMS covered written statements of usage rights associated with the copyrighted work. Consequently, it was not possible for computer systems to automatically interpret them. The second generation of DRM concentrated on cryptographic means of securing content. In particular, the content was encrypted and only distributed to those who paid for it. Pioneering projects in this era were Tod Nelson's Xanadu [90] project in the early 1980s and an EU-funded ESPRIT project CITED [50] in the 1990s. The third generation focuses more on the full life-cycle of rights and copyrighted works. It covers the identification, description, trading, protection, monitoring, and tracking of rights usages including the management of copyrights holders relationships.

Despite its recent research and industrial interest, there is a lack of a commonly accepted definition for DRM. The DRM Group established by the European Standards Committee/Information Society Standardization System (CEN/ISSS) cover standardization initiatives and the status of DRM in a report published in 2003 [42]. The report attempts to provide some useful definitions in the field of DRM. Throughout this thesis, the definition of DRM is taken from Federation of European Publishers (FEP) [9]:

> DRM can be separated into two distinct layers: The identification and description of intel-

lectual property, rights pertaining to works and parties (digital rights management). The (technical) enforcement of usage restrictions (digital management of rights) [42].

According to Barrow, a DRMS can be divided into a functional five-layer model [44]:

1. Legal layer: This layer aims at a harmonization of global legal and regulatory frameworks to consistently enforce laws. One such example is the Security Systems Standards and Certification Act (SSSCA) [27] in the United States which attempts to force device manufacturers to build DRM functionality, including copy protection, into their products.

2. Contractual and license layer: The contractual and license layer provides a means of expressing contractual terms in a machine-readable and easily exchangeable way. Contract terms are usually expressed in Rights Expression Languages (REL) [39, 25, 24] which provide a syntax and semantics to identify the parties involved, the copyrighted work(s), and access rules over the work.

3. Copyright accounting and settlement systems layer (CASS): CASS provides payment services in a DRMS.

4. Digital rights management layer: This layer represents the administration of rights including tracking and the management of the relationships of the rights-holders. It covers functionalities from provisioning servers, customer relationship, and asset management systems.

5. Technical copyright protection systems (TCPS): TCPS implement the enforcement of rights in order to protect unauthorized use by various technical means. Enforcement is based on two main approaches, containment and marking. Containment refers to the encryption of the copyrighted work so that only valid key-holders are permitted access to encrypted data. Marking describes mechanisms by which some hidden information is injected into the copyrighted work without altering its behavior or visible content. It is intended to complement cryptographic measures to track retransmission and reproduction of copyrighted work. The two marking schemes are watermarking [51, 88] and fingerprinting [45]. Marking ideally is an invisible identification code embedded in the data without rendering it useless where fingerprints carry additional personalized information. The enforcement component can be either implemented in hardware or software or in both. It deals with permission management to the copyright work by honouring the rights associated with the work and/or verifying an embedded mark. Furthermore it enables the tracking of usage.

The DRM areas addressed by this thesis are technical copyright protection systems or technical protection measures, and in particular the enforcement of rights expressed in PARMA REL. As mentioned previously, technical protection measures require a broad consensus for media type content. Hardware and software protection mechanisms have to be adopted for a successful enforcement. In contrast, application software allows different approaches to prevent piracy. A software installation usually requires an ID or an activation code which is obtained with the purchase. A strong enforcement of rights for an

application might be based on cryptographic techniques and requires a tamper-resistant device, such as a smart card, which contains a secret decryption key and a smart card reader. This approach is very costly and not widely adopted. A weaker approach is to establish an on-demand network connection to the software application vendor to activate a product. The user is required to enter the unique serial number. The serial number is verified against a database so that it prevents other users from installing the software with the same serial number.

Depending on the employed mechanism the application and enforcement logic is more or less tightly-coupled. Therefore, the enforcement can be distinguished on a conceptual level:

- explicit enforcement,
- implicit enforcement.

The meaning of explicit enforcement refers to the traditional methodology of implementing protection measures which involves the application having knowledge of enforcement APIs. Consequently, the enforcement logic has to be explicitly added to the application source-code. This results in bloated and difficult to maintain software, because the orthogonal enforcement logic is tightly-coupled with the application logic. The high cost of maintaining complex software, and the failure to follow good coding practices, have contributed to the emergence of implicit enforcement. In contrast to the traditional approach, implicit enforcement refers to a declarative way of injecting enforcement into an existing software application [52]. That means that no code has to be written in the application source-code to utilise enforcement, resulting in an implementation which is easier to maintain and to support. Enforcement can be declared in an REL. Before the software application is then delivered to customers the enforcement is embedded in the application distribution using a particular instance of the REL. The crucial advantage is that different enforcement implementations can be easily exchanged without modifying the original application source-code.

2.2.2 ODRL

The Open Digital Rights Language (ODRL) is an open REL standard [24] that provides a semantic description of rights over both tangible and intangible digital assets. It provides a language and vocabulary specified in XML Schemas [37] as a basis to express the relationship between parties to certain assets with their associated rights. ODRL does not mandate any policies for implementing a DRMS, but provides a means to express them. The core set of semantics can be extended for third-party value added services. PARMA [55, 56, 52] takes advantage of the open design to extend ODRL to add rights models which are especially applicable for pervasive computing environments and account for the expression of rights for certain modules of software.

2.2.2.1 Information Architecture

The Information Architecture models the entities and their relationships in the overall ODRL framework. The ODRL schemas are separated into data dictionary elements and the expression language. The data dictionary describes the semantics of the ODRL elements and forms the basis for extensions to the language [52, 55, 56]. The expression language details the models for the ODRL framework to which elements from the data dictionary can be applied. The models are detailed in the remainder of this section.

The Foundation Model

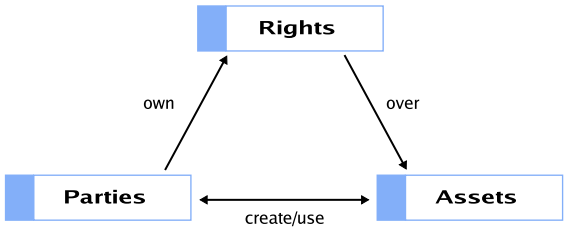

Figure 2.1: ODRL Core Entities

Figure 2.1 delineates the core entities as part of an ODRL's foundation model. It describes the rights under which a party allows another party to use an asset. Typically, the party who owns the copyright is called the rights-holder and the end user is the asset consumer. The rights entity consists of permissions, constraints, and obligations between the rights-holder and the consumer. An asset is an entity that describes any type of tangible or intangible work. The core entities must be uniquely identified. Identification can be accomplished with open standards, such as Digital Object Identifiers (DOI) [8] and Uniform Resource Identifiers (URI) [33] are well-suited for the identification of assets, whereas vCard [34] is the most well-known standard for describing the participating parties.

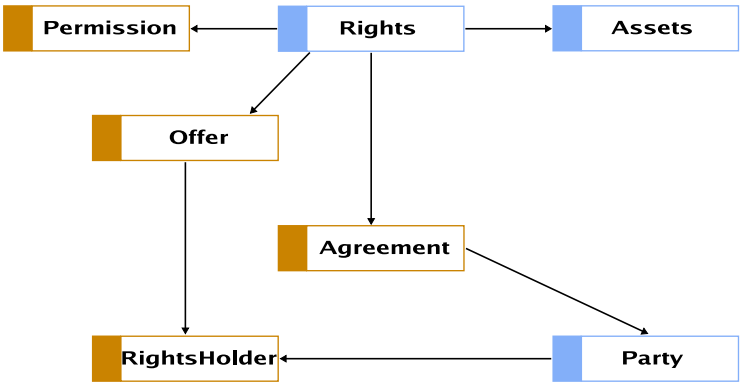

Figure 2.2: ODRL Foundation Model

Figure 2.2 shows the simplified foundation model from ODRL. The foundation model expresses offers and agreements where the offer is a proposal from a rights-holder specifying the rights over his/her

asset(s). An agreement is a specific instance of an offer where a party accepts the terms and conditions of a proposed offer. Another element, called a context element, is not shown in Figure 2.2, but it plays an important role by providing a mechanism to link entities together. For example an agreement can be linked to its original offer using unique IDs and references to these IDs.

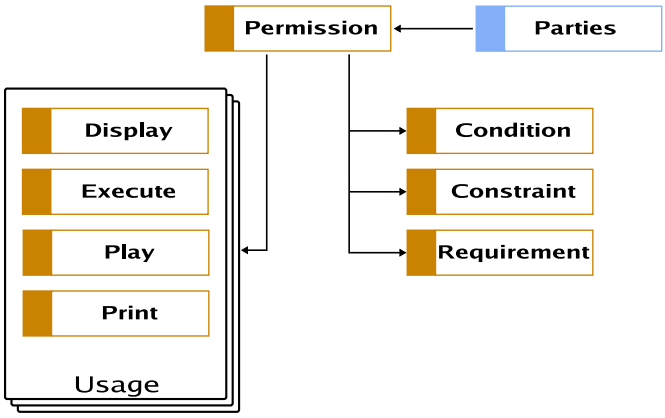

Figure 2.3: ODRL Permission Model

The permission model shown in Figure 2.3 can be used to further constrain offers and agreements. This model defines a set of activities allowed over the asset. The permission entity consists of conditions, constraints, requirements, and four abstract permission entities. Figure 2.3 details the four usage activities display, execute, play, and print to describe some usage scenarios for an asset. Further permission groups include reuse and asset management operations and transfer procedures. Requirements indicate obligations for the consumer in order to exercise the permission. An example of such an obligation would be a payment procedure that must be successfully completed before the associated permission can be granted.

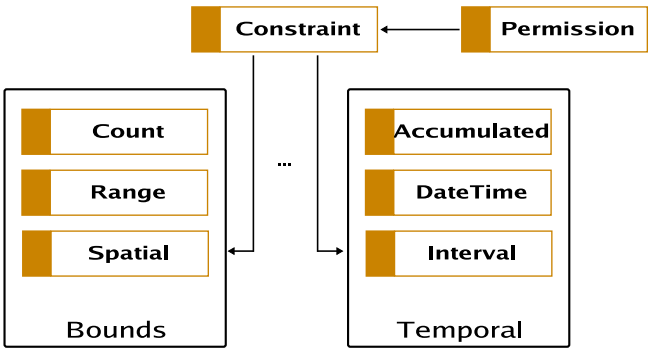

Figure 2.4: ODRL Constraint Model

Constraints realise a set of restrictions on the permissions. Figure 2.4 shows only two types of constraints, bounds and temporal, although there are seven types of constraints in total. Bound constraints limit usage to a fixed number or extent/coverage realised with the count, range, and spatial entities, e.g. the digital work can only be used x times. Temporal constraints limit usage to temporal boundaries

realised with the accumulated, date-time, and interval entities, e.g. the digital work can only be used in the time interval $(x; y)$.

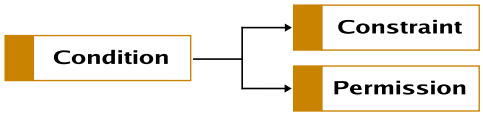

Figure 2.5: ODRL Condition Model

The condition model in Figure 2.5 expresses exceptions to the permission model which render permissions invalid, if an associated condition becomes true. For example, a permission is only valid, if the credit card has not expired. The expiry date of the credit card can be formulated as a condition that negates the permission, if it becomes true.

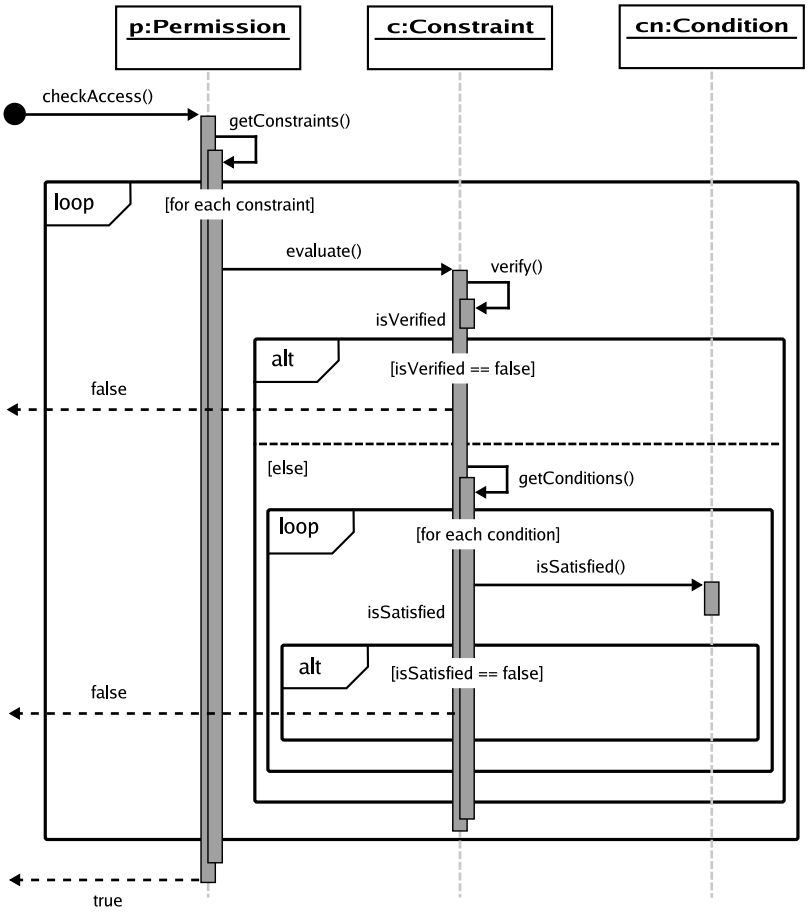

Figure 2.6: Sequence Diagram of Checking ODRL Permissions

The sequence diagram in Figure 2.6 outlines the abstract message flow for an enforcement component for ODRL permissions. When a permission is checked all constraints limiting the permission are evaluated. If the constraint itself disallows access it returns false. Otherwise each condition is checked to determine if it has been satisfied. If any one of the conditions is not satisfied the permission is negated

and the user may have to update the rights.

2.2.3 PARMA REL

Traditionally, DRM is concerned with copyrighted content (music, video, etc) but recently rights expression languages, such as Open Mobile Alliance's (OMA) REL [23], have been used to define rights on mobile application software as well [104].

The PARMA architecture primarily manages application usage rights on mobile devices and is not concerned with usage rights on other types of mobile content. Existing DRM systems that support applications as a possible content type are limited and support only simplified models such as whether the user/device is permitted to run the application or forward it to another user/device [12]. PARMA's focus on application usage rights provides a more fine-grained specification of rights over how the application can be used, as well as integration with payment and other services.

PARMA supports an extensible set of rights models including traditional rights for unlimited usage, named user, time-limited, feature-based model, subscription-based, pay-per-use where payment can be in real-time or audited, node-locked, and concurrent usage rights models.

OMA's REL version 2.0 [23] is designed specifically for mobile devices and its schema consists of a subset of ODRL elements extended with elements specific for mobile environments. The OMA REL is the basis for DRM systems by both Nokia and Sony-Ericsson. PARMA REL, designed in part for mobile devices, reuses some of the permissions and constraints from the ODRL REL (execute, date-time, count, etc) but beside simple permissions and constraints PARMA REL requires information about parties involved in issuing rights, amount and means of payment for the application, as well as some constraint definitions for pervasive computing environments. For example, PARMA extends the data dictionary of ODRL to introduce audit-based and feature-based usage rights models to address pervasive computing environments which are introduced in the next section.

2.2.3.1 Pervasive Rights Models

ODRL has defined an extensive list of permissions on content, however, the only permission applicable to software application usage is "execute". All the constraints PARMA REL uses are defined as constraints on the execute permission.

Feature-Based Model

Listing 2.5: PARMA REL Feature-based Model

```
1 <o-ex:constraint>
2   <parma:pointcut>
3     <parma:adviceType>before</parma:adviceType>
4     <parma:package>
```

```
5         <parma:packageName>ie.tcd.parma</parma:packageName>
6         <parma:class>
7           <parma:className>MyClass</parma:className>
8           <parma:method>
9             <parma:methodName>myMethod</parma:methodName>
10            <parma:returnType>void</parma:returnType>
11            <parma:argType>java.lang.String</parma:argType>
12          </parma:method>
13        </parma:class>
14      </parma:package>
15    </parma:pointcut>
16 </o-ex:constraint>
```

The feature-based model supports certain features of application software to be enabled or disabled depending on the rights agreement. For example, the multi-player over Bluetooth option for a mobile game might be disabled for a demo version of the application. PARMA REL allows the specification of such features (shown in Listing 2.5) using a new `pointcut` element which inherits the type definition from the `constraintElement` of ODRL. Listing 2.5 details a feature which is associated with the method signature `ie.tcd.parma.MyClass#myMethod(java.lang.String)`. It declares that any other constraint of the execute permission (not shown in Listing 2.5), e.g., an expiry date, should be validated by the enforcement before the method is called.

Audit-Based Model

Listing 2.6: PARMA REL Audit-based Model

```
1 <o-ex:rights
2   xmlns:o-dd="http://odrl.net/1.1/ODRL-DD"
3   xmlns:parma="http://www.dsg.cs.tcd.ie/parma/1.2"
4   xmlns:o-ex="http://odrl.net/1.1/ODRL-EX"
5   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
6   xsi:schemaLocation="http://www.dsg.cs.tcd.ie/parma/1.2
7                       parma-12.xsd">
8
9   <o-ex:agreement>
10    <o-ex:asset>
11      ...
12    </o-ex:asset>
13    <o-ex:permission>
14      <o-dd:execute>
```

```
15          <o-ex:constraint>
16            <parma:audit id="auditID" />
17          </o-ex:constraint>
18          ...
19        </o-dd:execute>
20      </o-ex:permission>
21      <o-ex:requirement>
22        <parma:auditLogData dateTime="true" duration="false" accumulated="false"
              />
23      </o-ex:requirement>
24    </o-ex:agreement>
25 </o-ex:rights>
```

One of the features that distinguishes PARMA from other similar projects is the very flexible pay-per-use policy. It can accommodate real-time pay-per-use where users make payments immediately before or after running the application for a certain amount of time or a certain number of times. In the audited pay-per-use model, information on the application usage is stored ideally in a secure storage area of the mobile device and is occasionally transferred to the rights server. Billing information is produced based on the usage data from this log. Listing 2.6 specifies the required information in the audit log. It depicts a timestamp as the only requirement which should be associated with an entry in the log file. An entry is further detailed by an audit identification which is specified in the audit constraint element. A certain feature associated with this constraint is logged with the audit identification "auditID". Therefore, it is possible to specify a global layout for the audit log while being able to distinguish billing information for individual features.

2.3 DRM Enforcement Architectures

There are a number of academic research and industry projects which tackle problems concerning the enforcement of rights on applications. This section introduces four different approaches for different execution domains and compares their functionality with the requirements for a pervasive enforcement architecture as outlined in Section 1.1.3 of this thesis.

2.3.1 Filigrane

Filigrane [10] was an European-funded project with the aim of increasing the cost of piracy for mobile applications [65, 71]. This project analysed the cost involved in circumventing security measures employed into an application compared to legally buying the application software. This work extended a model of the economics of piracy developed by Devanbu and Stubblebine [53] that specifies the cost of circumventing the enforcement in Equation 2.1, stealing the work in Equation 2.2, and reverse engineering

enforcement in Equation 2.3:

$$n * C_b >> C_h + n * C_c + P_l(n) * C_l(n) \tag{2.1}$$

$$C_{rw} >> C_s + P_l(n) * C_l(n) \tag{2.2}$$

$$C_{rw} >> C_e + P_l(n) * C_l(n) \tag{2.3}$$

Equation 2.1 models how an individual can either buy the software application (cost C_b), or circumvent the protection mechanism (at cost C_h) and then make n copies with cost C_C each. The software pirate has to face the probability of getting caught (P_l) with a possibly large fine (C_l) both depending on the number of copies. Apart from the illegal copy itself, copyright can be breached by stealing shown in Equation 2.2 (with the associated cost C_s) or reverse engineering modeled in Equation 2.3 (with the associated cost C_e) a software module and incorporating it in an application, instead of re-writing it. Rewriting usually is associated with a relative high cost (C_{rw}), because man months have to be invested to accomplish a module from scratch with similar or the same functionality. Essentially these three equations state that the higher the cost of buying the software the higher the incentive to commit piracy. Given the current technologies, C_h is not very high. In fact, it involves little effort to decompile a software and remove the protection implementation. That is especially the case for platform-neutral byte-code such as .Net and Java. The free software movement reduced C_b to zero with the remarkable result of discouraging software piracy.

In order to take advantage of these equations for commercial application software the cost of pirating has to be as high as possible. Filigrane uses encryption, smart cards, watermarking, and code obfuscation to fight the potential threats of piracy.

The packaging process involves the following steps to employ the security measures implemented in Filigrane. The first step is watermarking which injects hidden information into the application with the intention to make the application traceable. This information can be based on the user who has the rights to use the application. Watermarking does not prevent piracy, but it increases the probability of getting caught (P_l). After that, the code is obfuscated. Obfuscation renders the application source-code non-readable without affecting the behavior of the application. Obfuscation, too, is not aimed at preventing piracy, but it increases the cost of stealing (C_s) and reverse-engineering (C_e) the application. The next step is to encrypt the application with a user-dependent encryption key. Encryption is the central security measure in their architecture to increase the cost of circumventing (hacking) (C_h), stealing (C_s), and reverse-engineering (C_e). Finally, the encrypted application is packaged with usage rules, the Filigrane Security Engine. The resulting package is signed by the provider to ensure integrity and delivered to the user. A separate package containing the decryption key and the values of the rules, for instance duration limit and number of allowed usage, is created for the smart-card.

For Filigrane applications to work the client requires an extended Java Runtime Environment (JRE) to be able to execute the application and a smart card reader with a smart card issued by a trusted Certificate Authority (CA). Downloading and installing the application is divided into two steps. First, the user downloads the application and installs it into the local file system. Second, a secure end-to-end connection is established between the Java smart card proxy and the Filigrane server to store the decryption key and the values of the rules on the smart card. The extended JRE takes care of the runtime decryption of the application. This strict security policy forbids illegal copy, illegal use, illegal reuse, modifications, and reverse engineering. However, this comes at a relative high cost for the average user. The user has to install the extension from Filigrane to the JRE and requires a smart card reader which is not yet widely available. Also, the user needs a smart card issued by a trusted CA.

2.3.2 A generic DRM Framework for J2ME

In contrast to Filigrane WIT-Software [94] implemented a generic DRM framework for J2ME applications. It provides a server-side solution to instrumenting MIDlets that injects a DRM solution into an application. This framework was designed with three main objectives:

- Generic: Support of different DRM solutions.
- Transparency to the application developer: The application developer does not need to learn an API to implement copyright protection mechanisms.
- Transparency to the user: The average user should not be aware of DRM mechanisms.

The DRM framework is implemented as a module in an Over-The-Air (OTA) provisioning server. A provisioning server enables centralized distribution of MIDlet suites through a web interface. The Java Community Process (JCP) [5] specified a Java Service Request (JSR) for such an architecture based on J2EE [15]. It allows developers to submit their applications in a client bundle and supports a client in discovering and delivering them. When a client requests a MIDlet suite the OTA module retrieves the MIDlet suite bundle from a database and injects the required DRM solution into the MIDlet suite. This package is obfuscated and pre-verified before it is delivered to the client. The instrumentation of MIDlet suites is based on a byte-code engineering [47].

2.3.3 xoRBAC

Finally, Guth et. al. implemented enforcement of access rights (xoRBAC) extracted from ODRL-based digital contracts [64]. This work introduced a Contract Schema (CoSa) which is a generic representation of contract information based on Rights Expression Languages (REL) that has support for, but is not limited to, ODRL. A digital contract consists of five contract objects:

- Subject: Subjects represent the parties involved in a contract, the rights-holder and the beneficiary. The rights-holder grants usage rights to the beneficiary.

- Resource: The resource or resources identify the asset which the usage rights are applied to.
- Permissions: A permission is a `<operation, resource>` tuple describing the usage rights over a resource. For example the tuple `<view, video>` represents the permission to view the video.
- Role: A role represents a set of permissions assigned to a number of subjects which are members of a certain role. This greatly simplifies the management of permissions.
- Constraint: A constraint defines an invariant for further restricting permissions, for instance a temporal constraint (e.g., an expiry date) or a bound constraint (e.g., count to limit the number of times an asset/resource can be used).

The contract processing is triggered by an access request to a protected resource which is represented in a four valued vector `<subject, operation, object, contract-id>` where the contract-id is an URL. The contract engine is responsible for fetching the remote contract, if it is not already loaded into memory. It then delegates to the contract checker which verifies the contract signature, checks the status of the contract, and authenticates the beneficiary. If all checks are passed successfully, the REL interpreter parses the contract and creates an Document Object Model (DOM) [66] so that the access control component can validate the request based on the access control relevant contract information. In particular the access control service checks a set of constraints associated with a permission [102]. For example viewing a video may be constrained by an expiry date or a maximum usage count.

2.3.4 OMA DRM

The Open Mobile Alliance (OMA) [22] is an open standardization body dedicated to defining a DRM framework for mobile services. The membership includes all major players in the mobile industry covering all aspects of the mobile value chain, e.g., mobile operators, IT companies, wireless vendors, and content providers. The urgent need for a DRM standard for mobile networks and devices resulted in the development of a set of standards in a bottom-up approach. The aim for the first version of the OMA DRM standard was to concentrate on mobile-specific content, such as ring-tones and logos, but not rich media content, such as video or music. The inclusion of rich media content was targeted in a second version of OMA DRM which also covers the addition of Public Key Infrastructures (PKI) and encryption of content and permission keys [23]. So far, the major handset manufacturers, such as Nokia and Sony-Ericsson, implemented the first version of OMA DRM and will support the second version as soon as it reaches final release status.

The OMA DRM version 1.0 consists of three specifications, the general architecture, the rights expression language, and the DRM content format [87]. The rights expression language is a subset of ODRL. It only covers the consumption of rights, but not the management or distribution of them [80]. The content format is a package (DRM message) which contains the media object and optionally the rights associated with it. It supports metadata, e.g., to identify the original content type, a unique identifier, and encryption details.

Moreover, OMA DRM specifies three delivery methods, forward-lock, combined delivery, and separate delivery, to transmit a media object wrapped in a DRM message to the consuming device. Forward-lock is the most common mechanism used to protect content, and also the easiest mechanism to implement. The media object is wrapped in a DRM message without any rights associated with it. Therefore, a set of default rights are enforced on the client device to only render the content but disallow the further distribution of it. Combined delivery extends the forward-lock mechanism by adding the possibility for more fine-grained rights to the media object. The rights and the media object are packaged in the DRM message. Separate delivery employs symmetric encryption for the media object. Media content and the rights are delivered separately to the client device. Therefore, the content cannot be rendered without the decryption key that is included in the separately delivered rights object. Usually, the rights object is pushed to the client device with WAP Push over SMS. Consequently, the identity of the user can be verified and billed, because both WAP Push and SMS identify the recipient of the rights object by the mobile phone number.

2.3.5 Evaluation of Existing DRM Enforcement Architectures

	Filigrane	WIT-Software	xoRBAC	OMA DRM
Extensible	no	yes	yes	no
DRM Standard	no	-	yes	yes
Platform Independence	no	no	no	no
DRM Transparency	weak	strong	-	strong
Tool support	-	-	-	yes
Network Transparency	weak	weak	weak	strong
Mobile Networks	no	-	no	yes
Resource Constraint Devices	no	yes	no	yes
Enforcement	implicit	implicit	-	implicit
Functional Layers	3 - 5	(3) - 4/5	2/5	2-5
TCPS	strong	-	-	weak[a]
Fine-Grained Rights Access	no	yes	no	no
Audit-Based Rights Model	no	-	no	no

[a] Version 2.0 of OMA DRM adds enhanced security features, such as PKI support.

Table 2.1: Summary of DRM Enforcement Architectures

This section compares the functionality provided by existing enforcement architectures for application software. The encryption mechanism in Filigrane is their central and most important measure against piracy. However, the architecture of Java allows any encrypted class files to be decrypted with little effort [93]. As Java uses class-loaders [82] to load classes, even in a hierarchy of class-loaders, there is

only one method which inevitably is the interception point:
defineClass(String , byte [], int , int , ProtectionDomain). This method calls into the Java Virtual Machine (JVM) native code and is responsible for creating the class object based on the byte array representing the decrypted Java byte-code. As a result, this method is the single point of failure for Filigrane. A hacker can break the encryption scheme by re-implementing the system class-loader and passing it into the JVM with the -Xbootclasspath/p option. Or a less intrusive approach is by using the JVM Profiler Interface (JVMPI) [13] to listen for JVMPI_EVENT_CLASS_LOAD_HOOK events in a custom JVMPI agent. The resulting class files would still be watermarked and obfuscated, but the overall cost of hacking the application is reduced significantly. The TCPS is, however, strong with the aforementioned limitations. DRM is not transparent to the user. The user is required to install an extension to the JVM and also needs a smart-card plus reader. Filigrane is completely based on encryption mechanisms. It does not provide a pluggable architecture so that different DRM enforcement strategies can used instead of the one provided.

Compared to Filigrane, the user of WIT-Software's framework does not need to install additional packages to be able to execute an application. Therefore, it is less intrusive but also provides a less rigid protection scheme. A big advantage of this approach is its extensibility to allow custom instrumentation of MIDlet suites. Any DRM solution can be plugged into existing MIDlets. However, it does not address the specific characteristics of pervasive computing environments. Every time, the user would like to upgrade the rights, a new application has to be downloaded which has the usage rules injected into the application, because MIDlet suites can not be designed as loosely coupled components installed separately.

xoRBAC basically consists of two major components, the access control mechanism and the REL interpreter. The REL interpreter provides means for a generic representation of digital contracts described in an arbitrary REL. The resulting Contract Schema is then used as a basis for access control. While the strengths of this model are the independence of a REL and the access control mechanism, it is not feasible for pervasive computing environments, because the component that is responsible for the verification of a contract is bound to an HTTP server which forwards corresponding requests to the contract engine. Therefore, an on demand network connection is required to be able to determine whether the user is allowed to access a protected resource.

OMA DRM version 1.0 is implemented by all major handset manufacturers, such as Nokia and Sony-Ericsson which also provide tools to create DRM packages. It was particularly designed for mobile environments and resource-constrained devices and governs how content is used on the client device by the definition of a mobile profile that is a subset of ODRL. However, the OMA REL standard does not allow for any extensions, because it is enforced as part of the mobile device platform specific to the handset manufacturers, and is also covered by a patent held by ContentGuard, Intertrust, Matsushita, Philips, and Sony [57, 59]. For example Nokia implemented OMA DRM in their series 40/60/90 platforms with the result that one platform is more restrictive than the other one which causes interoperability

problems. Consequently, support for a broad DRM standard is still problematic, because DRM is based on proprietary implementations, and ContentGuard, Intertrust, Matsushita, Philips, and Sony have patents covering implementations of OMA DRM. However, the latest version 9.0 of Symbian OS [32, 106, 97] is underway which has added OMA DRM support [91], but will not be deployed in handsets before the end of 2005 [73]. Also, support for software application rights in OMA DRM version 1.0 and version 2.0 is missing and therefore does not allow fine-grained rights on a feature level.

From the preceding analysis of existing enforcement schemes, see Table 2.1, it can be seen that no existing system covers all the requirements for enforcing application rights in pervasive computing environments. Filigrane is not suitable because it imposes a high resource utilisation on the target platform due to strong cryptographic techniques. xoRBAC is not suitable because it is deployed as a remote service that verifies rights to protected works and thus requires an on-demand network connection. WIT-Software's approach is limited by the lack of standard-based pervasive rights models and an enforcement engine. OMA DRM does not provide enough functionality to allow fine-grained rights enforcement on applications.

2.4 Summary

This chapter presented an overview of container architectures, DRM and related work in the area of technical protection measures to prevent software application piracy.

Container architectures have become the accepted standard in developing enterprise applications. With the advent of component-oriented software development, container architectures are increasingly prominent for tackling complexity and extensibility of enterprise applications. They provide mechanisms to manage dependencies and the life-cycle among its registered components. Section 2.1 presented the key patterns, Hollywood Principle and Dependency Injection. It describes the advantages of container architectures to promote simple design in order to allow application software to evolve over time, and the limitations of existing container architectures for pervasive computing environments.

Section 2.2 introduced the area of DRM and provided a definition for DRM. The term DRM is usually referred to as a separation of two distinct layers, digital rights management and digital management of rights. An important part of this section is the introduction of the five-layered functional model of DRMS. These layers represent the legal framework, the expression of contract terms, payment, digital rights management, and the digital management of rights. The layer of digital management of rights can be further conceptually separated with the notion of implicit and explicit enforcement where implicit enforcement refers to a declarative way of embedding the enforcement into an existing application whereas explicit enforcement requires knowledge of enforcement APIs.

Also, an open REL standard called ODRL is presented. The information architecture of ODRL defines core entities as rights, parties, and assets and their relationship. The foundation model for ODRL is presented with special attention to the permission model because it forms the basis of the enforcement

framework. The PARMA REL is presented as an extension to ODRL to provide audit-based and feature-based rights models, designed specifically for applications in mobile environments.

Section 2.3 detailed four related works in the area of enforcement of rights. Filigrane employs a very strict DRM regime and introduces additional costs to the user. It does not implement a DRM standard, but rather it provides a cryptographic approach to protect applications. The user is required to extend his/her installed JVM and also has to request a smart-card with a private key stored on it in order to decrypt the application and execute it. However, the encryption mechanism can be circumvented with little effort and therefore reduces the usefulness of such strong protection measures.

WIT-Software implemented a DRM framework for J2ME applications. The DRM protection mechanism is embedded in the mobile application when the user downloads the application. This instrumentation module hooks into an OTA server and provides a generic interface so that different instrumentation mechanisms can be plugged in. It, therefore, provides a means to integrate any DRM enforcement framework which implements a specific verification interface.

xoRBAC is an access control implementation based on ODRL-based digital contracts. This project introduced a Contract Schema as a generic representation of RELs to keep their access control mechanism independent of a particular REL. xoRBAC, however, is not designed for pervasive computing platforms as it requires an on-demand network connection to verify the contract.

OMA DRM is a standard for DRM for mobile services with broad industry support. The first version of that standard is widely used by handset manufacturers, but implementations of the standard have interoperability problems. Also, feature level rights for software applications is not supported in the current version and in the proposed release candidate.

The next chapter outlines the design of the enforcement architecture based on the non-functional requirements described in Section 1.1.3.

Chapter 3

Design of the PARMA Enforcement Architecture

This chapter discusses the main design decision taken during the development of the enforcement framework. A set of requirements of the framework are described in the first section of this chapter. The second section deals with the enforcement architecture. The last section gives a summary of this chapter.

3.1 Design Requirements

The design requirements for the enforcement framework are divided into the following non-functional and functional ones. Section 1.1.3 formulated the requirements based on the analysis of pervasive computing environments and existing licensing application frameworks. The following list refines the requirements in more detail and aims at providing a guide for the design of the enforcement architecture.

3.1.1 Non-Functional Requirements

1. Platform independence, Extensibility and Adaptability
   In order to support multiple platforms and to allow an evolvable and extensible framework, it is a requirement to employ best practices for framework designs, such as container architectures. A container architecture has proved to be a powerful design for assembling applications and managing dependencies and the life-cycle of registered components. Since it is a requirement as well to be platform independent, the container architectures should provide mechanisms to instantiate concrete platform-dependent classes without re-writing the application.

2. Third-party hardware and software independence
   Many DRM systems require a broad consensus of hardware and software manufacturers. The framework should support enforcement in such a way that it can be incorporated into any application without demanding additional runtime support of third-parties.

3. Implicit enforcement

   The major drawback of existing rights enforcement solutions for application software is their inflexibility, because of the tight-coupling of enforcement code and application code. Implicit enforcement provides a non-intrusive way of embedding the enforcement component into an application. Therefore knowledge of the enforcement is not required by the software application developer. Enforcement logic is specified separately, using a declarative REL, and woven into the original application, where it can be interpreted by the enforcement framework.

4. Network failure transparency

   One of the main goals is to mask network failure to account for the characteristics of pervasive computing environments. Assumptions can not be made on network availability, because wireless connections can suddenly become unavailable. Sudden disconnection has to be dealt with in the communication layer of the enforcement in order to avoid a network failure to render the application useless.

5. User-level DRM enforcement transparency

   Enforcement should be transparent to compliant users of applications that are managed using the PARMA enforcement framework. The enforcement has to seamlessly integrate into the application and should not change the perception of the application dynamics, for instance due to a high runtime overhead.

3.1.2 Functional Requirements

1. Integrate with PARMA

   PARMA [55] defined the requirement of an audit-based rights model to account for pervasive computing environments. Therefore, the enforcement should implement these usage rights models and should be responsible for updating the audit log at certain intervals. The implementation of this requirement has implications on the design of the involved components, because platform-specific characteristics have to be taken into account.

2. Access control for the execute permission

   Many of the pre-defined permissions in ODRL are not applicable for software applications. Therefore, it is a requirement to implement access control based on the execution permission and its associated constraints.

3. Separate delivery

   The mechanism of separate delivery allows a user to first download an application with attached evaluation rights and at a later state upgrade the rights without downloading a new application. Separate delivery should be implemented as a server process on the client device. It listens to certain incoming requests to process the storage of the rights file to the appropriate location.

3.2 The PARMA Enforcement Architecture

The following sections providen an overview of the enforcement framework and its components. First some use cases are introduced to cover enforcement from a functional model and then the overall architecture is presented with the help of logical, object, and dynamic models. Subsequent sections deal with the details of the design of each component.

3.2.1 Functional Model

The following two use case diagrams introduce the functional model from two perspectives, the developer and the user perspective.

3.2.1.1 Developer Perspective

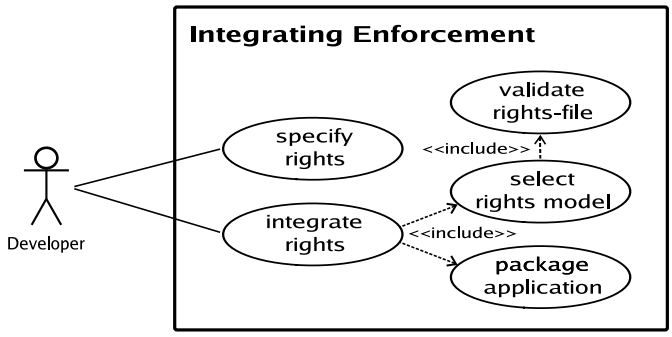

Figure 3.1: Use Case Diagram of Integrating Enforcement

Specify Rights

The developer specifies the rights for his/her application. Especially, the pointcuts have to be specified where the enforcement join points are going to be woven into the application. In essence the developer will be assisted by a tool that does not require the any knowledge of the PARMA REL.

Integrate Rights

The integration of rights has to be accomplished by the developer. Based on the overall rights description aspects are generated and woven into the application. A development tool could assist the developer in achieving this task. This task can be fully automated and should not require the developer to perform any manual tasks.

Select Rights Model

The developer selects the rights that should be attached to the application, so that any user can download and evaluate the application without paying any license fees. The selection of a default evaluation rights

model can be implemented as a configuration option in the goal of the build system. To ensure that the rights have the proper format, the rights must be validated.

Validate Rights-file

The rights file specified by the developer is validated against the PARMA REL Schema. This task can be accomplished with the help of XML Schema validators or XML editors when the file is created.

Package Application

Finally, the enforcement architecture embedded in the application has to be packaged with the default rights in a distributable format. For mobile platforms the distributable must be preverified. The classes can also be obfuscated and signed to ensure the integrity of the application software. A goal for the build system can be provided that automatically packages the application with the enforcement engine and hooks into the integration of rights use case.

3.2.1.2 User Perspective

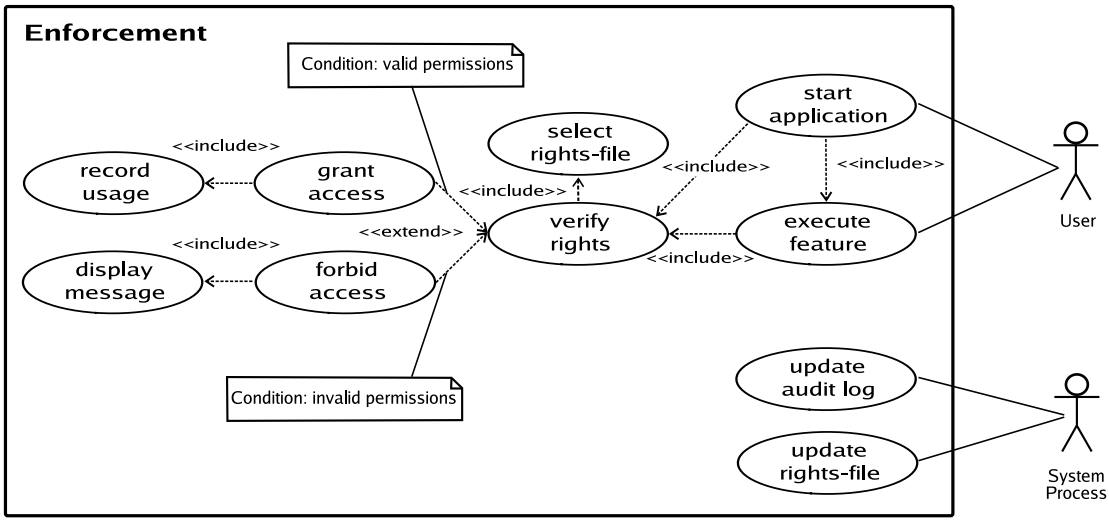

Figure 3.2: Use Case Diagram of the Enforcement

Start Application

The user downloaded and installed the application software either with the default evaluation rights, or with a purchased rights file which covers more advanced features of the application. The user starts the application. The start-up procedure could involve executing a feature declared in the rights file. In this case, the enforcement intercepts the execution and verifies the rights.

Execute Feature

The user executes a feature of an application. For example, this could be the option of changing the level or enabling a multiplayer option for a game. If the feature is declared in the rights file, the enforcement architecture will intercept this call to verify the rights.

Verify Rights

The verification of rights takes place at the join points which have been declared in the rights file. When this procedure is executed for the first time, the enforcement architecture has to be set up into memory in order to speed up the verification process for subsequent calls.

Select Rights File

The enforcement architecture has to select the correct rights file. If the user updated his/her rights agreement to cover more advanced features than specified in the default rights, then this updated rights file has to be verified. Otherwise the default file is selected to base the verification on.

Grant Access

This use case extends "Verify Rights". If the permissions specified in the rights file are valid, access is granted to the user.

Record Usage

If access has been granted to the user and if the rights file declares an audit-based rights model, the access usage is metered.

Forbid Access

This use case extends "Verify Rights". If the permissions specified in the rights file are invalid, access is denied to the user.

Display Message

If access has been denied to the user, the user has to be notified. This notification message could include a mechanism to initiate payment and hence update the rights file, so that the user can continue with his/her application.

Update Audit Log

If the rights file is associated with an audit-based rights model then a system process configures a server application which periodically submits the audit log to the PARMA rights server. The PARMA rights server exposes WSDL interfaces which should be used to upload audit data. If the connection fails,

because the network becomes unavailable or the user chooses not to submit at this time, the system process should schedule the next attempt.

Update Rights File

Once the user has registered with the PARMA rights server and requested a rights update, a message is sent to the client device, which carries information on where to download the new version of the file. This file is then downloaded and stored on the device at a well-known location so that the enforcement architecture can select it for subsequent enforcements.

3.2.2 Overview of the Enforcement Architecture

The following sections present an overview of the enforcement architecture by illustrating a logical, object, and a dynamic model representation.

3.2.2.1 Logical Model

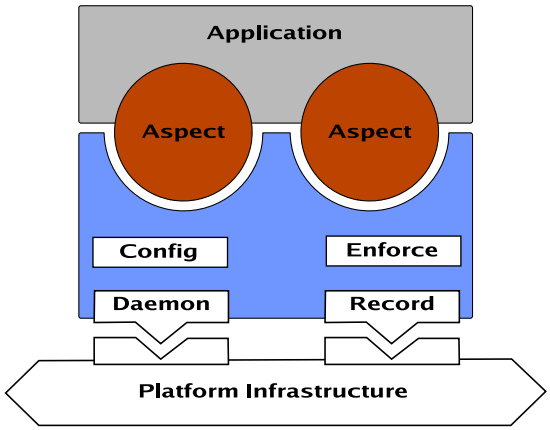

Figure 3.3: Logical Container Overview

This section presents a logical overview of the enforcement architecture as shown in Figure 3.3. The enforcement architecture leverages aspect-oriented software development techniques to compose the orthogonal enforcement service with software applications which otherwise leads to tangled logic and thus complex structures. The aspect shown in Figure 3.3 weaves any software application with the enforcement architecture based on a rights file specified for the application. The enforcement architecture consists of a container which manages the life-cycle of its registered components (Enforce, Config, Daemon, and Record) and also acts as an abstraction layer to hide the platform-specific requirements. Each component provides a high level view by exposing its interface to other registered components and a platform specific implementation. This is illustrated in Figure 3.3 where the Daemon and the Record components expose high-level views while their implementation is platform-specific. Therefore, it is possible to implement

enforcement on a multitude of platforms without changing the black-box view on rights enforcement provided to other components in the architecture.

3.2.2.2 Object Model

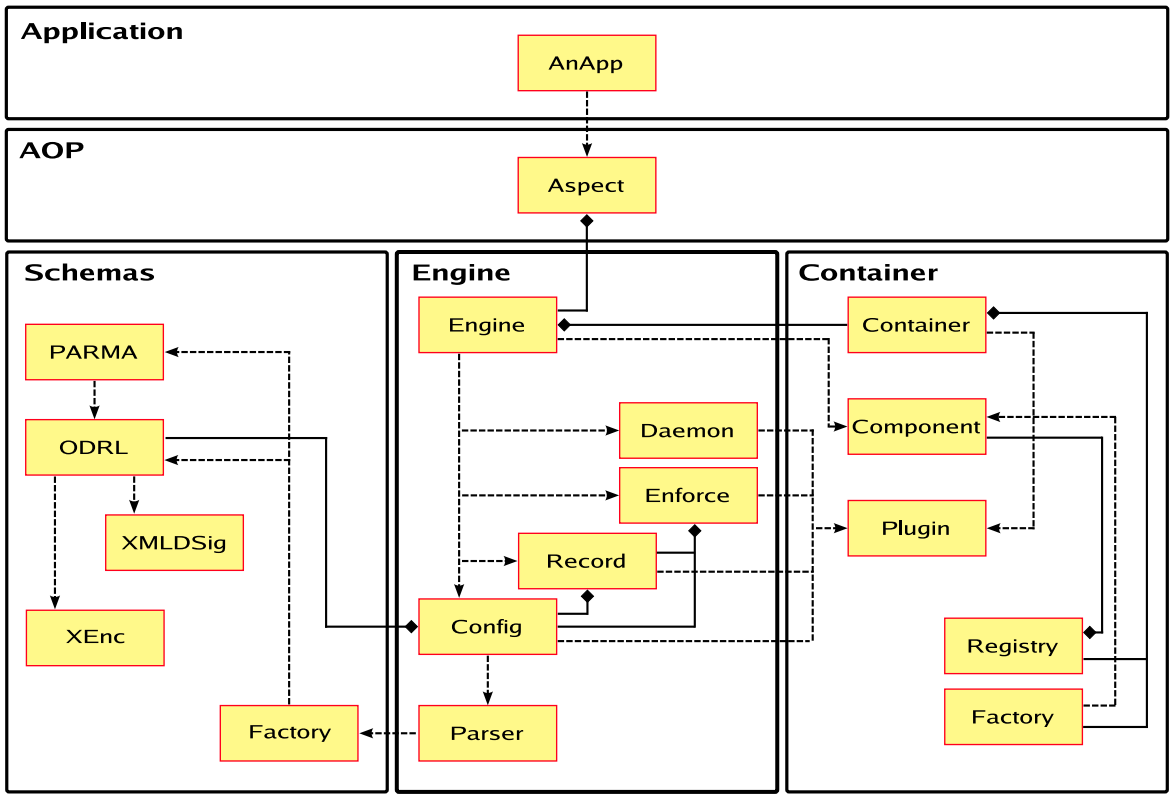

Figure 3.4: Architectural Overview

Figure 3.4 delineates the object model of the enforcement architecture. This model is divided into five domains: Application, AOP, Schemas, Engine, and Container. The application domain represents any application which is enabled to enforce rights associated with this application. The AOP realises the Aspect which defines the join points of the rights enforcement. The application is woven with the enforcement architecture based on the join point definitions in the Aspect. The enforcement architecture consists of the Schemas, the Container, and the Engine. The `Engine` registers its components (`Daemon` , `Enforce` , `Record` , and `Config`) with the `Container` , while the Schema corresponds to the in-memory representation of the REL. The Engine uses the Schema objects to validate application rights.

3.2.2.3 Dynamic Model

This section demonstrates the basic activities in the enforcement architecture shown in Figure 3.5. The first request to the enforcement architecture involves the initialisation of the enforcement services. The Engine registers all of its components with the Container and then starts the Container. The Container iterates through the components, instantiates each of them and resolves their dependencies. The Config

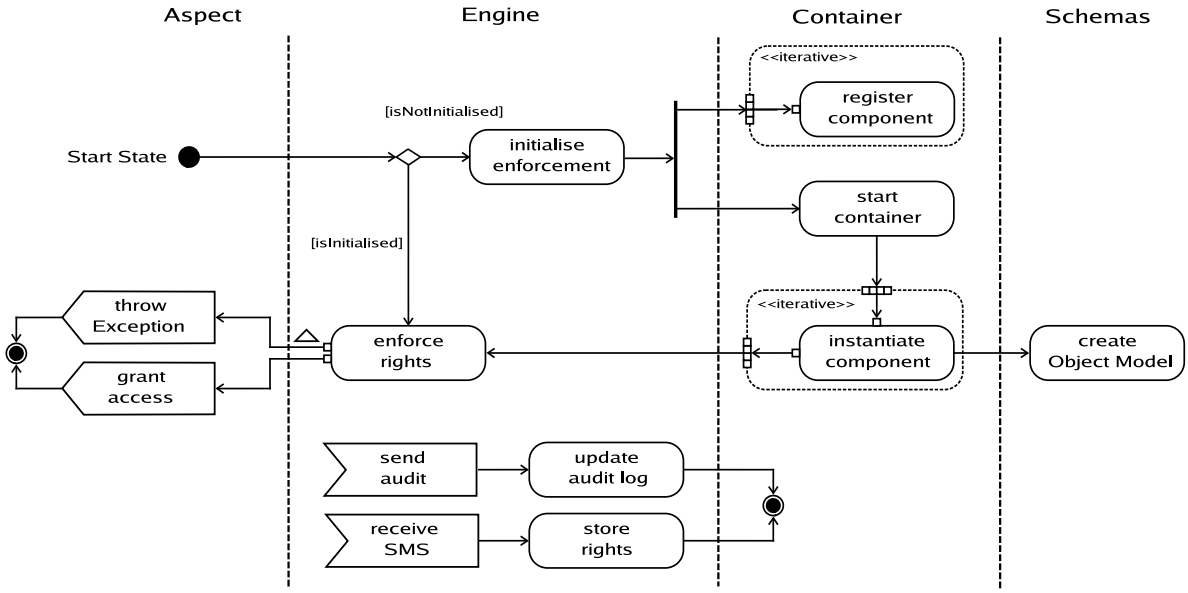

Figure 3.5: Activity Diagram of the Enforcement

component requires the deserialisation of the rights file into an object model. The Schemas module provides components to represent PARMA REL elements in Java objects. Once the enforcement is set up and configured, the rights are enforced. The enforcement request carries some additional context information about which feature requires enforcement. If the enforcement fails an exception is thrown, that can be caught by the aspect with the default handler displaying a pop-up window notifying the user of the rights enforcement failure. Otherwise the access is granted to the feature of the application.

3.2.3 Schemas

The Schemas components provides a Java object model based on the PARMA REL schema definition. The PARMA REL schema extends ODRL which in turn uses XMLDSig [38] and XEnc [35] schemas. XMLDSig specifies XML digital signature processing syntax to provide integrity for XML elements or a whole XML document and XEnc specifies a process for encrypting data and representing the result in XML. Although, PARMA does not currently use the security-related schemas, the Java object model covers all of them. So, it is theoretically possible to represent the full PARMA REL specification including all security relevant elements.

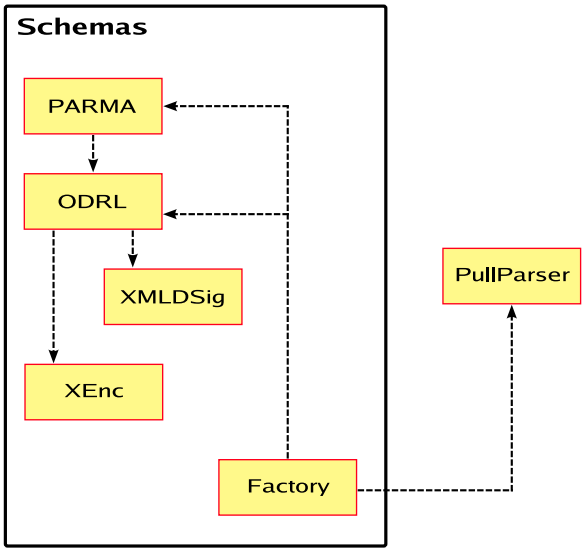

Figure 3.6: Schemas Overview

Figure 3.6 shows some basic collaboration of the Schemas components. There is one Factory component and four components that represent different elements of the REL. The Factory uses a pull parser approach to establish the object model. A pull parser gives full control over which elements of an XML file should be parsed and which ones should be left out. Consequently, the parser itself has a low overhead and the object model only covers the elements needed for enforcement, particularly the requirement and permission elements of the PARMA REL.

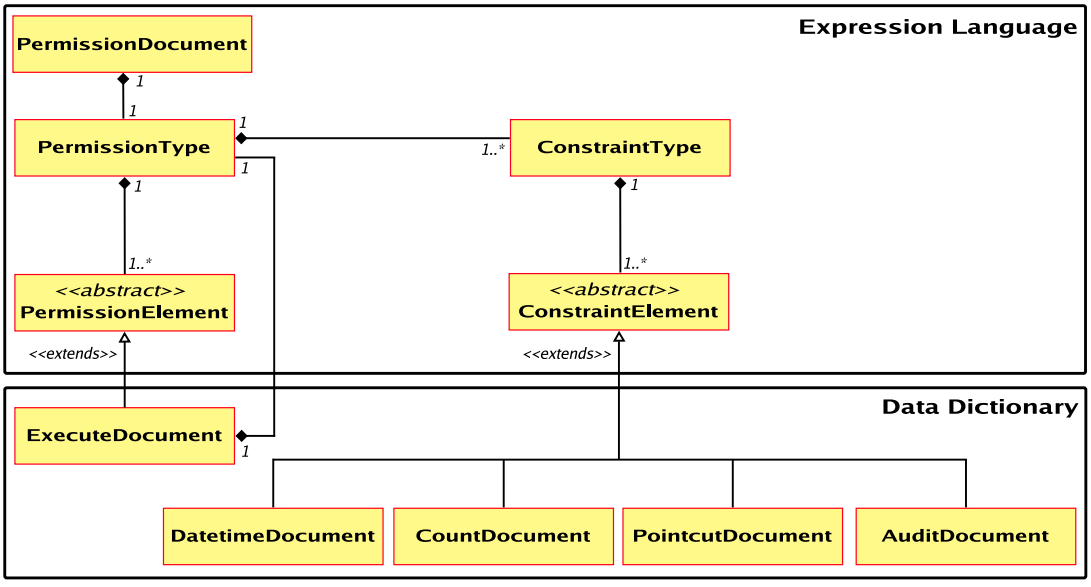

Figure 3.7: Class Diagram of the Schemas

Figure 3.7 outlines the class hierarchy for the `PermissionDocument` which is the root element for access control. As described in Section 2.2.2.1 ODRL consists of the data dictionary which describes

the basic elements and the expression language which realises the model describing abstract elements and their relationship. ODRL and consequently PARMA REL are defined in an XML Schema [37]. Each `Document` class represents an XML element, whereas `Type` classes refer to a complex type of the schema definition. Rather than implementing or extending `Type` classes, the Schemas object model uses association to indicate that a `Document` class is of a certain type. This is a result of limitations in Java design. It would be more convenient to use multiple inheritance. In XML Schema elements which are declared as a "substitutionGroup" of another element implement subtype polymorphic behavior, whereas the elements which are declared as a certain "type" implement the content model declared globally in the schema.

The data dictionary extends or inherits the abstract elements specified in the expression language. Figure 3.6 shows two abstract classes `PermissionElement` and `ConstraintElement` which are both declared in the expression language schema. `DatetimeDocument` , `CountDocument` , `PointcutDocument` , and `AuditDocument` extend `ConstraintElement` to implement a certain constraint where the `PointcutDocument` and the `AuditDocument` are defined in PARMA REL (Section 2.2.3). These elements are used to describe constraints on the permission associated with an application feature. `ExecuteDocument` realises the execute permission and extends the `PermissionElement` .

3.2.4 Container

Container architectures were introduced in Section 2.1. The container architecture for mobile devices has to be designed in such a way that it does not need to use the Reflection API to find out the dependencies of a class. The container architecture implements a variant of the setter-based dependency injection (Section 2.1.3).

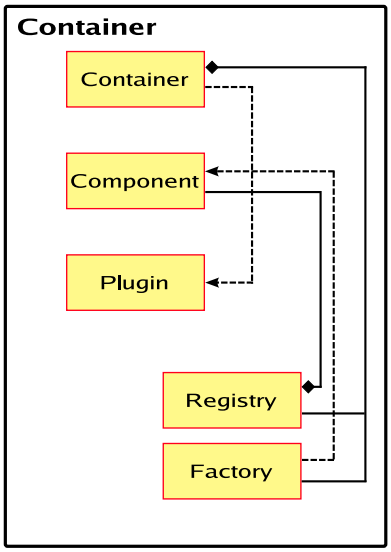

Figure 3.8: Container Overview

Figure 3.8 shows an overview of the Container. The Container is composed of the `Registry` and

the `Factory`. The `Registry` is a pool which contains all registered `Components` with the `Container`. The `Factory` implements the mechanism to instantiate `Components`, and is platform-specific.

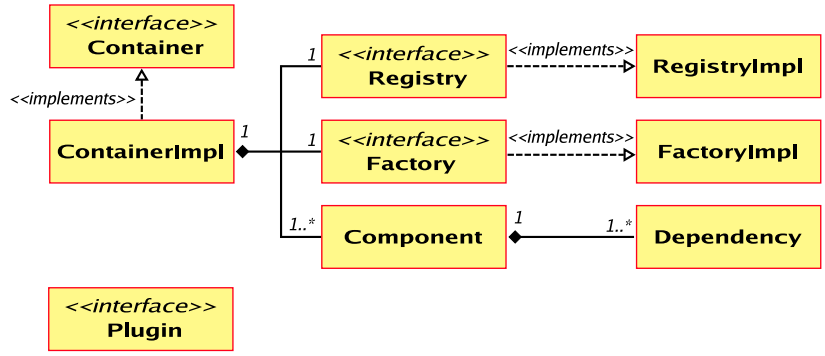

Figure 3.9: Class Diagram of the Container

Figure 3.9 is a more detailed diagram depicting the class hierarchy of the Container. The class `ContainerImpl` is an implementation of the `Container` interface which allows the construction of container hierarchies. An instantiation of the Container consists of at least one top-level container. This top-level container is the root element of a containment hierarchy and thereby scopes the visibility of components. A Container and its registered components provides access to the components registered in the parent container, but not vice-versa. Therefore, it is possible to address limitations with static singletons. A singleton, as it is suggested by Gamma et. al. [61], is a pattern to ensure that a class has only one instance and a global access point with a static get-operation. Because of its static nature and public availability, this pattern results in strong dependencies and therefore is problematic. Container hierarchies provide a solution to this problem, by sharing a component in a parent container with its children. The Container ensures that only one instance of this component is provided and therefore does not force the component to implement the singleton contract explicitly. As a consequence, it is still allowed to instantiate the same class outside the scope of the Container and so does not put restrictions on the usage of this component which would be in place otherwise.

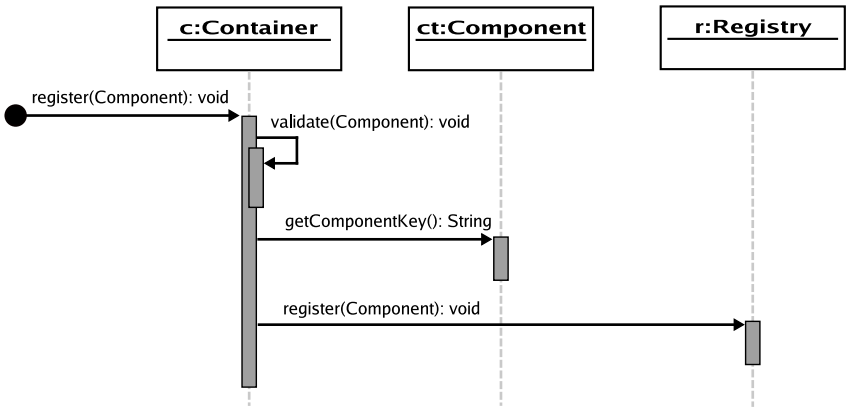

Figure 3.10: Sequence Diagram of Registering a Component with the Container

Figure 3.10 illustrates the sequence diagram of registering a component with the Container. The class that manages the Container registers the `Component` with the `Container`. Before the `Component` can be registered with the `Registry` the `Container` has to check whether this `Component` has already been registered. Also, the `Container` checks whether all the dependencies of the component are accessible.

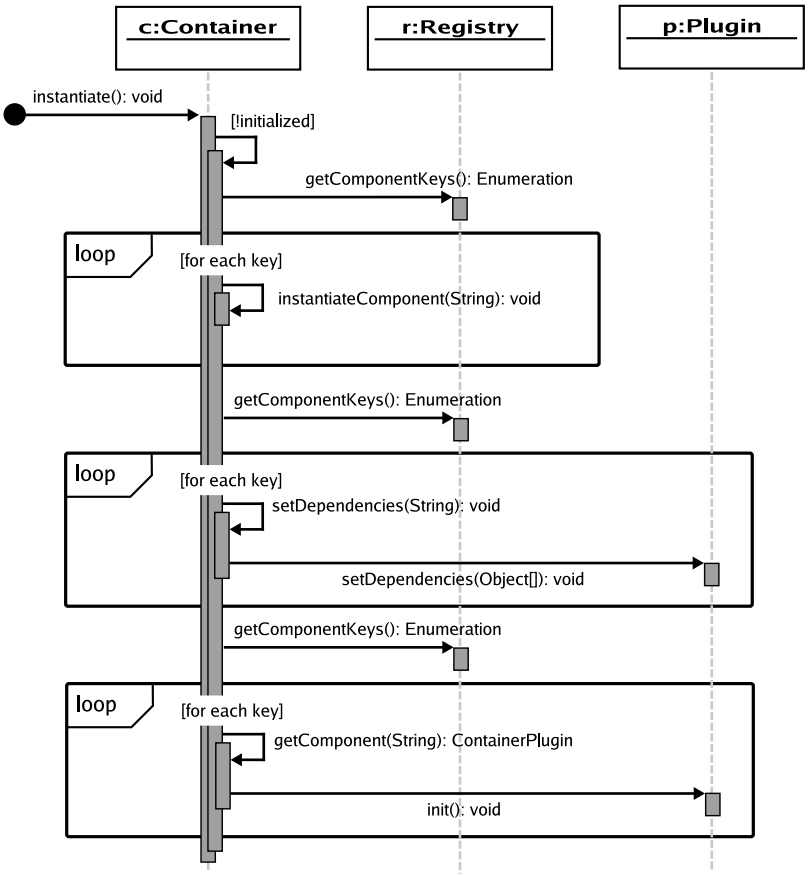

Figure 3.11: Sequence Diagram of Instantiating the Container

Figure 3.11 delineates how the Container is instantiated. The instantiation process covers three loops through the list of registered components. The first loop instantiates each component. The second one resolves the dependencies and injects them into the component via the `setDependencies(Object[])` callback declared in the `Plugin` interface and implemented by the component. The last loop initialises each component by calling the `init()` callback of the `Plugin` interface.

3.2.5 Engine

The enforcement engine is the heart of this framework. It provides components which implement the enforcement of rights according to the PARMA REL.

Figure 3.12 shows an overview of the Engine. It consists of the `Engine` component which is implemented as a singleton and registers the `Daemon`, `Enforce`, `Record`, and `Config` components

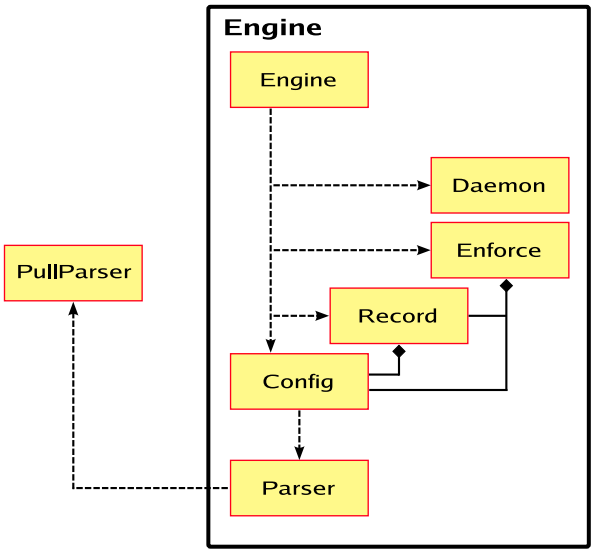

Figure 3.12: Enforcement Engine Overview

with the Container. The Engine is a singleton that implements lazy instantiation, which means that the enforcement container is set up when the Engine is first called (see Figure 3.5).

The `Config` object uses the `Parser` to create the object model representing the rights file. The root element of the rights file is exposed via the `Config` interface to interested components.

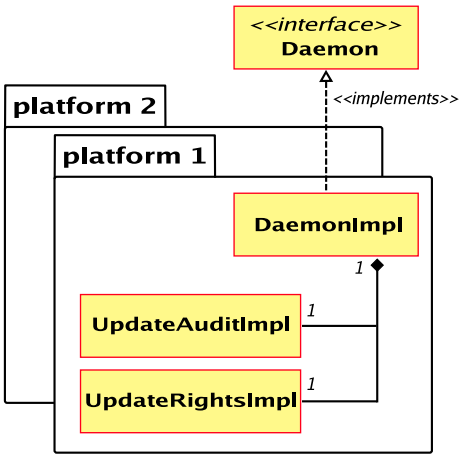

Figure 3.13: Daemon Class Diagram

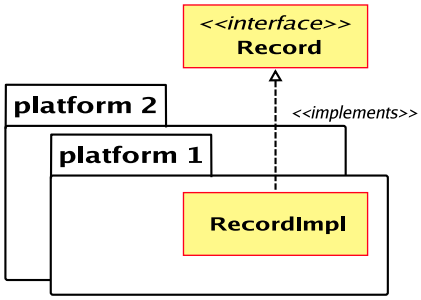

Figure 3.14: Record Class Diagram

Figure 3.13 and Figure 3.14 show overviews of the class diagrams of the `Daemon` and `Record` components respectively. Both components have platform-specific implementations which can be adapted to work on different platforms by using the Container.

The daemon service provides the functionality to update the audit log at certain time intervals and to accept a push request to update the rights file. These services are not initiated by the user and therefore act as servers on the client device. The record service is responsible for storing enforcement data on the client device.

3.2.5.1 Enforcement

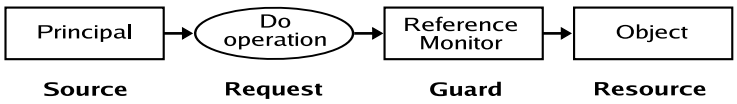

Figure 3.15: Traditional Access Control Model

The enforcement component is closely related to traditional access control mechanisms. Figure 3.15 details the elements of this model [76] where the principal represents the source of the request, the request to perform an operation on an object, the reference monitor acts as a guard to examine the request for an object, and the object or the resource as the destination of the request. The principal or the user owns rights to perform operations on an object or features of an application. This request is verified by the enforcement engine to grant or prohibit access depending on the validity of the rights. Each request is associated with a set of permissions to indicate whether access for a particular user is granted or denied. Also, an enforcement request carries contextual information on the destination object, the operation name, and parameter types in order to match this particular request with permissions which are associated with it.

As mentioned in Section 2.2.2.1 a permission is restricted by constraints which have to be satisfied in order to grant access. A constraint can be represented in a `<constraint, context-function, context-attribute>` triple, where the constraint represents the type of the constraint, e.g., count or date, the context-function which returns the value to determine the validity of the context-attribute. For example context-attribute could represent the actual number of times a certain request has been executed and the context-function returns the maximum allowed value. The context-function obtains its return value from the rights file.

This enforcement engine evaluates four types of constraint triples and any combination thereof:

- `<count, max(), actual_value>`
- `<date, start(), date_now>`
- `<date, end(), date_now>`
- `<audit, -, timestamp>`

The audit constraint exhibits a special case. It is a marker constraint, which indicates to the enforcement engine to meter the access usage. Unless, there is an error in writing into the audit log, this constraint never fails. Therefore, it is possible to implement post-pay rights models by uploading the accumulated log file to the PARMA rights server.

Figure 3.16 depicts the class diagram of the enforcement engine. Similar to the other components, it exposes an interface `Enforcement` which is implemented by the class `EnforcementImpl` . The `EnforcementImpl` class delegates the enforcement request to the `Execute` class, if a matching permission

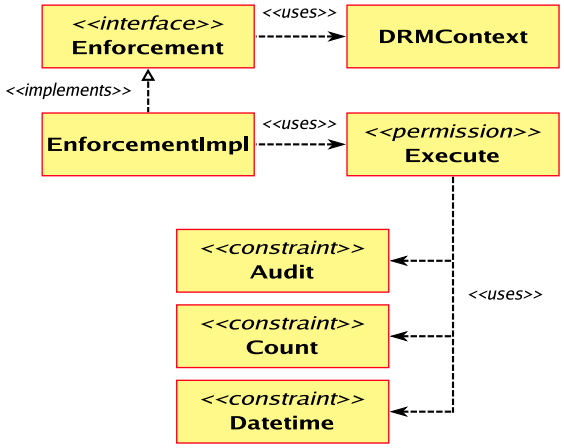

Figure 3.16: Enforcement Class Diagram

for the enforcement request, described by the `DRMContext` , is found. The `Execute` class then delegates to all associated constraints which all have to be satisfied in order to grant access to the feature.

3.2.6 Rights Aspect

Simple object-oriented software techniques that integrate components into applications, which provide orthogonal services, tend to complicate software designs and implementation in several ways. First, they lead to tight-coupling among components, and second, the code that implements the integration concerns is often scattered across and entangled with the functional logic of the application. This compromises modularity in ways that increase development time and maintenance costs.

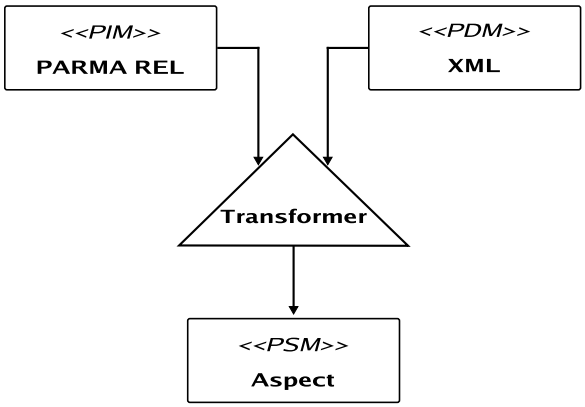

Figure 3.17: MDA Approach to Generating Aspects

The process of weaving the enforcement architecture into an existing application can be further improved by hiding the complexity involved in specifying the aspect and therefore provide a certain integration transparency to the developer. For this reason, this thesis has chosen an MDA-oriented approach as shown in Figure 3.17 to separate the specification of enforcement join points and their implementation into existing applications. The enforcement join points are specified using PARMA REL

(Section 2.2.3) which represents the platform-independent model (PIM). The platform description model (PDM) provides platform-specific descriptions that are in combination with the PIM transformed into the platform-specific model (PSM).

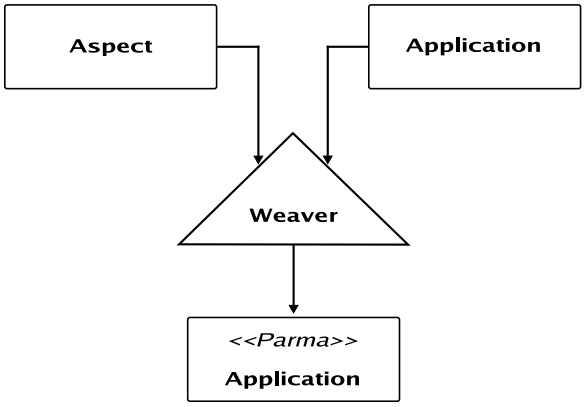

Figure 3.18: Weaving the Aspect into the Application

Figure 3.18 illustrates the weaving process. The aspect generated by the transformer shown in the previous figure is woven into the application. The weaver injects logic at the specified join points in the application, providing entry points to the enforcement engine.

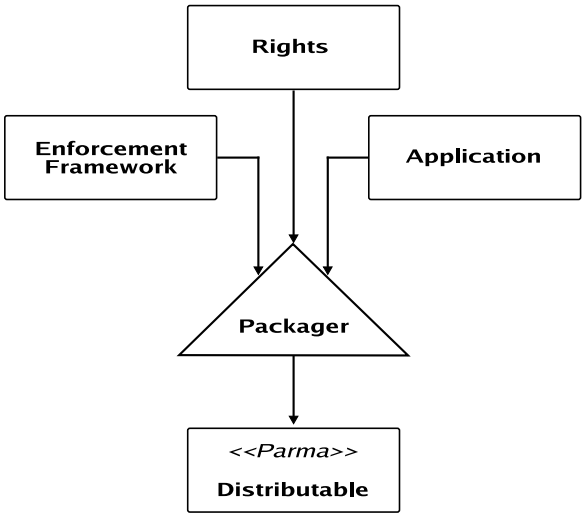

Figure 3.19: Packaging the Enforcement with the Application

The next step is to package the application with the enforcement framework and some rights (Figure 3.19). The result is a self-contained distributable with the enforcement engine co-located with the application. Consequently, the user does not need to install additional devices or extensions to the JVM in order to execute the application.

3.3 Design Summary

This chapter described the PARMA enforcement architecture's key requirements, its use cases and the overall architecture with a detailed view on each component.

The enforcement architecture has two main perspectives. The developer is concerned with a simple way of integrating the rights into his/her existing application as an orthogonal service without the requirement to learn a new set of APIs. The goal of the enforcement architecture is to provide a protection mechanism for the application software provider to prevent piracy and copyright infringements. The user is mainly interested in the application and not in the security measures it provides. From that stand point, the enforcement architecture should be seamlessly integrated into the application without disrupting the user's perception of the behavior of the application. These two considerations form the basis for the requirements outlined in Section 3.1. Also, the enforcement architecture has to support multiple platforms and especially account for pervasive computing environments.

Section 3.2 started with a high-level overview of the enforcement architecture by providing the functional model describing the use cases of both of the developer and user perspectives. Subsequent sections dealt with the specifics of the design of each component. The enforcement architecture basically consists of the Schemas, the Container, the Engine, and the Rights Aspect. With the help of the Schemas an object model representing the rights file can be established in memory (Section 3.2.3). The Container provides the main framework to build an extensible and adaptable enforcement architecture. It is designed to use the setter-based Dependency Injection strategy to resolve dependencies among components. The Container supports a basic configuration file to exchange the implementation classes of its registered components (Section 3.2.4). The enforcement engine is the heart of the enforcement architecture. A number of components are designed to provide a platform-independent implementation of the enforcement framework. The enforcement supports the enforcement of "execute" permissions and three constraints, audit, count, date-time, and pointcut. These constraints can be combined in different ways to restrict permissions on application features (Section 3.2.5). Section 3.2.6 introduced an MDA-oriented approach to integrate rights into an application in a non-intrusive fashion. An aspect-oriented technique is presented to weave the enforcement into an application based on the join points in the rights file.

The next Chapter presents the implementation of the PARMA Enforcement Architecture for both J2ME and J2SE environments.

Chapter 4

Implementation of the PARMA Enforcement Architecture

This chapter presents the implementation of the enforcement architecture. The implementation is based on the requirements set out in Section 3.1 and open-source principles. The result of this project is going to be contributed to the open-source community.

The source-code presented in this chapter does not contain any logging messages or error handling logic. As well, for sake of brevity, constants have been replaced by their actual value.

A number of Maven [86] plug-ins, which assisted the developer in certain tasks have been implemented as well. Some of them have been contributed to the open-source community already (see Appendix A).

The first section of this chapter details the implementation environment followed by specifics of certain implementation issues of the enforcement architecture. The chapter concludes with the implementation statistics and the summary of what has been achieved.

4.1 Java Packages

The implementation of the enforcement architecture covers the following packages:

- `ie.tcd.parma.container.core` - contains the container implementation.
- `ie.tcd.parma.drma.config` - contains the config component.
- `ie.tcd.parma.drma.context` - contains the context classes.
- `ie.tcd.parma.drma.core` - contains the core enforcement engine.
- `ie.tcd.parma.drma.enforce` - contains the enforcement implementation.
- `ie.tcd.parma.drma.record` - contains the implementations for the record component for both platforms, J2ME and J2SE.

- `ie.tcd.parma.drma.schema` - contains the JavaBeans representing the PARMA REL in an Java object model.
- `ie.tcd.parma.drma.update` - contains the classes for updating the rights and the audit log.
- `ie.tcd.parma.drma.xml` - contains a helper class to parse XML content with a pull parser.

4.2 Development Environment

The enforcement architecture has been developed based on MIDP 2.0, CLDC 1.1 and J2SE 1.4.2 (build 1.4.2_04-b05) with support for Linux and Windows platforms. To emulate and evaluate the enforcement architecture, the Wireless Toolkit version 2.1 from SUN has been used. Tests were implemented with JUnit [85] and J2MEUnit [98] and the test coverage was analysed using JCoverage [14, 67]. Several tools for quality assurance were adopted. Among these are JDepend [48], Findbugs [68], Simian [28], PMD [108], and Checkstyle [46].

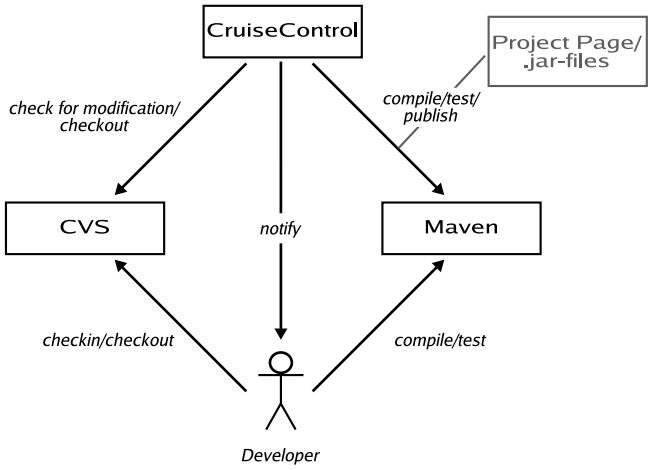

Figure 4.1: Development Environment and Continuous Integration

Figure 4.1 illustrates the development environment of this project. It follows open-source principles and best practices. The source-code for this thesis was maintained with CVS [6, 67]. The Maven [1, 20, 67] build system was used to manage the whole project with all its individual sub-projects and components. CruiseControl [7, 49] was set up to trigger automatic builds every hour and every night. It notifies the developers, involved in the last modifications, when a build fails and thereby allows for an early resolution of any problems. A nightly build of the project publishes the web-site and .jar-file artifacts.

AspectJ [3, 74], as an aspect-oriented technique, has been used to integrate the enforcement framework into an application. Throughout this chapter the use of the word aspect-oriented techniques, and aspects refer to the underlying AspectJ implementation.

4.3 PARMA Enforcement Architecture

4.3.1 Schemas

The Schemas component implements the object model of the schemas which are referenced in PARMA REL. The object model of the Schemas is represented by JavaBean classes which provide access methods to child or data elements. This allows the engine to browse elements and read the value of elements in a PARMA REL file. A JavaBean exposes a contract to access internal variables with a set naming scheme. Methods, such as `setName` and `getName` indicate that the JavaBean encapsulates an internal member called `name` . Based on the return type of the getter method or the parameter type of the setter method, the type of this member can be obtained.

Listing 4.1: Abstract Bean Class

```
1  public abstract class Bean {
2      private Hashtable table;
3
4      public Bean() {
5          table = new Hashtable();
6      }
7
8      protected final boolean containsKey(final String key) {
9          return table.containsKey(key);
10     }
11
12     protected final void set(final String key, final Object value) {
13         table.put(key, value);
14     }
15
16     protected final Object get(final String key) {
17         return table.get(key);
18     }
19
20     protected final void remove(final String key) {
21         table.remove(key);
22     }
23
24     protected final int size() {
25         return table.size();
26     }
27 }
```

All of the JavaBean classes that represent schema elements extend the abstract `Bean` class shown in Listing 4.1. `Bean` provides access to a `Hashtable` which consists of key-value pairs. The XML element name constitutes the key, whereas the value of a `Hashtable` entry is the value of the XML element. Sub-classes of `Bean` can use the methods `containsKey`, `get`, `set`, `remove`, and `size` to verify whether the `Hashtable` already contains a specified key-value pair, to retrieve a value, to set a new key-value pair, to remove a key-value pair, and to find out the size of the `Hashtable` respectively.

The following Listing 4.2 shows an example of a Schemas JavaBean:

Listing 4.2: PermissionDocument Class

```
1  public class PermissionDocument extends Bean {
2      public PermissionDocument() {
3          super();
4      }
5
6      public final PermissionType getPermission() {
7          return (PermissionType) super.get("permission");
8      }
9
10     public final void setPermission(final PermissionType permission) {
11         super.set("permission", permission);
12     }
13
14     public final PermissionType addNewPermission() {
15         PermissionType type = new PermissionType();
16         setPermission(type);
17         return type;
18     }
19 }
```

The class `PermissionDocument` in Listing 4.2 extends `Bean` and provides strongly typed access (`getPermission`, `setPermission`, and `addNewPermission`) to the "permission" element.

The JavaBean object model is created with the help of factory classes to separate the responsibility of data access and the construction of the object model. The factories use the pull parser API to obtain the specific information it needs, thereby reducing the overhead of the parsing procedure, because only elements the factory is interested in are pulled from the XML rights file. Other elements are ignored.

Listing 4.3: PermissionTypeFactory Class

```
1  public final class PermissionTypeFactory {
2      public static PermissionType parse(final XmlPullParser parser)
```

```
3           throws XmlPullParserException
4       {
5           PermissionType type = new PermissionType();
6           int event = 0;
7
8           while (true) {
9               event = parser.next();
10              if (event == XmlPullParser.START_TAG) {
11                  String tagname = parser.getName();
12
13                  if ("execute".equals(tagname)) {
14                      type.addPermissionElementArray(
15                          ExecuteDocumentFactory.parse(parser));
16                  }
17              } else if (event == XmlPullParser.END_TAG) {
18                  if ("permission".equals(parser.getName())) {
19                      break;
20                  }
21              }
22          }
23
24          return type;
25      }
26  }
```

The factory `PermissionDocumentFactory` class is responsible for creating the `PermissionDocument` object. The only element of the `PermissionDocument` is a `PermissionType` and therefore delegates to the appropriate `PermissionTypeFactory` factory class illustrated in Listing 4.3. `PermissionTypeFactory` has only knowledge of the "execute" permission and therefore adds objects representing the "execute" element to the permission element array. If the current position of the parser points to the end element of "permission", the factory leaves the `while` loop and returns to the calling method. As a result, the `PermissionType` object was created with all its "execute" elements as children.

4.3.2 Container

The Container is the IoC implementation with dependency injection discussed in Section 3.2.4. In contrast to many of the open-source containers, such as Picocontainer [26], the container presented here puts some restrictions on the design of components in order to register with the Container. Each component implementation has to implement the interface `ContainerPlugin` . This interface provides a setter method which accepts an `Object[]` to inject the dependencies into a component. When the

Container resolves the dependencies it calls back to this method and hands them into the component. Apart from this method there are also `init` and `shutdown` methods to initialise and cleanly shutdown the component respectively. This design restriction is mainly, because reflection cannot be used as a mechanism to find out how a dependency should be injected into a component.

Listing 4.4: Container.validateContainerPlugin() Method

```
1  private void validateContainerPlugin(final Component component)
2      throws ValidationException
3  {
4      Class clazz;
5
6      try {
7          String className = component.getClassName();
8
9          if (className == null) {
10             className = props.getProperty(component.getComponentKey());
11
12             if (className == null) {
13                 throw new ValidationException();
14             }
15             component.setClassName(className);
16         }
17
18         clazz = Class.forName(className);
19
20         if (!ContainerPlugin.class.isAssignableFrom(clazz)) {
21             throw new ValidationException();
22         }
23     } catch (ClassNotFoundException cnfe) {
24         throw new ValidationException();
25     }
26 }
```

When a component is registered with the Container, the Container has to check that the component is valid and obeys the design the Container imposes onto a component (Figure 3.10). Listing 4.4 illustrates the first half of the verification process. The aim of the method `validateContainerPlugin` is to determine whether the component implementation class implements the `ContainerPlugin` interface. If the implementation class of the component was declared when the component was registered then the component supplies its fully qualified string representation (line seven) and tries to instantiate it (line

18). Otherwise, the Container reads the fully qualified class name identifying the concrete component class from a properties file (line ten). In both cases, it determines whether the `ContainerPlugin` class is a super-interface of the component instance (line 20). If it does not fulfill the requirement or if the fully qualified class name cannot be instantiated, the Container aborts with the registration (lines 21/24).

Listing 4.5: Container.validateComponentInterface() Method

```
1  private void validateComponentInterface(final Component component)
2      throws ValidationException
3  {
4      Class clazz;
5      Class interfaceOfClazz;
6
7      try {
8          clazz = Class.forName(component.getClassName());
9          interfaceOfClazz = Class.forName(component.getComponentKey());
10
11         if (!interfaceOfClazz.isAssignable(clazz)) {
12             throw new ValidationException();
13         }
14     } catch (ClassNotFoundException cnfe) {
15         throw new ValidationException();
16     }
17 }
```

The second verification process checks the internal consistency of the component (Listing 4.5). That means, it verifies whether the concrete component implementation implements the given component interface (line eleven).

Both validation methods have to succeed in order to register the component with the `Registry`. Once all components are registered the Container can be advised to instantiate the components as shown in Figure 3.11.

Listing 4.6: Container.setDependencies() Method

```
1  private void setDependencies(final String componentKey)
2      throws InstantiationException
3  {
4      Component component = registry.getComponent(componentKey);
5      Dependency[] dependencies = component.getDependencies();
6      Object[] dependencyObjects = new Object[dependencies.length];
7
```

```
8       for (int i = 0; i < dependencies.length; i++) {
9           String dependencyKey = dependencies[i].getComponentKey();
10          Component dependency = registry.getComponent(dependencyKey);
11
12          if (dependency == null) {
13              ContainerPlugin plugin = parent.getComponent(dependencyKey);
14              if (plugin == null) {
15                  throw new InstantiationException();
16              }
17              dependencyObjects[i] = plugin;
18          } else {
19              dependencyObjects[i] = dependency.getComponentInstance();
20          }
21      }
22
23      try {
24          component.getComponentInstance().setDependencies(dependencyObjects);
25      } catch (InitializationException ie) {
26          throw new InstantiationException();
27      }
28  }
```

Listing 4.6 depicts the `setDependencies` method to resolve the dependencies of a particular component and inject them into it. The parameter of the method identifies the component interface. Based on this interface, the component is retrieved from the registry (line four). For each dependency, the components representing these dependencies are obtained from the registry (line ten). If the component cannot be resolved in the current Container, the parent container is looked up (line 13). The Container fails to instantiate the component if one of the dependencies cannot be resolved (lines 14/15). Finally, the dependencies is injected into the component (line 24).

4.3.3 Engine

The Engine represents the core of the enforcement architecture (Section 3.2.5). It is responsible for setting up the Container and delegating enforcement calls to the container service implementing it.

Listing 4.7: Engine.newInstance() Method

```
1  public static synchronized DRMEngine newInstance()
2      throws DRMException
3  {
4      if (instance == null) {
```

```
5           instance = new DRMEngine();
6           configure();
7       }
8
9       return instance;
10  }
```

Listing 4.7 depicts the singleton implementation of the `DRMEngine`. The purpose of the singleton is to control object creation, limiting the number to one. The `newInstance` method implements lazy instantiation and it guarantees synchronised access to it. The static `instance` field is initialised when it is first used followed by the configuration which includes setting up the Container.

Listing 4.8: Engine.configure() Method

```
1  private static synchronized void configure()
2      throws InstantiationException, RegistrationException
3  {
4      Component config = new Component(
5          "ie.tcd.parma.drma.config.Configuration",
6          new Dependency[] {
7              new Dependency("ie.tcd.parma.drma.record.Recorder")
8          });
9
10     rootContainer = new ContainerImpl();
11     rootContainer.registerComponent(config);
12     rootContainer.registerComponent("ie.tcd.parma.drma.record.Recorder");
13
14     Component enforcement = new Component(
15         "ie.tcd.parma.drma.enforce.Enforcement",
16         new Dependency[] {
17             new Dependency("ie.tcd.parma.drma.config.Configuration"),
18             new Dependency("ie.tcd.parma.drma.record.Recorder")
19         });
20     Component daemon = new Component(
21         "ie.tcd.parma.drma.core.services.DaemonServices",
22         new Dependency[] {
23             new Dependency("ie.tcd.parma.drma.config.Configuration"),
24             new Dependency("ie.tcd.parma.drma.record.Recorder")
25         });
26
27     enforcementContainer = new ContainerImpl(rootContainer);
```

```
28      enforcementContainer.registerComponent(enforcement);
29
30      servicesContainer = new ContainerImpl(rootContainer);
31      servicesContainer.registerComponent(daemon);
32
33      rootContainer.instantiateComponents();
34      enforcementContainer.instantiateComponents();
35      servicesContainer.instantiateComponents();
36  }
```

The configuration of the Container involves establishing the container hierarchy and registering the components with the appropriate container (see Listing 4.8). The Container is set up in such a way that the root container provides the Config and Record services to its children. The two child containers are responsible for the enforcement and hosting the Daemon services respectively. First the configuration component is created (lines four to eight) and registered with the root container (line eleven), as is the Record service (line twelve). Then the enforcement component is instantiated, which declares a dependency to the root container's configuration component (lines 14 - 19), and is registered with a dedicated enforcement container (line 28) in order to protect the visibility to it, as are the Daemon services (lines 20 - 25/30 - 31). Each container is instantiated along with the container components. The root container is instantiated first as it is responsible for providing services/components to the other two child containers. It should be noted that all components are registered with only their interfaces. It does not require any API calls to identify respective implementation classes. But rather a properties file is provided with the Engine, which maps the fully qualified interface name to its appropriate implementation class shown in the following Listing 4.9:

Listing 4.9: Container Properties

```
1 ie.tcd.parma.drma.config.Configuration=ie.tcd.parma.drma.config.
      ConfigurationImpl
2 ie.tcd.parma.drma.record.Recorder=ie.tcd.parma.drma.record.RecorderImpl
3 ie.tcd.parma.drma.enforce.Enforcement=ie.tcd.parma.drma.enforce.
      EnforcementImpl
4 ie.tcd.parma.drma.core.services.DaemonServices=ie.tcd.parma.drma.core.services
      .DaemonServicesImpl
```

Upon instantiation of the components, the Container pulls the required information from the configuration file and instantiates the concrete class which maps to a given interface. Thus, with the help of the configuration file, the Container can be adapted to multiple platforms by changing the mappings of the interfaces to the appropriate platform-specific implementation class.

4.3.3.1 Enforcement

The enforcement component registered with the Container realises the enforcement of "execute" rights and "datetime", "count", "audit", and "pointcut" constraints. The "pointcut" element constraints a particular permission in the rights file to a certain feature of an application by specifying the exact Java method signature this permission applies to. The aspect logic instantiates a context object that contains the current method signature and delegates the call to the enforcement architecture with the context information. The signature of the protected method in the context object is checked at runtime to determine whether it matches any permissions in the rights file.

Listing 4.10: Enforce Execute Permissions

```
1  public void enforce(DRMContext context) throws EnforcementException {
2      RightsType rightsType = rights.getRights();
3      OfferAgreeType[] agreeTypes = rightsType.getAgreementArray();
4
5      for (int i = 0; i < agreeTypes.length; i++) {
6          PermissionType[] permissionTypes = agreeTypes[i].getPermissionArray();
7
8          for (int j = 0; j < permissionTypes.length; j++) {
9              PermissionElementDocument[] elements
10                 = permissionTypes[j].getPermissionElementArray();
11
12             for (int k = 0; k < elements.length; k++) {
13                 if (elements[k] instanceof ExecuteDocument) {
14                     Execute execute
15                         = new Execute(recorder, rights.getRights(), context);
16                     execute.enforce((ExecuteDocument) elements[k]);
17                 }
18             }
19         }
20     }
21 }
```

Listing 4.10 illustrates the enforcement of all "execute" permissions of all the agreements as detailed in the rights file. The enforcement is then delegated to the `Execute` object which implements the enforcement for the "execute" permission. It evaluates whether the current context has a matching permission and if that is the case, obtains all associated constraints and determines whether these are satisfied (see Figure 2.6).

The nested for-loops have implications on the performance of the enforcement of rights. In common scenarios, the first and the second loops are flat, which means that the first loop just contains one

agreement type element and the second one contains a single permission type element. Also, the factories of the Schemas component instantiate only those elements which are of interest for the enforcement engine. Therefore, the third loop just consists of `ExecuteDocument` s and no additional overhead is created to loop through elements which are unknown to the enforcement.

As mentioned in Section 3.2.5.1, the audit constraint is a marker element to indicate that the enforcement requires the metering of access usage at this point.

Listing 4.11: Audit Constraint

```
1  private void audit(AuditDocument document) throws EnforcementException {
2      String id = document.getAudit().getId();
3      OfferAgreeType offer = rights.getAgreementArray(0);
4      RequirementType requirement = offer.getRequirementArray(0);
5      AuditLogDataDocument logData = getAuditLogData(requirement);
6
7      if (logData == null) {
8          throw new EnforcementException();
9      }
10
11     AuditLogDataDocument.AuditLogData audit = logData.getAuditLogData();
12     Calendar now = Calendar.getInstance();
13     int year = now.get(Calendar.YEAR);
14     int month = now.get(Calendar.MONTH) + 1;
15     int day = now.get(Calendar.DAY_OF_MONTH) + 1;
16     StringBuffer buf = new StringBuffer();
17
18     buf.append("<dateTime>");
19     buf.append(year);
20     buf.append("-");
21     buf.append(month);
22     buf.append("-");
23     buf.append(day);
24     buf.append("</dateTime>");
25
26     recorder.addRecord("AuditID:" + id, buf.toString().getBytes());
27 }
```

Listing 4.11 implements the constraint triple `<audit, - , timestamp>` where the timestamp has the format yyyy-mm-dd (lines 18 - 24). The timestamp, along with the ID of the audit constraint, is stored locally using the Record component (line 26). Each audit request adds a new entry in the audit log. After a certain configurable time interval the accumulated audit log is transmitted to the PARMA

rights server (see Section 4.3.3.2).

Listing 4.12: Count Constraint

```
1  private void enforceCount(Integer count)
2      throws EnforcementException
3  {
4      String recordStore = getPointcut();
5      byte[] actual = recorder.getFirstRecord(recordStore);
6      int current = 0;
7
8      if (actual != null) {
9          current = Integer.valueOf(new String(actual)).intValue();
10     }
11
12     if (count.intValue() <= current) {
13         throw new EnforcementException();
14     } else {
15         current++;
16         byte[] newCount = String.valueOf(current).getBytes();
17         recorder.setFirstRecord(recordStore, newCount);
18     }
19 }
```

The count constraint implements the constraint triple `<count, max(), actual_value>` as shown in Listing 4.12. Line four retrieves the signature of the current method invocation and retrieves the associated count value from the recorder (line five). The count value obtained from the Record component is parsed into an integer (lines eight - ten) and then compared to the value in the rights file (line twelve). If the user has exceeded the count value specified in the rights file, then the constraint is not satisfied and thereby prohibiting access (line 13). Otherwise the counter is incremented and stored (lines 15 - 17).

Listing 4.13: Date Constraint

```
1  private void enforceDate(DateType date) throws EnforcementException {
2      Calendar now = Calendar.getInstance();
3
4      if (date.isSetEnd()) {
5          if (date.getEnd().before(now)) {
6              throw new EnforcementException();
7          }
8      }
```

```
9       if (date.isSetStart()) {
10          if (now.before(date.getStart())) {
11              throw new EnforcementException();
12          }
13      }
14  }
```

The date constraint implements two constraint triples, `<date, end(), date_now>` and `<date, start(), date_now>` (Listing 4.13). The date_now value is retrieved with the Java API call `Calendar.getInstance()` which returns the current date in a `Calendar` object (line two). If the rights file specifies an expiry date, then it is determined whether the current date is before the expiry date (lines four/five). Likewise for the second triple `<date, start(), date_now>` (lines nine/ten). If either one of them are not satisfied, access is denied and returned to the user (lines six and eleven respectively).

The J2ME platform provides a `Calendar.getInstance()` class which is reduced in functionality and implements simplified methods. That means, that a J2SE `Calendar.getInstance()` class is not compatible with J2ME. However, the J2ME implementation of it can be used in a J2SE environment. The use of this class has some security-related implications, because date and time information is supplied by the operating system. Therefore, a user can circumvent the enforcement by resetting the date in the operating system.

4.3.3.2 Daemon Services

The Daemon component is responsible for two services, one which implements the transfer of the audit log to the PARMA rights management server and one which implements the separate delivery of rights. Both are implemented as `MIDlet` s in the MIDP instantiation of the framework and do not require user input. These `MIDlet` s are enabled to launch automatically upon an external event. The `PushRegistry` of MIDP 2.0 provides this mechanism to register `MIDlet` s for certain activation events, such as a timer- or network initiated events. Incoming messages can be stream-based (Listing 4.14), message-based (Listing 4.15), or datagram-based.

Figure 4.2 details the class diagram of the `Daemon` component. It is responsible to update the audit log at a configurable time interval to the PARMA rights server and to open a port to receive the separate delivery of rights. There are two different implementations, one for the J2ME and one for J2SE platform.

The J2ME implementation facilitates the `PushRegistry` to receive and act on asynchronous events, without user initiation. The `PushRegistry` manages network- and timer-initiated `MIDlet` activation. The `UpdateAudit` component is registered with the `PushRegistry` to update the audit log at certain time-intervals. In case of a network failure, the activation of the `UpdateAudit` component is re-scheduled instead of propagating the network failure up into the application level and thereby addressing the issue of network failure transparency. The `UpdateRights` component is registered with the `PushRegistry` to listen on a certain port of the mobile device. If an SMS is sent to this port, the `PushRegistry` reads

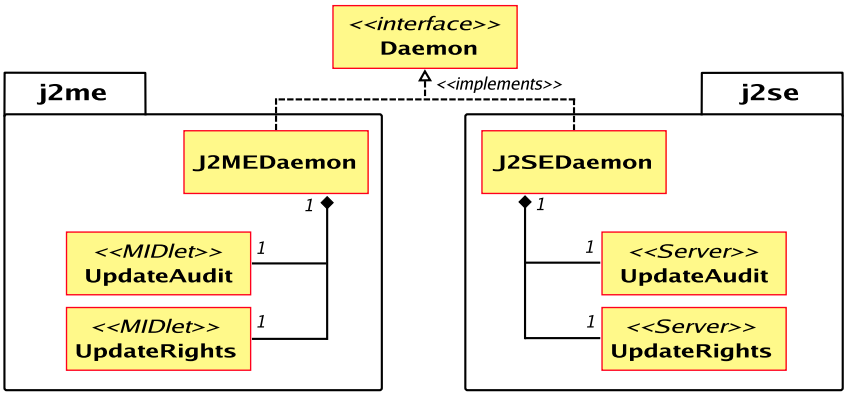

Figure 4.2: Daemon Class Diagram

the expected URL in the payload of the message and downloads the new rights file and stores it on the device.

The J2SE implementation works in a similar way, but rather than using the `PushRegistry`, which is not available on the J2SE platform, it uses TCP/IP sockets to perform the remote communication. While the `UpdateAudit` component uses a Timer API to trigger an update event at scheduled intervals, the `UpdateRights` component listens on a TCP/IP port to receive an update rights request.

Listing 4.14: UpdateAuditMidlet.startApp Method

```
1  public void startApp()
2  {
3      if (checkPermission("javax.microedition.io.Connector.http") == 0) {
4          return;
5      }
6
7      Vector envelopes = getAuditLog();
8      Enumeration enum = envelopes.elements();
9
10     if (enum.hasMoreElements()) {
11         while (enum.hasMoreElements()) {
12             SoapSerializationEnvelope envelope =
13                 new SoapSerializationEnvelope(SoapEnvelope.VER11);
14
15             envelope.bodyOut = (SoapObject) enum.nextElement();
16
17             HttpTransport ht = new HttpTransport(
18                 getAppProperty("AUDIT_SERVICE_ENDPOINT"));
19             ht.call(
20                 "rightsmgmt.vds_eds.parma.tcd.ie#sendAuditData",
```

```
21                  envelope);
22          }
23
24          deleteAuditLog();
25      }
26
27      Date alarm = new Date();
28      PushRegistry.registerAlarm(
29          this.getClass().getName(), alarm.getTime() + delta);
30
31      notifyDestroyed();
32  }
```

Listing 4.14 shows the `startApp` method of the `UpdateAuditMidlet` class. It is registered by the Daemon component with the `PushRegistry` to wake up upon a scheduled alarm. This alarm activates the `UpdateAuditMidlet` and executes the `startApp` method. The first statement in this method checks whether the `MIDlet` has sufficient permissions to initiate an HTTP request (line three). The method `getAuditLog` retrieves the audit log from the Record Management System wrapped in a SOAP envelope (line seven). This SOAP envelope conforms to the WSDL definition of the PARMA rights server in order to be able to transmit the audit log via a SOAP call. In line 17 a `HttpTransport` connection is established to the remote server and the SOAP envelope is transmitted (lines 19 - 21). If the transmission does not fail, the audit log on the client device is deleted to save persistent memory. Finally, the activation of this `MIDlet` is re-scheduled to a configurable time interval (lines 28/29) and the Application Management System (AMS) is notified that the `MIDlet` completed its task (line 31).

The configuration of the server endpoint and the time interval is in the Java Application Descriptor (JAD) shown below. This `MIDlet` uses dynamic push registration, using the `PushRegistry` API, explicitly to schedule the next activation time. In contrast the next example uses static push registration using the JAD file.

Listing 4.15: UpdateRightsMidlet.startApp Method

```
1  public void startApp()
2  {
3      String smsConnection = "sms://:" + smsPort;
4      String[] connections = PushRegistry.listConnections(true);
5      MessageConnection smsconn = null;
6
7      if ((connections != null) && (connections.length > 0)) {
8          if (smsconn == null) {
9              smsconn = (MessageConnection) Connector.open(smsConnection);
```

```
10              Message msg = smsconn.receive();
11
12              if ((msg != null) && (msg instanceof TextMessage)) {
13                  String payload = ((TextMessage) msg).getPayloadText();
14                  getViaHttpConnection(payload);
15                  save(buffer.toString());
16              }
17          }
18      }
19
20      PushRegistry.registerConnection(
21          smsConnection, this.getClass().getName(), "*");
22      notifyDestroyed();
23  }
```

Listing 4.15 illustrates how an incoming SMS is processed in order to update a PARMA rights file. When the `UpdateRightsMidlet` is installed on the client device, the AMS registers this `MIDlet` statically with the `PushRegistry` based on the configuration in the JAD descriptor. The first few lines check whether there are any incoming SMS connections on a configured port (lines 3/4). If an incoming connection is available (line seven), a connection to the message is established (line nine) and the payload of the text message read into a string (line 13). The payload of the message represents a URL to the new rights file. In the next step, a HTTP connection is set up to download the new rights file to the client device (line 14) and stores it at a well-known location (line 15).

Both `MIDlet` s use the JAD descriptor to read configuration values. The following Listing 4.16 lists the configuration:

Listing 4.16: JAD Descriptor

```
1 MIDlet-1: UpdateAuditMidlet, , ie.tcd.parma.drma.update.UpdateAuditMidlet
2 MIDlet-2: UpdateRightsMidlet, , ie.tcd.parma.drma.update.UpdateRightsMidlet
3 MicroEdition-Configuration: CLDC-1.1
4 MicroEdition-Profile: MIDP-2.0
5 MIDlet-Push-1: sms://:60009, ie.tcd.parma.drma.update.UpdateRightsMidlet, *
6 MIDlet-Permissions: javax.microedition.io.Connector.http, javax.microedition.
      io.Connector.PushRegistry, javax.wireless.messaging.sms.receive
7 allow: javax.microedition.io.Connector.PushRegistry, javax.microedition.io.
      Connector.http, javax.wireless.messaging.sms.receive
8 AUDIT_SERVICE_ENDPOINT: http://localhost:8080/axis/services/
      EDSRightsManagement?wsdl
9 AUDIT_UPDATE_INTERVAL: 2592000000
```

The first two lines outline the `MIDlet` s in the application, followed by the J2ME Profile and J2ME Configuration settings. The `MIDlet-Push-1` details the static registration of the `UpdateRightsMidlet` class. Here, it is configured to listen on port 60009 for incoming SMS messages. The `MIDlet-Permissions` identifies the specific permissions which are requested for the `MIDlet` suite. This descriptor requests permissions for the `PushRegistry` , `sms.receive` , and `http` . The last three lines are application specific properties for the PARMA rights server endpoint to submit the audit log, the alarm interval for the `UpdateAuditMidlet` , and the SMS port to listen for the separate delivery of rights.

4.3.3.3 Record Service

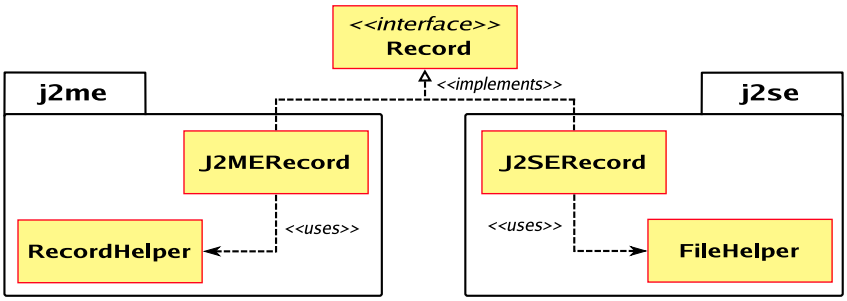

Figure 4.3: Record Class Diagram

Figure 4.3 details the class diagram of the `Record` component. To enable the support of multiple platforms, two platform-specific implementations are provided. The difference between both implementations is that MIDP does not have the notion of a file-system. But rather, MIDP provides a Record Management Systems (RMS) for on-device data persistence. The class `J2MERecord` implements access to this RMS, while `J2SERecord` uses the J2SE File IO APIs.

The use of a non-tamper resistant storage area for enforcement data, such as counter values, have a significant impact on the effectiveness of the enforcement architecture. In the J2ME environment MIDP applications have a private storage area which can be accessed with the RMS API. The enforcement architecture leverages the RMS to store sensitive enforcement information on the device. It does not, however, provide any data security which poses the potential risk of circumvention without the enforcement architecture recognising it. A hacker could perform a replay attack on parts of the persistent store to roll back to a previous state and consequently let the enforcement architecture believe that the data is correct. The enforcement of rights on J2SE environments poses the same risks. To overcome these problems a tamper-resistant storage area has to be employed to store the sensitive information in a secure and tamper-resistant way.

4.3.4 Rights Aspect

The rights aspect promotes loose-coupling between the original software application and the enforcement architecture. It facilitates aspect-oriented techniques to be able to weave the enforcement into an application and thereby composing the enforcement as an orthogonal service with the application.

Rather than introducing the XSLT stylesheets (Appendix B) that transform the PARMA REL into aspects that are woven in with the original application, this section concentrates on illustrating the aspect-oriented feature based on an example. The aspect was generated for a MIDP application, called the worm game, which comes as a demo with the wireless toolkit from SUN Microsystems.

Listing 4.17: Count-based Rights File for the Worm Game

```
1  <?xml version="1.0" encoding="UTF-8"?>
2
3  <o-ex:rights
4    xmlns:o-dd="http://odrl.net/1.1/ODRL-DD"
5    xmlns:parma="http://www.dsg.cs.tcd.ie/parma/1.2"
6    xmlns:o-ex="http://odrl.net/1.1/ODRL-EX"
7    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
8    xsi:schemaLocation="http://www.dsg.cs.tcd.ie/parma/1.2
9                        http://www.cs.tcd.ie/Dominik.Dahlem/parma-1.2.xsd">
10
11   <o-ex:agreement>
12     <o-ex:asset id="doi:1234567-9876">
13       <o-ex:context>
14         <o-dd:dLocation>URIofApplication</o-dd:dLocation>
15       </o-ex:context>
16     </o-ex:asset>
17     <o-ex:permission>
18       <o-dd:execute>
19         <o-ex:constraint>
20           <o-dd:count>2</o-dd:count>
21           <parma:pointcut>
22             <parma:adviceType>after</parma:adviceType>
23             <parma:package>
24               <parma:packageName>example.wormgame</parma:packageName>
25               <parma:class isMain="true">
26                 <parma:className>WormMain</parma:className>
27                 <parma:method>
28                   <parma:methodName>startApp</parma:methodName>
29                   <parma:returnType>void</parma:returnType>
```

```
30                    </parma:method>
31                  </parma:class>
32                </parma:package>
33              </parma:pointcut>
34            </o-ex:constraint>
35          </o-dd:execute>
36        </o-ex:permission>
37    </o-ex:agreement>
38 </o-ex:rights>
```

The rights file depicted in Listing 4.17 declares an execute permission with a count and a pointcut constraint. The pointcut element in the XML format specifies the signature of the method for which execution is protected (lines 21 - 33). In this example the method `startApp` of the class `WormMain` in package `example.wormgame` is constrained to a total usage count of two as dictated by the `o-dd:count` element (line 20). The `parma:adviceType` element determines when the enforcement is taking place (line 22). In this case, the execute permission is evaluated just after the `startApp` method has finished executing.

With the help of a platform-specific XSLT one concrete aspect class is generated. This process resembles the transformation from the PIM to the PSM in MDA (Figure 3.17).

The following Listing 4.18 details an aspect generated for the J2ME platform. J2ME puts some restrictions on the use of aspects. It is not possible to facilitate dynamic features of AspectJ to obtain runtime information using the Reflection API, e.g., target object, method signature, and therefore requires the aspect to account for this limitation. The XSLT stylesheet which implements the generation of the concrete aspect has to support a strongly typed model where no reflection is used. Also, it is important to distinguish between the enforcement of rights for standard MIDlet APIs, such as the `startApp` method, and APIs which implement a certain feature for the application. Therefore, the stylesheet generates a pointcut pair for each pointcut constraint specified in the rights file, one which implements the special case of enforcing rights for the `startApp` method of a `MIDlet` (Listing 4.19), and one which covers all other cases (Listing 4.18). That is, because the stylesheet has no means to find out whether the current pointcut applies to a `MIDlet` class or not and therefore leaves this decision to the aspect weaver.

Listing 4.18: Pointcut for non-MIDlet Methods

```
1 pointcut pcut_NoMIDletStartApp_startApp(example.wormgame.WormMain target):
2     call (void example.wormgame.WormMain.startApp())
3     && !call(void MIDlet+.startApp())
4     && !within(ie.tcd.parma..*)
5     && target(target)
```

```
6       && args();
7
8   after(example.wormgame.WormMain target):
9       pcut_NoMIDletStartApp_startApp(target)
10  {
11      DRMContext context = new DRMContext();
12      context.setPackage("example.wormgame");
13      context.setClazz("WormMain");
14      context.setMethod("startApp");
15      context.setParameters("");
16
17      try {
18          DRMEngine.newInstance().enforce(context);
19      } catch (DRMException drme) {
20          Alert alert = new Alert(
21              "PARMA Rights Enforcement",
22              drme.getMessage(),
23              null,
24              AlertType.WARNING);
25          alert.setTimeout(5000);
26          addAlert(alert);
27      }
28  }
```

Listing 4.18 illustrates the first pointcut of a pointcut pair for the `startApp` method of the worm game. The pointcut `pcut_NoMIDletStartApp_startApp` identifies all join points in the worm game application. These join points are calls to the method with the signature `WormMain.startApp` (line two) where the `startApp` is not overridden from the `MIDlet` class (line three) and whose calling object is not part of the `ie.tcd.parma` package (line four). The crosscutting enforcement behavior is implemented in an after advice (lines eight - 28). If the aspect weaver finds a matching join point, then the crosscutting behavior is woven into the application. The after advice with the enforcement-specific logic is then executed just after the method body of the join point and before control is returned to the caller. The context object is set up which identifies the method signature of the protected call (lines eleven - 15). This context is handed into the enforcement engine to verify the permissions for this particular method (line 18). If the enforcement fails an MIDP alert pop-up window is presented to the user to inform him/her about the particular failure. For example, the rights might have expired or the access usage has exceeded.

In the case of Listing 4.18, the `WormMain` class is a sub-class of `MIDlet` and therefore the crosscutting behavior is not woven into the application at the specified join points, because the call join point depicts

the location in the application which calls the `startApp` method. However, this method is called by the JVM and therefore the aspect weaver cannot weave the crosscutting behavior into the application. Also, the aim is to terminate the execution of the `MIDlet` when access to the `startApp` method is prohibited by the enforcement engine. Thus, the `startApp` method has to be handled differently to other methods. The aspect has to present an appropriate message to the user and prevent the further execution of the `MIDlet` .

Listing 4.19: Pointcut for MIDlet's startApp Method

```
1  pointcut pcut_MIDletStartApp_startApp(example.wormgame.WormMain target):
2      execution (void example.wormgame.WormMain.startApp())
3      && target(target)
4      && execution(void MIDlet+.startApp())
5      && !within(ie.tcd.parma..*)
6      && args();
7
8  after(example.wormgame.WormMain target):
9      pcut_MIDletStartApp_startApp(target)
10 {
11     DRMContext context = new DRMContext();
12     context.setPackage("example.wormgame");
13     context.setClazz("WormMain");
14     context.setMethod("startApp");
15     context.setParameters("");
16
17     try {
18         DRMEngine.newInstance().enforce(context);
19     } catch (DRMException drme) {
20         Alert alert = new Alert(
21             "PARMA Rights Enforcement",
22             drme.getMessage(),
23             null,
24             AlertType.WARNING);
25         alert.addCommand(OK);
26         alert.setCommandListener(new WormMainCommandListener(target));
27         alert.setTimeout(5000);
28         addAlert(alert);
29     }
30 }
```

This special case is handled by the second pointcuts pair (Listing 4.19). One difference to the previous

pointcut is that this one picks out execution join points, rather than call join points. This addresses the problem, mentioned above, that the aspect weaver cannot find a match to a join point which is called by JVM internal logic. The after advice (lines eight - 28) of an execution join point can be understood as the last statement of the `startApp` method body. Likewise to the Listing 4.18, a context object is created which represents the protected method signature (lines eleven - 15) and the enforcement engine is called (line 18). Another difference is the way an enforcement failure is handled. The `Alert` object is instantiated with an command listener object (line 26). The command listener is generated for every `MIDlet` class and provides a callback method to the methods `destroyApp` and `notifyDestroyed` . When the user acknowledges the alert, the `MIDlet` is automatically destroyed, so that the user cannot execute it. However, this poses a limitation to the design of MIDlet-based applications which should be protected by the enforcement architecture presented in this thesis. If enforcement is applied to a `startApp` method of any `MIDlet` , the `MIDlet` has to make the methods `destroyApp` and `notifyDestroyed` public, so that the command listener can call these methods. The `MIDlet` class declares them as protected in order to hide the visibility.

4.4 Implementation Statistics

Package	C	F	NCSS	JD	JDL	SCL	MCL	LC	BC
ie.tcd.parma.container	15	86	520	99	852	219	9	84%	85%
ie.tcd.parma.drma.config	5	15	81	20	98	66	0	84%	83%
ie.tcd.parma.drma.context	2	20	52	22	102	25	0	100%	100%
ie.tcd.parma.drma.core	4	10	107	14	68	60	0	0%[a]	0%[a]
ie.tcd.parma.drma.record	4	19	87	21	128	58	18	0%[a]	0%[a]
ie.tcd.parma.drma.schema	405	3312	13695	3855	17160	5003	915	100%	100%
ie.tcd.parma.drma.update	2	17	223	19	107	56	0	0%[a]	0%[a]
ie.tcd.parma.drma.xml	1	5	47	6	40	12	0	83%	100%
Total	438	3484	14812	4056	18555	5499	942	-	-

[a]Despite having tests for this package, the test coverage cannot be measured, because the tests are J2ME tests.

Legend

- **C** - Classes
- **F** - Functions
- **NCSS** - Non-Commenting Source Statements
- **JD** - JavaDoc
- **JDL** - JavaDoc Lines
- **SCL** - Single Commented Lines
- **MCL** - Multi Commented Lines
- **LC** - Line Coverage
- **BL** - Branch Coverage

Table 4.1: Package statistics

Table 4.1 presents the code statistics calculated with JavaNCSS [77]. These statistics are solely based on the Java sources. It does not include any test implementations.

4.5 Summary

This chapter described the implementation of the PARMA enforcement architecture. An introduction was given to the packages in the enforcement architectures and the supporting tools. Also, the development environment was introduced which conforms to best-practices of the open-source community (Section 4.2).

Section 4.3 detailed technical aspects of the enforcement architecture along with each of its components. The Schemas component is responsible for deserialising a valid PARMA REL rights file into a Java object model. The object model is realised in JavaBeans to provide access to the elements of the rights file. Data access and object model creation are separated from one another. Factories for each PARMA REL element are provided to construct the object model using a pull parser approach and thereby reducing the runtime overhead compared to other parsing approaches, e.g., DOM or SAX.

The Container component was implemented without the use of the Reflection API, differentiating it from others for mobile computing environments. It is an Inversion of Control implementation with Dependency Injection. In order to overcome the limitations posed by existing container projects, a stricter component model has to be followed. First, a compliant component has to implement a certain container interface, so that the Container can call back to the component to inject the dependencies. Second, upon registration a component has to declare its dependencies as well. The Container promotes the separation of interfaces from their implementation in order to support multiple platforms. The concrete implementation can be configured depending on the target platform and therefore adds flexibility to the enforcement architecture to deal with multiple platforms.

The Engine presented in Section 4.3.3 depicts individual technical aspects of the enforcement architecture and its components. It describes how the enforcement architecture supports multiple platforms by leaving the decision on which concrete component implementation to instantiate, to the Container. This section covered the details of the enforcement of rights with special attention to the enforcement of execution permissions and the audit, count, datetime, and pointcut constraints.

Section 4.3.3.2 described the implementation of the Daemon services based on the J2ME implementation. Updating the audit log and updating rights are implemented as MIDlets which facilitate the PushRegistry of MIDP 2.0. Thus, it is possible to automatically launch both services upon external events, such as timers and incoming messages. As a result, the user does not need to interact with these services directly. Also, the overview of the Record service was given in Section 4.3.3.3 with J2SE and J2ME specific implementations.

Section 4.3.4 showed how aspect-oriented techniques are used to integrate the enforcement into an existing application. A particular rights file for the worm game demo from SUN's wireless toolkit was introduced in order to simplify the explanation of aspects, especially the difference of the pointcuts and advices between MIDlet's startApp method and application internal methods.

The last section lists the implementation statistics calculated with JavaNCSS (Section 4.4) and the code coverage of each package analysed with JCoverage.

Chapter 5

Evaluation

This chapter reports on the evaluation of the enforcement architecture based on objectives and requirements presented in Chapter 1 and Chapter 2. The enforcement is evaluated concerning platform-independence in Section 5.2 and describes how it is achieved in this thesis. It measures the footprint and the runtime overhead of the enforcement in resource constrained environments and gives suggestions on how to improve upon the results. Also the enforcement architecture follows a deductive approach to comprehend the economics of piracy with respect to security measures which are and are not accomplished with the enforcement. Also, some best practices on enforcement for J2ME environments is presented based on experience with the worm game demo of SUN's wireless toolkit. Finally, the enforcement architecture is compared to other related work which was introduced in Section 2.3.

5.1 Meeting the Requirements

In Section 1.1.3 and Section 3.1, a set of requirements were identified for the enforcement architecture. The following list depicts each individual requirement and outlines how they were met.

1. Platform independence, Extensibility and Adaptability
   In order to achieve an extensible and adaptable framework, a container architecture was implemented (Figure 3.8). The container allows for a consistent resolution of its registered components. In particular, the container maintains a clear separation of interfaces and implementation classes and thereby guarantees that a component can be used independent of its platform-specific implementation. The container only needs to be configured for a target platform using a configuration file which maps the interfaces to their appropriate implementation class.

2. Third-party hardware and software independence
   The enforcement framework is designed in such a way that it does not require additional hardware or external third-party software components. The application software can be distributed as a self-contained package, which contains the enforcement architecture, default rights, and the application.

3. Implicit enforcement

   In order to compose functional logic and the enforcement logic as an orthogonal service, this work implements an aspect-oriented approach. The aspect declares the join points of the application where the enforcement should be woven into the application. Therefore, the application does not have knowledge of any enforcement APIs. The integration of enforcement code into an existing application is supported by an MDA-oriented approach. The platform-independent model is realised with the PARMA REL and transformed with a platform-specific description model into a platform-specific model. The developer's responsibility is solely to specify a valid rights file which is then transformed into an aspect and woven into the application, without additional developer effort required.

4. Network failure transparency

   For the audit-based usage rights model, a network connection is only required when the enforcement attempts to upload the audit log to the PARMA rights server. If a sudden disconnection occurs, the upload is re-scheduled at a later stage and the failure is not propagated to the user.

5. DRM enforcement transparency

   Several mechanisms address the requirement of DRM enforcement transparency. The design and implementation of the enforcement framework has to account for resource-constrained devices. Two approaches have been taken to reduce the runtime overhead of the enforcement, so that it does not disrupt the user perception of the application behavior. Firstly, the rights file is parsed with a pull parser, instead of e.g., a DOM parser which is more memory intensive. Secondly, the enforcement is implemented as a singleton using lazy instantiation. Only the first request to the enforcement requires a set up of the engine. Subsequent requests are directly delegated to the enforcement logic. Also, compliant users are not aware of DRM enforcement, unless it fails to validate the rights.

6. Integrate with PARMA

   The PARMA rights server exposes WSDL interfaces to allow the interaction with a back-end system. The audit log is uploaded to the PARMA rights server using the appropriate WSDL endpoint.

7. Access control for the execute permission

   The only permission which has been applied to the enforcement of rights for application software in PARMA is the "execute" permission defined in ODRL. The enforcement architecture supports this permission and allows the evaluation of audit, count, and date-time constraints.

8. Separate delivery

   To be able to update an existing rights file on the device without replacing the whole application, this thesis presents an SMS-based approach to enable the separate delivery of rights. An SMS is sent to a particular port on the client device with the end-point URL of the new rights file in the message body. This file is downloaded and stored on a well-known location on the client device without user intervention.

5.2 Platform-Independence

As stated in Section 3.1, one of the main goals of the enforcement architecture is to be able to adapt to multiple platforms. The consideration of platform-independence occurs in two different levels. The first level covers design decisions to accommodate platform-independent behavior using the container architecture. The second level covers a platform-independent approach of integrating the enforcement architecture into any existing Java-based application. The following sections assess both levels.

5.2.1 Supporting Multiple Platforms by Design

To date, supporting multiple platforms early in the design of an application plays a pivotal role in driving the adoption for application software. The distributed systems community adopted approaches to provide a "middleware" which tackles the problems of the heterogeneity of the underlying hardware and software architectures by exposing APIs to developers for consuming and interacting with distributed services. Mostly, applications built using a particular middleware architecture or paradigm, such as J2EE, are server-side solution controlled by enterprises. The movement to pervasive computing environments introduces challenges to deal with devices that may be disconnected from the network and that are controlled by the user instead of an enterprise. Despite the difference in scale, the architectural patterns for both environments haven shown significant similarities in order to support multiple platforms by design. Object-oriented software development introduced powerful programming mechanisms to abstract data and to encapsulate the implementation. While data abstraction represents an object's contract or responsibility (interface) towards other objects, encapsulation separates the internal structure and behavior from the interface. As a result, encapsulation provides some advantages in order to accomplish a platform-independent design. First, the separation of the interface and the implementation of that interface allows localised internal modifications to occur on the concrete implementation without affecting clients using this object. Second, the implementation of the interface can be exchanged completely to account for different platform infrastructures. The Dependency Inversion Principle formulates the aforementioned mechanisms into a pattern which states that dependencies on objects should not be to concrete implementations, but to abstractions of them.

Each component of the enforcement architecture is designed in such a way that it hides the implementation and exposes an interface to interested components. Additionally, to reduce the complexity of instantiating dependencies an Inversion of Control container with Dependency Injection has been implemented to provide the functionality to resolve and instantiate components. Thus, the responsibility to instantiate dependencies is removed from a component and moved to the container. With the help of the container the enforcement engine is assembled. Each component is registered and declares its dependencies on abstract representations of other components. That leaves the responsibility to the container to resolve and instantiate the implementations of the interfaces and inject them into a component. The container in this thesis expects a configuration file packaged with it which maps the interfaces to

their implementation class. Therefore, it is not necessary to re-compile the enforcement architecture for different platforms. In order to adapt to different platforms, only the configuration file has to be modified to reflect a platform-specific mapping.

5.2.2 An MDA-Oriented Approach to Weaving Enforcement

Ultimately, the goal is to integrate enforcement into an existing application. An aspect-oriented approach has been taken to weave the enforcement architecture as an orthogonal service into an existing application based on a valid PARMA REL rights file. In order to provide a certain integration transparency for the developer and to support multiple platforms without developer interaction a Model-Driven Architecture approach is followed. MDA envisions a development paradigm where designers create a Platform-Independent Model (PIM) representing the high-level perspective on a system and subsequently transform this model into a Platform-Specific Model (PSM). In the case of an application provider who wants to protect his/her software application with the enforcement architecture presented in this thesis, it is required to create a valid rights file which conforms to the PARMA REL schema. This rights file represents the PIM. Based on the target platform, the application provider has to choose a platform-specific XSLT stylesheet which transforms the PIM into a PSM. The PSM is realised with a single aspect which identifies the join points where the enforcement logic should be applied to. Finally, this aspect is woven with the application. The process of transforming the rights file into a platform-specific aspect and weaving this aspect into the application is completely automated. It is implemented as a Maven plugin and requires just a configuration property to identify the target platform of a distributable of the application.

Strictly speaking, the MDA approach has not been achieved to the maximum extent, because the PSM is not represented in a model, but rather in a specific instance in form of an aspect. However, responsibilities of the developer have been removed by automating the whole process of weaving the enforcement architecture into an application and therefore providing a non-intrusive integration mechanism.

5.3 Resource-Constrained Environments

The key objective of this thesis was to present an enforcement architecture for pervasive computing environments, yet providing support for traditional desktop/server environments. Consequently, the design of such a system has to reflect these decisions and account for characteristics of pervasive computing environments, such as resource constraints and disconnected operation. This section covers how the enforcement architecture dealt with resource constraints, in particular the parsing of XML, the memory footprint, and the runtime overhead of the application.

5.3.1 Parsing XML

XML has become a standard-based way of passing data structures between processes, such as Simple Object Access Protocol (SOAP) [30]. It provides a platform-independent representation of data to foster interoperability between loosely-coupled systems. However, the interpretation of XML and the deserialisation into an object model is a very resource intensive task. This is especially apparent on resource-constrained devices which do have limited processor power and memory capacity. There are two different models of parsing XML and presenting the data to an application: push and pull. Push-based models including DOM and Simple API for XML (SAX) [29] are considered not suitable for resource constrained environments, because both memory consumption and footprint are too high. DOM parsers first read the XML file and the build a complete object model representing it into memory. This memory image of the XML file has to be present as long as the application operates on it. SAX parsers are event-based. They read the XML file sequentially and events are fired every time the parser encounters an XML entity. Equivalent event-handlers registered with the parser capture and handle these events to gather the information needed by the application. However, this approach requires the event-handlers to keep an internal state to obtain a compliant object representation of the XML file. One problem of the previous approaches is that parsers run through the entire XML document in one pass without leaving control over this process to the developer. The common API for XML pull parsing (XMLPull) [36] gives more control over the parsing process. It is possible to pause or stop the parsing and thereby read only parts of the XML file the application is interested in. That significantly reduces the runtime overhead and as a consequence of its simplicity reduces the footprint of the parser so that it is more suitable for pervasive computing environments.

This thesis uses the kXML library [18], because of the following reasons. First, it implements the XMLPull API and second it is used by the kSOAP [17] SOAP WebService client library. The memory footprint is relatively small at 11 KB.

As stated in Section 4.3.1 the object model is created using the XMLPull API. It does not however, make any assumptions on the XMLPull implementation and therefore leaves it to the client of these APIs which implementation to use. The Config component of the engine (see Section 3.2.5 and Section 4.3.3) instantiates the concrete pull parser and passes it to the factories to establish the PARMA REL object model. The implementation of the enforcement architecture also parses just those elements it is interested in to reduce the amount of objects created into memory.

5.3.2 Memory Footprint

The memory footprint of an application is of key importance when it is deployed into pervasive computing environments. Even though the latest generation of JVMs for J2ME devices do not constrain the heap size except by available memory, some devices limit the heap and the .jar-file size.

The following Figure 5.1 depicts the .jar-file sizes for the enforcement architecture based on the original enforcement and obfuscated versions of it.

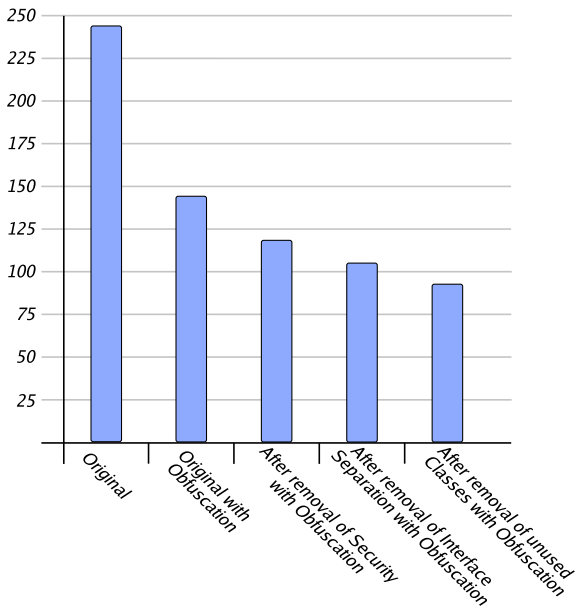

Figure 5.1: Memory Footprint of the Enforcement Architecture

Figure 5.1 outlines the memory footprint of the enforcement architecture. The original .jar-file size is 244 KB which is quite high considering that many of the classes are not used. In particular these are the JavaBeans to convert the PARMA REL rights file into a Java object model. With obfuscation the size is still 144 KB.

A further reduction in the memory footprint can be achieved by refactoring the application and removing the logic which is not used so far. The scope of this thesis did not cover any cryptographic measures to secure the enforcement. However, the JavaBeans of the Schemas component which deal with security, e.g., XMLDsig, are hard-wired. The removal of those classes results in a .jar-file size of 117 KB with obfuscation.

One shortcoming of the design of the Schemas JavaBeans is the separation of interfaces and implementation. This separation does not provide any advantage and therefore should be refactored into single-class implementations instead. Merging the interfaces with the implementation classes yields another 11 KB.

Finally, several XML elements of the ODRL REL are not needed for the enforcement of application rights. These include all permissions except the execute permission. The removal of these classes gives another advantage of 14 KB with obfuscation.

So, refactoring and obfuscation together yield dramatic reductions in code size of 152 KB (62%), from 244 KB to 92 KB (see Figure 5.1), where refactoring and obfuscation have an equal share on the reductions.

Listing 5.1: ProGuard Obfuscation Configuration

```
1 -dontshrink
```

```
2 -defaultpackage ''
3
4 -keepclassmembernames class * {
5     java.lang.Class class$(java.lang.String);
6     java.lang.Class class$(java.lang.String,boolean);
7 }
8
9 -keep class ie.tcd.parma.drma.enforce.EnforcementImpl
10 -keep class ie.tcd.parma.drma.config.ConfigurationImpl
11 -keep class ie.tcd.parma.drma.core.DRMEngine
12 -keep class ie.tcd.parma.drma.core.services.DaemonServicesImpl
13 -keep class ie.tcd.parma.drma.record.RecorderImpl
```

Listing 5.1 illustrates the configuration for ProGuard version 2.1 that had been used to obfuscate the enforcement architecture. The component implementation classes have to be preserved, because they are instantiated by the container with `Class.forName()`. The configuration options `-dontshrink` and `-defaultpackage ''` advice the obfuscator to not shrink the classes and to move all classes into one default package respectively.

5.3.3 Runtime Overhead

The runtime overhead of the enforcement is calculated using SUN's wireless toolkit version 2.1 with tracing enabled for all method calls.

Package	Preparation	Configuration	Enforcement
org/kxml	0	7335	0
ie/tcd/parma/container	0	64	1
ie/tcd/parma/schema	0	126	30
ie/tcd/parma/factory	0	16	0
ie/tcd/parma/drma/config	0	7	0
ie/tcd/parma/drma/core	2	5	0
ie/tcd/parma/drma/record	0	3	6
ie/tcd/parma/drma/enforce	0	3	15
ie/tcd/parma/drma/context	6	0	8
RightsAspect	6	0	0
Total	**14**	**7559**	**60**

Table 5.1: Method Invocations for the Enforcement Architecture

An enforcement call can be divided into three phases: preparation, configuration, and the enforcement

itself. The preparation phase is realised by the `RightsAspect` which had been woven into the application. It sets up the `DRMContext` object and invokes the engine. The configuration phase is responsible for parsing the XML rights file and setting up the enforcement services. The enforcement performs the verification of the access to a protected method. As Table 5.1 indicates, the phase with the highest runtime overhead is the configuration. This phase is executed only once for the first enforcement call. The first enforcement call invokes 7633 methods, whereas subsequent calls to the enforcement use the previously set up enforcement services and thus invoke only 74 methods (1%). Section 3.2.5 details the singleton pattern which implements the control over the object creation of the `DRMEngine` class and therefore allows the configuration to occur only once.

Table 5.1 outlines the number of method invocations for each package which participates in the enforcement of rights. The configuration phase accounts for 99% percent of the method invocations, because it involves registering each enforcement service with the container, which in turn instantiates and initialises each one of them. Most of the runtime overhead is spent parsing the PARMA rights file (96%) and creating the Java object model representing it (2%).

The user's perception of the system's performance can be further improved by executing the configuration phase explicitly when the application is started up, instead of lazy instantiating the singleton with the first enforcement call. This increases the startup time of a `MIDlet` slightly, but consequently real calls to the enforcement do not disrupt the user. One of two mechanisms could be implemented to achieve an early configuration. First, the singleton could be implemented to support load-time singletons by having a single instance of the class as a static field. A second solution is to extend the aspect to call the configuration phase explicitly before the `MIDlet` is started.

To conclude, the enforcement does not involve a high runtime overhead. Only the first enforcement call requires the execution of the configuration phase which could disrupt the perception of the application flow. However, subsequent enforcement calls impose only a little runtime overhead. The user experience can be further improved, if the configuration phase is moved to class-loading time instead of lazy initialisation.

5.4 Economic Model of Piracy

In an ideal world, there are no software pirates and everyone is an honest user who buys the software, rather than cracking it. However, the incentives to commit piracy are high for expensive software applications. Devanbu and Stubblebine [53] introduced a model of the economics (Equation 2.1) of piracy which had been extended (Equation 2.2 and Equation 2.3) by Filigrane [65] to also cover the extraction of proprietary algorithms (see related works in Section 2.3.1).

A legal framework to combat piracy is not sufficient. The legal aspects have to be augmented by technical protection measures to prevent a pirate to circumvent the security measures or to increase the cost associated with it. But, at the same time, such measures should not shy away the honest

user. Consequently, practical security measures have to be considered for the special case of application software individually. It is not appropriate to provide a most advanced and sophisticated protection system for cheap applications. Or vice versa, a basic protection mechanism is not sufficient for expensive applications.

In the case of this thesis, these equations can be applied as well to analyse the weaknesses of the system. However, the mathematical model is used to estimate the relative values, rather than calculating them. The remainder of this section looks at potential attacks and puts them into context of the model of the economics of piracy in Equation 2.1. For this framework it is not important to look at the Equation 2.2 and the Equation 2.3, because it is going to be open-sourced and therefore stealing algorithms or reverse-engineering parts of it are not applicable. Also, the protection measures to prevent stealing and reverse-engineering are orthogonal to any application. Mostly, obfuscation and watermarking techniques can be applied after the application is developed, but hacking an application based on intrinsic flaws or holes in the design are considered more important as part of this analysis.

The enforcement itself is a mechanism to prevent piracy and therefore should provide appropriate protection measures as an added value to any application software.

5.4.1 Potential Attacks

This section describes the potential attacks in order to render the enforcement in one way or the other ineffective. The enforcement architecture can be circumvented in several ways, especially so, because it was out of scope of this thesis to employ cryptographic techniques into the enforcement. This section deals with potential attacks and discusses the problems the enforcement architecture is facing now and gives suggestions on how to improve the situation. The suggestions do not provide a complete solution to the problem, but pointers to future work and research.

5.4.1.1 Prevent Audit Log Update

The audit-based rights model is a post-pay model to overcome limitations of pervasive computing platforms. Pervasive computing platforms cannot guarantee an on-demand network connection to verify rights or provide a pay-per-use model. Therefore, post-pay models provide a nice alternative to the user, but also introduces a high incentive to prohibit the submission of audit logs. The user always has control over his/her device which allows him/her to disconnect an update request. The reason, however, may not be malicious. The user could be on a holiday in a foreign country and just does not want to incur the roaming charges associated with a remote call. The intention to prevent all update request can be seen as hacking the system, because the user thereby uses the application without paying for it. However, the probability of getting caught is very high, because the PARMA rights server is aware of users who signed up for an audit-based agreement and therefore can remotely enforce the agreement by notifying the user to update the audit log or in the worst case revoke the rights by sending a kill-pill to the user's device. The associated fine is probably not very high, because it would be difficult to extrapolate usage,

unless some historic data is available.

The enforcement presented in this thesis generally re-schedules the update with the assumption that the user is more willing to submit the audit log at a later stage. At the moment the enforcement does not have the ability to perform an on-device audit log analysis and notify the user, if he/she is overdue.

5.4.1.2 Delete/Modify the Enforcement Data

An infringement with little effort is the deletion or modification of the runtime enforcement data. That includes audit logs and counters kept in a persistent storage area on the device. Deleting a counter is the hardest to come by and therefore the probability of getting caught is very low. Modifying or deleting the audit log is associated with a higher probability of getting caught, because of PARMA rights server's awareness. In a scenario where no tamper resistant secure storage is available, audit logs have to accommodate for these vulnerabilities by using e.g., Method Authentication Codes (MAC) [101]. A MAC is a checksum appended to a log entry to prove the integrity of the log. Also, a MAC code could be derived from the previous entry as well, so that subsequent log entries are linked to the according previous one. This mechanism fights log modifications, however, deleting a complete log is still possible. A tamper resistant solution is necessary to protect counter values and audit logs in a more appropriate way. Usually, these solutions are more expensive to deploy, because tamper-resistant storage is not a built-in feature of client devices. Maheshwari et. al. [81] introduced a trusted secure database system that leverages a trusted processing environment and trusted storage to extend tamper-detection to a scalable amount of untrusted storage. Shapiro and Vingralek [95] discuss a client system model for DRM persistent data which is also based on tamper-resistant storage. The SIM card on a mobile phone provides such a secure and tamper-resistant storage which can be utilised with a Java optional package for J2ME in the near future (Security and Trust Services API for J2ME [16]).

5.4.1.3 Circumvent the Enforcement

One of the main benefits of aspect-oriented software development is also problematic in certain scenarios. Aspects can be woven into applications to provide orthogonal services for the application without changing the source-code. This fact can also be abused in order to negate the same feature. In the case of enforcement this is indeed a problem, because it is relatively easy to do and the probability of getting caught is low. A technical savvy developer could implement an "anti-aspect" in five lines of code which declares a call or execute join point to the `DRMEngine.enforce()` with an empty around advice and weave it into the protected application. As a result, no enforcement logic is ever executed and the user can utilise all the protected features of the application.

This problem cannot be easily solved. The application could be locked to only the intended user with dynamic code-decryption mechanisms similar to Filigrane [65], but that would induce a resource and hardware intensive task and therefore would not be feasible for pervasive computing environments. A signed application could be as easily circumvented. If the enforcement verified the signature at runtime

to ensure the integrity of the application, the same "anti-aspect" approach can be used to cancel this logic.

5.4.1.4 Modify the Rights File

One of the most obvious attacks against the enforcement architecture is by modifying the rights file itself. Without any cryptographic security measures, it is possible to delete the date-time and/or count constraints to render the enforcement ineffective. At runtime the enforcement would not encounter any constraints, and as a result of it, grants access to protected features. The cost of hacking the rights file is zero and the probability of getting caught low.

However, the enforcement can be extended to provide a higher degree of security for the rights file. Using established cryptographic techniques, the rights file can be signed, which usually means that the value based on the hash-code of the document and a private signing function is appended to the document. The private signing function encrypts the hash-code with the rights-holder's private key. On the client device the integrity of the document can be checked with the according public key and deny access to protected features if the verification fails.

The XML standard XMLDsig [38] provides this functionality for XML documents and is fully supported in PARMA REL. The corresponding Java object model can also be established with only few additions to the Schemas factories. The enforcement engine would need to be extended with another enforcement service which verifies the rights file before the call is delegated to the access control.

5.5 Best Practices Rights Enforcement

Throughout the writing of this thesis, several demonstrations had been performed to test certain features of the enforcement. As a result, several best practices rules had been collected and are now presented in this section.

5.5.1 Strategies to Prevent Application Execution

While it is straight-forward to protect the startup of J2SE applications, it is more complicated for `MIDlet` s. The advice type of the enforcement of rights for J2SE desktop applications' `main()` method can be declared as a before or an around advice in the rights file, so that the application does not initialise unless access is granted. This assumption cannot be made for the `MIDlet` s' equivalent `startApp` method, because in case access is denied for that `MIDlet` an alert has to be presented to the user. However, it cannot be guaranteed that the required display is available until after the `startApp` method has successfully completed. Consequently, if a `MIDlet` 's `startApp` method is protected the advice type of this method has to be declared as an after advice.

5.5.2 Do not Protect Busy Methods

In order to efficiently apply enforcement rules to an application, the person responsible for the declaration of the rights file has to know the application. The enforcement can not be associated with any feature/method of the application. Enforcement should not be applied to recursive methods or methods which are executed very often.

5.5.3 Do not Protect Methods which Change External State

Also, enforcement should not be applied to methods which change an external state. This external state could be relied on by other parts of the application. If a certain state is expected and the enforcement prevents the transition into that state than the whole application may block and not respond to the user.

5.5.4 Associate Protection with User Interaction

The most successful way of protecting applications is to associate protection to methods which implement user interaction. For example, enforcing rights for the multiplayer feature over Bluetooth or changing levels of a game does not render the application useless, if access is prohibited.

5.5.5 Make notifyDestroyed() and destroyApp(boolean) public

If the execution of a MIDlet should be protected, the method which signals the MIDlet to terminate and enter the "destroyed" state and the method that notifies the application management software that the MIDlet has entered the "destroyed" state should be declared public. This is a limitation posed by the PARMA enforcement architecture, in order to be able to call these methods in the aspect, which is woven into the application.

5.6 Comparison with Existing Enforcement Architectures

The implementation of an enforcement architecture based on the overall PARMA architecture has implications on the design. The PARMA REL was designed for applications in pervasive computing environments by providing audit-based and feature-based usage rights models. Also, PARMA exposes management interfaces in WSDL to interact with the rights server, e.g., updating audit logs.

The remainder of this section deals with the benefits and weaknesses of the enforcement architecture compared to other related works.

Table 5.2 summarises the comparison of existing enforcement architecture as described in Section 2.3 and the enforcement architecture presented in this thesis.

Both OMA DRM and the PARMA enforcement architecture derive from a single source and therefore resemble similar system level features. While OMA DRM is a joint effort by major players in the industry to deliver and implement a standard to protect the rights of the copyright holder for media content, the

	Filigrane	WIT-Software	xoRBAC	OMA DRM	PARMA
Extensible	no	yes	yes	no	yes
DRM Standard	no	-	yes	yes	yes
Platform Independence	no	no	no	no	yes
DRM Transparency	weak	strong	-	strong	strong
Tool support	-	-	-	yes	yes
Network Transparency	weak	weak	weak	strong	strong
Mobile Networks	no	-	no	yes	yes
Resource Constrained Devices	no	yes	no	yes	yes
Enforcement	implicit	implicit	-	implicit	implicit
Functional Layers	3 - 5	(3) - 4/5	2/5	2 - 5	5[a]
TCPS	strong	-	strong	weak[b]	weak
Fine-Grained Rights Access	no	yes	no	no	yes
Audit-Based Rights Model	no	-	no	no	yes

[a]Excluding the overall PARMA architecture.
[b]Version 2.0 of OMA DRM adds enhanced security features, such as PKI support.

Table 5.2: Comparison with Existing Enforcement Architectures

PARMA enforcement architecture's goal was to implement the enforcement of PARMA REL rights in a platform-independent manner but at the same time target pervasive computing platforms. The shift from the protection of media content to the protection of fine-grained features of application software comes along with the need of an architecture that can adapt to multiple platforms, because the PARMA enforcement architecture has to be packaged with the application. Unlike implementations of OMA DRM, no operating system support is available so far, which allows the enforcement of feature-based application usage rights.

From an architectural perspective the PARMA enforcement architecture is similar to the xoRBAC implementation. Both approaches concentrate on the enforcement of rights of ODRL REL. The difference is that xoRBAC enforces the full set of rights expressed in ODRL with attention to security related aspects, such as remotely verifying the identity and the rights of the consumer. The PARMA enforcement architecture, described in this thesis, on the other hand, is more concerned with feature-level application usage rights enforcement expressed in PARMA REL. Because of the requirement to verify the rights and the identity of the user, xoRBAC is not suitable for pervasive computing environments where network availability is intermittent. The PARMA enforcement architecture, however, supports occasionally-connected network environments using the audit-based rights model.

One of the major goals of the PARMA enforcement architecture was the support for multiple platforms (see Section 5.2) by providing an overall framework that can be extended in platform-specific ways. This thesis presented two implementations, one for J2ME and one for J2SE, and demonstrated

the adaptability on two levels. First, the container architecture can be configured to instantiate the platform specific components. Second, the rights and the implementation of the enforcement architecture can be integrated into existing applications transparently. Compared to the enforcement architectures presented in Section 2.3 only the framework implemented by WIT-Software achieved a similar level of DRM transparency for application rights enforcement. As well as the PARMA enforcement architecture WIT-Software's framework injects enforcement logic into existing applications in a module of an Over-The-Air provisioning server using byte-code engineering. Due to their limitations on MIDP application rights enforcement, this approach is not platform-independent. Also, WIT-Software aimed at providing a framework to inject rights transparently into an existing MIDP application. It does not, however, implement a means of enforcing rights. Instead, this work left the framework extensible in order to support the injection of any enforcement architectures.

The main deficiency of the PARMA enforcement architecture is the lack of security (see Section 5.4). Compared to other related works two projects implement a satisfactory security level. However, both approaches involve relatively high costs. Filigrane implemented a strong protection mechanism based on cryptographic techniques which is based on smart cards. For a user of this system, this entails a certain investment to purchase smart cards and a smart card reader. Also, the JVM has to be extended with the Filigrane engine that decrypts the Java byte-code of the protected application at runtime. In contrast to Filigrane, xoRBAC only implements the access control mechanism for rights expressed in ODRL, rather than providing a complete rights management architecture. This implementation of the access control mechanism is realised in a server. That means the user has to issue a remote access request to the access control server in order to verify access. Both approaches are not suitable, however, for pervasive computing environments. The former one is solely targeted to desktop applications, while the latter requires on-demand network connections which cannot be guaranteed in pervasive environments.

OMA DRM version 2.0 is underway which introduces a stronger protection based on Public Key Infrastructure. But it is assumed to be not widely deployed before the end of the year 2005. Also, the support for feature-level application usage rights is not likely to be specified in the near future.

From the preceding comparison of existing systems with the PARMA enforcement architecture, see Table 5.2, it is the only enforcement architecture that is suitable for the fine-grained enforcement of rights for software applications in pervasive environments.

5.7 Summary

This chapter described an evaluation of the work presented in this thesis. The evaluation criteria included platform-independence (Section 5.2), resource-constrained environments (Section 5.3), the model of the economics of piracy (Section 5.4), best practices gathered from the experience of using the enforcement architecture (Section 5.5), and the PARMA enforcement architecture is compared to other related work (Section 5.6).

The platform-independence of the enforcement architecture was presented focusing on two aspects, design decisions of the architecture and an MDA-oriented approach of weaving aspects into existing applications. This section concludes that a high degree of platform-independence was accomplished by separating interface declaration and implementation of enforcement services and by assembling the enforcement services with a container architecture.

Further, the enforcement was evaluated on how certain issues of resource constrained environments were solved. The choice of the appropriate XML parsing model was discussed and the footprint as well as the runtime overhead were analysed. Suggestions were made to improve the footprint of the enforcement by refactoring parts of the enforcement and by using obfuscation techniques.

The effectiveness of the enforcement architecture was measured with a model of the economics of piracy and potential attacks on the system. The potential attacks covered the prevention of updating audit logs, modifications to the enforcement data and rights file, and how aspect-oriented techniques can circumvent the enforcement.

Finally, some best practices were presented which are based on experience in using the enforcement followed by a comparison of the enforcement architecture with existing related works.

Chapter 6

Conclusion

This thesis described the design and implementation of an ODRL-based enforcement architecture and presented how it can be adapted to different computing platforms. A container architecture was introduced as a centralised component that can be used to assemble enforcement services, and that provides the ability to resolve and instantiate its registered services with the help of a configuration file. The PARMA enforcement architecture was evaluated with respect to platform-independence, ability to operate in resource-constrained environments, and piracy criteria.

In this chapter, the contents of this thesis are summarised and an overview of the contributions of the thesis as well as potential future work are presented.

6.1 Thesis Summary

Chapter 1 introduced the problem of enforcing application rights in applications that operate in pervasive computing environments. It motivated the challenges of dealing with resource-constrained devices, volatile network environments, and the use of standard-compliant rights expression languages. Based on the high-level goals of a standard-compliant enforcement engine for pervasive computing environments some non-functional requirements for this thesis were stated followed by their implementation into the contributions of this thesis. The contributions covered the design and implementation of a container architecture to assemble and manage enforcement services, the enforcement architecture, and an MDA-oriented approach of composing rights-enabled applications.

Chapter 2 provided the background for container architectures and digital rights management. PARMA REL was introduced as an extension to ODRL to account for the characteristics of application rights management for pervasive computing platforms, particularly with regard to the feature-based and audit-based rights models. Finally, four related enforcement architectures were detailed and evaluated based on the requirements of this thesis.

Chapter 3 started with non-functional and functional requirements for the enforcement architecture. The following sections described the architecture based on functional, logical, object, and dynamic mod-

els. Chapter 4 detailed the implementation of the different subsystems, important algorithms of the architecture, and some of the main employed patterns. The implementation presented in this chapter was based mainly on a version of the enforcement architecture for the J2ME platform.

Chapter 5 evaluated the enforcement architecture with respect to platform-independence, ability to operate in resource-constrained environments, and Filigrane's model of the economics of piracy criteria. The container architecture and the MDA-oriented approach of weaving the enforcement with an application provided the basis of a platform-independent deployment. The resource utilisation of the enforcement engine were measured for three identified phases of the enforcement: preparation, configuration, and enforcement. Potential attacks on the enforcement were evaluated with the model of the economics of piracy followed by best practices for the use of the enforcement architecture. The evaluation concluded with a feature-based comparison of the enforcement architecture with the related systems introduced in Chapter 2.

6.2 Contributions

This thesis addressed the problem of developing an enforcement architecture for pervasive computing environments with the capability to adapt to different platforms by simply updating deployment configuration files. The main contributions of this thesis are an Inversion of Control container architecture, the enforcement architecture, and an MDA-oriented approach of transparently weaving the enforcement into any existing J2ME and J2SE application.

The container architecture contributes a light-weight implementation of an IoC container with Dependency Injection designed to be MIDP 2.0 compliant, but also designed to work on other platforms, such as J2SE. It provides a central and consistent way of resolving and instantiating components. Thus, the responsibility to instantiate dependencies is moved from a component to the container. The container expects a configuration file packaged with it which maps the interfaces to their implementation class. Therefore, it is not necessary to re-compile the enforcement architecture for different platforms. In order to adapt to different platforms, only the configuration file has to be modified to reflect a platform-specific mapping.

The enforcement architecture consists of a number of components which are registered with the container. The enforcement architecture exposes one service which implements the enforcement of PARMA REL compliant permissions specified in a rights file. The rights file is parsed with a pull parser and deserialised into a JavaBean object model. The enforcement component operates on this rights representation to enforce the "execute" permissions for the features of an application that have been specified in as requiring rights. A permission to an application can be constrained by a maximum counter value, or date time restrictions, e.g., expiry date. Moreover, two server implementations for the J2ME platform are provided, one to update the audit log at regular time intervals and one to realise the separate delivery of rights to the mobile phone.

An MDA-oriented approach contributes a transparent weaving process for the application provider. In the case of an application provider who wants to protect his/her software application with the enforcement architecture, he/she is only required to create a valid rights file which conforms to the PARMA REL schema. This rights file represents the platform-independent model (PIM) of rights for an application. Based on the target platform, the application provider has to choose a platform-specific XSLT stylesheet which transforms the PIM into a platform-specific model (PSM) for a target application. The PSM is realised with a single aspect which identifies the join points where the enforcement logic should be woven into the application. Finally, this aspect is woven into the application. The process of transforming the rights file into a platform-specific aspect and weaving this aspect into the application is completely automated and only requires a configuration property to identify the target platform of a distributable of the application.

6.3 Future Work

The scope of this work has led to many interesting research paths that could not be fully investigated in this thesis, but would provide useful extensions to accomplish a fully integrated more feature rich enforcement architecture.

Section 5.4 illustrated some shortcomings of the enforcement architecture with regards to security. Despite the fact that the enforcement of rights is always vulnerable to attacks, some practical security measures could be implemented to make it more difficult for an attacker to breach the system. Firstly, any communication channels could incorporate an end-to-end encryption scheme to prevent man-in-the-middle attacks and to ensure the integrity of the message. For example, this would be important for the transfer of audit logs to a PARMA rights server. Secondly, local enforcement data, such as the constraint values of the rights file and the runtime enforcement data, could be stored in a tamper-resistant storage area. In the near future JSR-177 is going to be employed as an optional package on mobile phones which enables MIDP applications to facilitate access to SIM cards. This could provide the secure storage area that is required for the local enforcement data. Also, the audit log could be extended to leverage a MAC checksum to link log entries to the respective previous one in order to prevent tampering with single log entries. Third, a signature-based mechanism could be used to verify the validity of PARMA REL rights files. The verification of the rights file could be implemented by another enforcement service. The implications of the implementation of these features raise some questions of how to optimise the memory footprint and runtime overhead for pervasive computing environments and how to manage relationships of the rights holder and the user with the use of public key infrastructure (PKI) without compromising privacy.

Further potential work is the integration of a payment and update service to allow real-time swapping of rights while the application is in use. This would be useful if the user of a protected application finds a feature disabled, but he/she would like to enable it for future requests. One way of approaching this

task would be to extend the alert messages presented to the user to allow the update of the current rights model in such a way that access to the currently disabled feature is granted. ODRL supports the specification of payment requirements. This information can be used in the XSLT stylesheets that generate the platform-specific aspects to integrate an update option in the alert message. Once the update is complete, the container has to re-configure certain services to reflect this change.

From the perspective of rights management realised by the PARMA rights server, the enforcement architecture could be extended to be capable of revoking rights. This requires the implementation of another server process which listens for a "kill-pill" message from the PARMA rights server and deletes or invalidates the rights file stored on the local device.

The MDA-oriented approach implemented in this thesis transforms rights into aspects and weave them into an application is not fully MDA compliant. MDA describes the model-to-model transformation of MOF-based XMI encoded models. The code-generation presented in this thesis could be extended to cover the model-to-model transformation as a superset in order to be able to integrate this process into a fully MDA driven product life-cycle.

Appendix A

Maven Plugins

- `maven-axis-plugin` - implements the Axis developer tools [60, 2, 4]. (Contributed to maven-plugins.sf.net [20])
- `maven-jcoverage-plugin` - implements Java code coverage analysis [14, 67]. (Contributed to maven.apache.org [1])
- `maven-dotuml-plugin` - implements the generation of UML class diagrams based on doclet tags in Java source-code. Also, provides the functionality to generate sequence diagrams using UMLGraph [100]. (Contributed to maven-plugins.sf.net)
- `maven-parma-plugin` - implements the generation and weaving of PARMA Aspects in Section 3.2.6.
- `maven-transform-plugin` - implements the transformation of documentation in Docbook [105] format into HTML and PDF. (Contributed to maven-plugins.sf.net)

Appendix B

Stylesheet To Transform Rights Into Aspects

This appendix lists the XSLT stylesheets that were implemented to transform the platform-independent rights specification in PARMA REL into platform-specific aspect (for the J2ME and J2SE versions see Listing B.1 and Listing B.2 respectively).

Listing B.1: XSLT Stylesheet for the J2ME Platform

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
3 <xsl:stylesheet version="1.0"
4   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5   xmlns:o-dd="http://odrl.net/1.1/ODRL-DD"
6   xmlns:parma="http://www.dsg.cs.tcd.ie/parma/1.3"
7   xmlns:o-ex="http://odrl.net/1.1/ODRL-EX"
8   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
9
10
11  <xsl:output method="text"/>
12
13
14  <xsl:template match="/">
15 // ************************************************************************
16 //
17 // Generated by PARMA
18 //
19 // Copyright (c) 2003 - 2005
```

```
20 // Distributed Systems Group,
21 // Department of Computer Science
22 // Trinity College Dublin
23 // http://www.dsg.cs.tcd.ie
24 //
25 // All Rights Reserved
26 // ********************************************************************
27
28
29 import ie.tcd.persl.drma.context.DRMContext;
30
31 import ie.tcd.persl.drma.core.DRMEngine;
32 import ie.tcd.persl.drma.core.DRMException;
33
34 import java.util.Vector;
35
36 import javax.microedition.lcdui.Alert;
37 import javax.microedition.lcdui.AlertType;
38 import javax.microedition.lcdui.Canvas;
39 import javax.microedition.lcdui.Command;
40 import javax.microedition.lcdui.CommandListener;
41 import javax.microedition.lcdui.Display;
42 import javax.microedition.lcdui.Displayable;
43 import javax.microedition.lcdui.Graphics;
44
45 import javax.microedition.midlet.MIDlet;
46
47     <xsl:variable name="countExecutes"  select="count(//o-ex:permission/o-dd:
          execute)"/>
48     <xsl:variable name="countPointcuts" select="count(//o-ex:permission/o-dd:
          execute/o-ex:constraint/parma:pointcut)"/>
49
50 /**
51  * The RightsAspect has been generated for Asset ID <xsl:value-of select="o-ex
       :rights/o-ex:agreement/o-ex:asset/@id"/>.
52  * Entry points into the DRMEngine are:<xsl:if test="number($countExecutes >
       $countPointcuts)">
53  * - MIDlet.startApp()s (Note: if you would not like to have all MIDlet.
       startApp automatically
54  *     included in the rights enforcement, you would need to specify each one
```

```
      you would like to
55  *      have individually!)</xsl:if>
56  * <xsl:for-each select="//parma:pointcut">
57  * - <xsl:apply-templates select="parma:package" mode="signature"/>
58      </xsl:for-each>
59  *
60  * This aspect has to be woven into the application which is intended
61  * for rights enforcement.
62  *
63  * @author Dominik Dahlem
64  */
65 public aspect RightsAspect
66 {
67     private static final Command OK = new Command("OK", Command.EXIT, 1);
68     private static final int ALERT_TIMEOUT = 4000;
69     private static final String ALERT_TITLE = "PARMA - Rights Enforcement";
70
71     public AlertCanvas canvas = new AlertCanvas();
72     private boolean initialized = false;
73     private Display display;
74
75
76     /**
77      * This method handles alerts. Only if the no midlet has been initialized
            it will
78      * queue the Alert with the AlertCanvas. Otherwise, the alert is displayed
            immediately.
79      */
80     private void addAlert(Alert alert)
81     {
82         if (initialized &amp;amp;&amp;amp; (display != null)) {
83             Displayable d = display.getCurrent();
84             display.setCurrent(alert, d);
85         } else {
86             canvas.show(alert);
87         }
88     }
89
90     /**
91      * This pointcut makes sure that the private member midlet is set to the
```

```
        midlet which has the
92       * current display.
93       */
94      pointcut pcut_initializeMidletMember(MIDlet midlet):
95          target(midlet)
96          &amp;&amp; !within(ie.tcd.persl..*)
97          &amp;&amp; execution(* MIDlet+.startApp());
98
99      after(MIDlet midlet): pcut_initializeMidletMember(midlet)
100     {
101         display = Display.getDisplay(midlet);
102         canvas.init(display);
103         initialized = true;
104     }
105
106     <xsl:for-each select="//parma:pointcut">
107       <xsl:variable name="returnType" select="parma:package/parma:class/parma:
            method/parma:returnType"/>
108       <xsl:variable name="method" select="parma:package/parma:class/parma:
            method/parma:methodName"/>
109       <xsl:variable name="class" select="parma:package/parma:class/parma:
            className"/>
110       <xsl:variable name="package" select="parma:package/parma:packageName"/>
111       <xsl:variable name="targetObjectParameter"><xsl:value-of select="
            $package"/>.<xsl:value-of select="$class"/> target</xsl:variable>
112       <xsl:variable name="argsTypesAndVars"><xsl:apply-templates select="parma
            :package/parma:class/parma:method/parma:argType" mode="TypesAndVars"
            /></xsl:variable>
113       <xsl:variable name="argsVarsOnly"><xsl:apply-templates select="parma:
            package/parma:class/parma:method/parma:argType" mode="varsOnly"/></
            xsl:variable>
114       <xsl:variable name="argsTypesOnly"><xsl:apply-templates select="parma:
            package/parma:class/parma:method/parma:argType" mode="typeOnly"/></
            xsl:variable>
115       <xsl:variable name="parameterList">
116         <xsl:choose>
117           <xsl:when test="string($argsTypesAndVars) = ''">
118             <xsl:value-of select="$targetObjectParameter"/>
119           </xsl:when>
120           <xsl:otherwise>
```

```
121             final <xsl:value-of select="$targetObjectParameter"/>, <xsl:value-
                    of select="$argsTypesAndVars"/>
122           </xsl:otherwise>
123         </xsl:choose>
124       </xsl:variable>
125       <xsl:variable name="parameterNameList">
126         <xsl:choose>
127           <xsl:when test="string($argsVarsOnly) = ''">target</xsl:when>
128           <xsl:otherwise>target, <xsl:value-of select="$argsVarsOnly"/></xsl:
                  otherwise>
129         </xsl:choose>
130       </xsl:variable>
131       <xsl:variable name="adviceReturn">
132         <xsl:choose>
133           <xsl:when test="parma:adviceType = 'around'">
134             <xsl:value-of select="$returnType"/>
135           </xsl:when>
136           <xsl:otherwise>
137           </xsl:otherwise>
138         </xsl:choose>
139       </xsl:variable>
140
141     /* Pointcut pair: The first one makes sure that each join point is picked
            which is
142      * neither a midlet.startApp() method nor inside the ie.tcd.persl packages
            .
143      * The second one picks join points which override the startApp() method
            of MIDlet.
144      */
145     /**
146      * Pointcut describing a call joinpoint which is not a startApp method of
            a subclass of
147      * MIDlet and which is not called within the package ie.tcd.persl packages
            .
148      */
149     pointcut pcut_NoMIDletStartApp_<xsl:value-of select="$method"/>(<xsl:value
          -of select="$parameterList"/>):
150         call (<xsl:value-of select="$returnType"/> <xsl:text> </xsl:text><xsl:
              value-of select="$package"/>.<xsl:value-of select="$class"/>.<xsl:
              value-of select="$method"/>(<xsl:value-of select="$argsTypesOnly"
```

```
            />))
151         &amp;amp;&amp;amp; !call(void MIDlet+.startApp())
152         &amp;amp;&amp;amp; !within(ie.tcd.persl..*)
153         &amp;amp;&amp;amp; target(target)
154         &amp;amp;&amp;amp; args(<xsl:value-of select="$argsVarsOnly"/>);
155
156     <xsl:value-of select="$adviceReturn"/><xsl:text> </xsl:text><xsl:value-of
            select="parma:adviceType"/>(<xsl:value-of select="$parameterList"/>):
157     pcut_NoMIDletStartApp_<xsl:value-of select="$method"/>(<xsl:value-of
            select="$parameterNameList"/>)
158     {
159         // instantiate the context for this method invocation
160         DRMContext context = new DRMContext();
161
162         context.setPackage("<xsl:value-of select="$package"/>");
163         context.setClazz("<xsl:value-of select="$class"/>");
164         context.setMethod("<xsl:value-of select="$method"/>");
165         context.setParameters("<xsl:apply-templates select="parma:package/
                parma:class/parma:method/parma:argType" mode="
                typeOnlyWithoutWhitespace"/>");
166
167         // enforce the rights in a given context
168         try {
169             DRMEngine.newInstance().enforce(context);
170     <xsl:if test="parma:adviceType = 'around'">
171             proceed(<xsl:value-of select="$parameterNameList"/>);
172     </xsl:if>
173         } catch (DRMException drme) {
174             Alert alert = new Alert(
175                 ALERT_TITLE,
176                 drme.getMessage(),
177                 null,
178                 AlertType.WARNING);
179
180             alert.setTimeout(ALERT_TIMEOUT);
181             addAlert(alert);
182         }
183     }
184
185     /**
```

```
186      * Pointcut describing a execution joinpoint which is a startApp method of
             a subclass of
187      * MIDlet and which is not called within the package ie.tcd.persl packages
             .
188      */
189     pointcut pcut_MIDletStartApp_<xsl:value-of select="$method"/>(<xsl:value-
             of select="$parameterList"/>):
190         execution (<xsl:value-of select="$returnType"/> <xsl:text> </xsl:text
                ><xsl:value-of select="$package"/>.<xsl:value-of select="$class"
                />.<xsl:value-of select="$method"/>(<xsl:value-of select="
                $argsTypesOnly"/>))
191         &amp;&amp; target(target)
192         &amp;&amp; execution(void MIDlet+.startApp())
193         &amp;&amp; !within(ie.tcd.persl..*)
194         &amp;&amp; args(<xsl:value-of select="$argsVarsOnly"/>);
195
196     <xsl:value-of select="$adviceReturn"/><xsl:text> </xsl:text><xsl:value-of
             select="parma:adviceType"/>(<xsl:value-of select="$parameterList"/>):
197     pcut_MIDletStartApp_<xsl:value-of select="$method"/>(<xsl:value-of select=
             "$parameterNameList"/>)
198     {
199         // instantiate the context for this method invocation
200         DRMContext context = new DRMContext();
201
202         context.setPackage("<xsl:value-of select="$package"/>");
203         context.setClazz("<xsl:value-of select="$class"/>");
204         context.setMethod("<xsl:value-of select="$method"/>");
205         context.setParameters("<xsl:apply-templates select="parma:package/
                parma:class/parma:method/parma:argType" mode="
                typeOnlyWithoutWhitespace"/>");
206
207         // enforce the rights in a given context
208         try {
209             DRMEngine.newInstance().enforce(context);
210     <xsl:if test="parma:adviceType = 'around'">
211             proceed(<xsl:value-of select="$parameterNameList"/>);
212     </xsl:if>
213         } catch (DRMException drme) {
214             Alert alert = null;
215
```

```
216             if (!"startApp".equals("<xsl:value-of select="$method"/>")) {
217                 alert = new Alert(
218                     ALERT_TITLE,
219                     drme.getMessage(),
220                     null,
221                     AlertType.WARNING);
222 
223                 alert.setTimeout(ALERT_TIMEOUT);
224             } else if ("startApp".equals("<xsl:value-of select="$method"/>"))
                    {
225                 alert = new Alert(
226                     ALERT_TITLE,
227                     drme.getMessage(),
228                     null,
229                     AlertType.WARNING);
230 
231       <xsl:if test="boolean(parma:package/parma:class/@isMain)">
232                     alert.addCommand(OK);
233                     alert.setCommandListener(new <xsl:value-of select="$class"
                            />CommandListener(target));
234       </xsl:if>
235                 alert.setTimeout(ALERT_TIMEOUT);
236             }
237             addAlert(alert);
238         }
239     }
240     </xsl:for-each>
241 
242     <xsl:for-each select="//o-ex:permission/o-dd:execute/o-ex:constraint/parma
          :pointcut">
243       <xsl:if test="boolean(parma:package/parma:class/@isMain)">
244         <xsl:variable name="class" select="parma:package/parma:class/parma:
              className"/>
245         <xsl:variable name="package" select="parma:package/parma:packageName"
              />
246         /**
247          * We need a strongly typed CommandListener in J2ME in order to
              destroy the midlet. We can not
248          * call destroyApp on the MIDlet class, because it is protected.
              Instead we expect this
```

```
249          * method to be public in the sub-class.
250          */
251         class <xsl:value-of select="$class"/>CommandListener implements
                CommandListener
252         {
253             private <xsl:value-of select="$package"/>.<xsl:value-of select="
                    $class"/> m;
254
255             <xsl:value-of select="$class"/>CommandListener(<xsl:value-of
                    select="$package"/>.<xsl:value-of select="$class"/> midlet)
256             {
257                 this.m = midlet;
258             }
259
260             /**
261              * command-callback method. Handle commands from the user
                     interface.
262              *
263              * @param c user interface command requested
264              * @param s screen object initiating the request
265              */
266             public void commandAction(Command c, Displayable s)
267             {
268                 if (c == OK) {
269                     m.destroyApp(false);
270                     m.notifyDestroyed();
271                 }
272             }
273         }
274       </xsl:if>
275     </xsl:for-each>
276
277     class AlertCanvas extends Canvas
278     {
279         private Display      display;
280         private Vector       pending = new Vector();
281         private Displayable  destination;
282         private Alert        current ;
283
284
```

```
285         public AlertCanvas()
286         {
287         }
288
289         protected void paint(Graphics g)
290         {
291             // no painting
292         }
293
294         protected synchronized void showNotify()
295         {
296             if (pending.size() > 0) {
297                 current = (Alert) pending.elementAt(0);
298                 pending.removeElementAt(0);
299                 display.setCurrent(current, this);
300             } else {
301                 current = null;
302                 display.setCurrent(destination);
303                 destination = null;
304             }
305         }
306
307         public synchronized void show(Alert alert)
308         {
309             pending.addElement(alert);
310
311             if ((current == null) &amp;amp;&amp;amp; (display != null)) {
312                current = (Alert) pending.elementAt(0);
313                display.setCurrent(this);
314             }
315         }
316
317         public synchronized void init(Display d)
318         {
319             this.display = d;
320
321             if (destination == null) {
322                 destination = display.getCurrent();
323
324                 if (destination == null) {
```

```
325                     destination = this;
326                 }
327             }
328
329             if (current == null) {
330                 if (!pending.isEmpty()) {
331 //                    current = (Alert) pending.elementAt(0);
332                     display.setCurrent(this);
333                 }
334             }
335         }
336     }
337 }
338   </xsl:template>
339
340
341   <xsl:template name="var_num">variable_<xsl:param name="varCount" select="
        count(preceding::parma:argType)+1"/><xsl:value-of select="$varCount"/></
        xsl:template>
342
343   <xsl:template match="parma:argType" name="processVars" mode="TypesAndVars">
344     <xsl:param name="N" select="1" />
345     <xsl:for-each select=".">
346       <xsl:value-of select="."/>
347       <xsl:text> </xsl:text>
348       <xsl:call-template name="var_num"/>
349     </xsl:for-each>
350     <xsl:if test="position()!=last()">
351       <xsl:text>, </xsl:text>
352     </xsl:if>
353   </xsl:template>
354
355   <xsl:template match="parma:argType" mode="typeOnly">
356     <xsl:param name="N" select="1" />
357     <xsl:for-each select=".">
358       <xsl:value-of select="."/>
359     </xsl:for-each>
360     <xsl:if test="position()!=last()">
361       <xsl:text>, </xsl:text>
362     </xsl:if>
```

```
363   </xsl:template>
364
365   <xsl:template match="parma:argType" mode="typeOnlyWithoutWhitespace">
366     <xsl:param name="N" select="1" />
367     <xsl:for-each select=".">
368       <xsl:value-of select="."/>
369     </xsl:for-each>
370     <xsl:if test="position()!=last()">
371       <xsl:text>,</xsl:text>
372     </xsl:if>
373   </xsl:template>
374
375   <xsl:template match="parma:argType" mode="varsOnly">
376     <xsl:param name="N" select="1" />
377     <xsl:for-each select=".">
378       <xsl:call-template name="var_num"/>
379     </xsl:for-each>
380     <xsl:if test="position()!=last()">
381       <xsl:text>, </xsl:text>
382     </xsl:if>
383   </xsl:template>
384
385   <xsl:template match="parma:package" mode="signature"><xsl:value-of select="
        parma:packageName"/>.<xsl:value-of select="parma:class/parma:className"
        />#<xsl:value-of select="parma:class/parma:method/parma:methodName"/>(<
        xsl:apply-templates select="parma:class/parma:method/parma:argType" mode=
        "typeOnly"/>)</xsl:template>
386
387 </xsl:stylesheet>
```

Listing B.2: XSLT Stylesheet for the J2SE Platform

```
1 ï■¿<?xml version="1.0" encoding="UTF-8"?>
2
3 <xsl:stylesheet version="1.0"
4   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
5   xmlns:o-dd="http://odrl.net/1.1/ODRL-DD"
6   xmlns:parma="http://www.dsg.cs.tcd.ie/parma/1.2"
7   xmlns:o-ex="http://odrl.net/1.1/ODRL-EX"
8   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

```
9
10
11    <xsl:output method="text"/>
12
13
14    <xsl:template match="/">
15 // ********************************************************************
16 //
17 // Generated by PARMA
18 //
19 // Copyright (c) 2003 - 2005
20 // Distributed Systems Group,
21 // Department of Computer Science
22 // Trinity College Dublin
23 // http://www.dsg.cs.tcd.ie
24 //
25 // All Rights Reserved
26 // ********************************************************************
27
28
29 import ie.tcd.persl.drma.context.DRMContext;
30
31 import ie.tcd.persl.drma.core.DRMEngine;
32 import ie.tcd.persl.drma.core.DRMException;
33
34 import java.util.Vector;
35
36 import java.awt.Container;
37 import java.awt.FlowLayout;
38 import java.awt.event.ActionEvent;
39 import java.awt.event.ActionListener;
40
41 import javax.swing.JButton;
42 import javax.swing.JFrame;
43 import javax.swing.JLabel;
44 import javax.swing.JOptionPane;
45
46      <xsl:variable name="countExecutes"   select="count(//o-ex:permission/o-dd:
            execute)"/>
47      <xsl:variable name="countPointcuts" select="count(//o-ex:permission/o-dd:
```

```
        execute/o-ex:constraint/parma:pointcut)"/>
48
49 /**
50  * The RightsAspect has been generated for Asset ID <xsl:value-of select="o-ex
        :rights/o-ex:agreement/o-ex:asset/@id"/>.
51  * Entry points into the DRMEngine are:<xsl:if test="number($countExecutes >
        $countPointcuts)">
52  * - MIDlet.startApp()s (Note: if you would not like to have all MIDlet.
        startApp automatically
53  *     included in the rights enforcement, you would need to specify each one
        you would like to
54  *     have individually!)</xsl:if>
55  * <xsl:for-each select="//parma:pointcut">
56  * - <xsl:apply-templates select="parma:package" mode="signature"/>
57     </xsl:for-each>
58  *
59  * This aspect has to be woven into the application which is intended
60  * for rights enforcement.
61  *
62  * @author Dominik Dahlem
63  * @author Ismet Celebi
64  */
65 public aspect RightsAspect
66 {
67     <xsl:for-each select="//parma:pointcut">
68       <xsl:variable name="returnType" select="parma:package/parma:class/parma:
            method/parma:returnType"/>
69       <xsl:variable name="method" select="parma:package/parma:class/parma:
            method/parma:methodName"/>
70       <xsl:variable name="class" select="parma:package/parma:class/parma:
            className"/>
71       <xsl:variable name="package" select="parma:package/parma:packageName"/>
72       <xsl:variable name="targetObjectParameter"><xsl:value-of select="
            $package"/>.<xsl:value-of select="$class"/> target</xsl:variable>
73       <xsl:variable name="argsTypesAndVars"><xsl:apply-templates select="parma
            :package/parma:class/parma:method/parma:argType" mode="TypesAndVars"
            /></xsl:variable>
74       <xsl:variable name="argsVarsOnly"><xsl:apply-templates select="parma:
            package/parma:class/parma:method/parma:argType" mode="varsOnly"/></
            xsl:variable>
```

```
75      <xsl:variable name="argsTypesOnly"><xsl:apply-templates select="parma:
            package/parma:class/parma:method/parma:argType" mode="typeOnly"/></
            xsl:variable>
76      <xsl:variable name="parameterList">
77        <xsl:choose>
78          <xsl:when test="string($argsTypesAndVars) = ''">
79            <xsl:value-of select="$targetObjectParameter"/>
80          </xsl:when>
81          <xsl:otherwise>
82            final <xsl:value-of select="$targetObjectParameter"/>, <xsl:value-
                  of select="$argsTypesAndVars"/>
83          </xsl:otherwise>
84        </xsl:choose>
85      </xsl:variable>
86      <xsl:variable name="parameterNameList">
87        <xsl:choose>
88          <xsl:when test="string($argsVarsOnly) = ''">target</xsl:when>
89          <xsl:otherwise>target, <xsl:value-of select="$argsVarsOnly"/></xsl:
                otherwise>
90        </xsl:choose>
91      </xsl:variable>
92      <xsl:variable name="adviceReturn">
93        <xsl:choose>
94          <xsl:when test="parma:adviceType = 'around'">
95            <xsl:value-of select="$returnType"/>
96          </xsl:when>
97          <xsl:otherwise>
98          </xsl:otherwise>
99        </xsl:choose>
100     </xsl:variable>
101
102   pointcut pcut_NoMain_<xsl:value-of select="$method"/>():
103       call (public static <xsl:value-of select="$returnType"/> <xsl:text> </
              xsl:text><xsl:value-of select="$package"/>.<xsl:value-of select="
              $class"/>.<xsl:value-of select="$method"/>(<xsl:value-of select="
              $argsTypesOnly"/>))
104       &amp;amp;amp; !call(* *..*.main(String[]))
105       &amp;amp;amp; !within(ie.tcd.persl..*);
106
107   <xsl:value-of select="$adviceReturn"/><xsl:text> </xsl:text><xsl:value-of
```

```
        select="parma:adviceType"/>():
108     pcut_NoMain_<xsl:value-of select="$method"/>()
109     {
110         // instantiate the context for this method invocation
111         DRMContext context = new DRMContext();
112
113         context.setPackage("<xsl:value-of select="$package"/>");
114         context.setClazz("<xsl:value-of select="$class"/>");
115         context.setMethod("<xsl:value-of select="$method"/>");
116         context.setParameters("<xsl:apply-templates select="parma:package/
                parma:class/parma:method/parma:argType" mode="
                typeOnlyWithoutWhitespace"/>");
117
118         // enforce the rights in a given context
119         try {
120             DRMEngine.newInstance().enforce(context);
121     <xsl:if test="parma:adviceType = 'around'">
122             proceed();
123     </xsl:if>
124         } catch (DRMException drme) {
125             ParmaWarning alert = new ParmaWarning(drme.getMessage());
126             JButton closeButton = new JButton("OK");
127             JButton upgradeButton = new JButton("Upgrade");
128             closeButton.addActionListener(new ActionListener() {
129                 // process text field events
130                 public void actionPerformed(ActionEvent event) {
131 //                    if ("OK".equals(event.getActionCommand())) {
132                       // change this to return an error code.
133 //                    }
134                 }
135             });
136             upgradeButton.addActionListener(new ActionListener() {
137                 // process text field events
138                 public void actionPerformed(ActionEvent event) {
139 //                    if ("Upgrade".equals(event.getActionCommand())) {
140                         // change this to return an error code.
141                         JOptionPane.showMessageDialog(
142                         null,
143                         "You pressed: " + event.getActionCommand(),
144                         "PARMA Rights Managment",
```

```
145                             1);
146 //                      }
147                     }
148                 });
149
150                 alert.addButton(closeButton);
151                 alert.addButton(upgradeButton);
152                 alert.setDefaultCloseOperation( JFrame.EXIT_ON_CLOSE );
153                 alert.setVisible(true);
154             }
155         }
156
157         /**
158          * Pointcut describing a execution joinpoint which is a main method
159          * which is not called within the package ie.tcd.persl packages.
160          */
161         pointcut pcut_Main_<xsl:value-of select="$method"/>():
162             execution (public static <xsl:value-of select="$returnType"/> <xsl:
                    text></xsl:text><xsl:value-of select="$package"/>.<xsl:value-of
                    select="$class"/>.<xsl:value-of select="$method"/>(<xsl:value-of
                    select="$argsTypesOnly"/>))
163             &amp;&amp; execution(* *..*.main(String[]))
164             &amp;&amp; !within(ie.tcd.persl..*);
165
166         <xsl:value-of select="$adviceReturn"/><xsl:text></xsl:text><xsl:value-of
                select="parma:adviceType"/>():
167         pcut_Main_<xsl:value-of select="$method"/>()
168         {
169             // instantiate the context for this method invocation
170             DRMContext context = new DRMContext();
171
172             context.setPackage("<xsl:value-of select="$package"/>");
173             context.setClazz("<xsl:value-of select="$class"/>");
174             context.setMethod("<xsl:value-of select="$method"/>");
175             context.setParameters("<xsl:apply-templates select="parma:package/
                        parma:class/parma:method/parma:argType" mode="
                        typeOnlyWithoutWhitespace"/>");
176
177             // enforce the rights in a given context
178             try {
```

```
179             DRMEngine.newInstance().enforce(context);
180     <xsl:if test="parma:adviceType = 'around'">
181             proceed();
182     </xsl:if>
183         } catch (DRMException drme) {
184             ParmaWarning alert = new ParmaWarning(drme.getMessage());
185             JButton closeButton = new JButton("OK");
186             JButton upgradeButton = new JButton("Upgrade");
187             closeButton.addActionListener(new ActionListener() {
188                 // process text field events
189                 public void actionPerformed(ActionEvent event) {
190 //                    if ("OK".equals(event.getActionCommand())) {
191                       // change this to return an error code.
192                       System.exit(0);
193 //                    }
194                 }
195             });
196             upgradeButton.addActionListener(new ActionListener() {
197                 // process text field events
198                 public void actionPerformed(ActionEvent event) {
199 //                  if ("Upgrade".equals(event.getActionCommand())) {
200                       // change this to return an error code.
201                       JOptionPane.showMessageDialog(
202                       null,
203                       "You pressed: " + event.getActionCommand(),
204                       "PARMA Rights Managment",
205                       1);
206 //                   }
207                 }
208             });
209
210             alert.addButton(closeButton);
211             alert.addButton(upgradeButton);
212             alert.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
213             alert.setVisible(true);
214         }
215     }
216     </xsl:for-each>
217
218     public class ParmaWarning extends JFrame
```

```
219     {
220         private JButton closeButton;
221         private JButton upgradeButton;
222         private JLabel textLabel;
223         private Container container;
224 
225         public ParmaWarning(String message)
226         {
227             super("PARMA - Rights Enforcement");
228 
229             //get content pane and set its layout
230             container = getContentPane();
231             container.setLayout( new FlowLayout() );
232 
233             textLabel = new JLabel(message);
234             container.add(textLabel);
235             setSize(300, 100);
236             setVisible(false);
237         }
238 
239         public void addButton(JButton button) {
240             container.add(button);
241         }
242     }
243 }
244   </xsl:template>
245 
246 
247   <xsl:template name="var_num">variable_<xsl:param name="varCount" select="
        count(preceding::parma:argType)+1"/><xsl:value-of select="$varCount"/></
        xsl:template>
248 
249   <xsl:template match="parma:argType" name="processVars" mode="TypesAndVars">
250     <xsl:param name="N" select="1" />
251     <xsl:for-each select=".">
252       <xsl:value-of select="."/>
253       <xsl:text></xsl:text>
254       <xsl:call-template name="var_num"/>
255     </xsl:for-each>
256     <xsl:if test="position()!=last()">
```

```
257       <xsl:text>, </xsl:text>
258     </xsl:if>
259   </xsl:template>
260
261   <xsl:template match="parma:argType" mode="typeOnly">
262     <xsl:param name="N" select="1" />
263     <xsl:for-each select=".">
264       <xsl:value-of select="."/>
265     </xsl:for-each>
266     <xsl:if test="position()!=last()">
267       <xsl:text>, </xsl:text>
268     </xsl:if>
269   </xsl:template>
270
271   <xsl:template match="parma:argType" mode="typeOnlyWithoutWhitespace">
272     <xsl:param name="N" select="1" />
273     <xsl:for-each select=".">
274       <xsl:value-of select="."/>
275     </xsl:for-each>
276     <xsl:if test="position()!=last()">
277       <xsl:text>,</xsl:text>
278     </xsl:if>
279   </xsl:template>
280
281   <xsl:template match="parma:argType" mode="varsOnly">
282     <xsl:param name="N" select="1" />
283     <xsl:for-each select=".">
284       <xsl:call-template name="var_num"/>
285     </xsl:for-each>
286     <xsl:if test="position()!=last()">
287       <xsl:text>, </xsl:text>
288     </xsl:if>
289   </xsl:template>
290
291   <xsl:template match="parma:package" mode="signature"><xsl:value-of select="
        parma:packageName"/>.<xsl:value-of select="parma:class/parma:className"
        />#<xsl:value-of select="parma:class/parma:method/parma:methodName"/>(<
        xsl:apply-templates select="parma:class/parma:method/parma:argType" mode=
        "typeOnly"/>)</xsl:template>
292
```

```
293 </xsl:stylesheet>
```

Bibliography

[1] Apache Maven Homepage. URL http://maven.apache.org/. last accessed in March 2005.

[2] Apache WebServices Project. URL http://ws.apache.org/axis/index.html. last accessed in March 2005.

[3] AspectJ - Crosscutting objects for better modularity. URL http://eclipse.org/aspectj/. last accessed in March 2005.

[4] Axis2. URL http://ws.apache.org/axis2/. last accessed in March 2005.

[5] Community Development of Java Technology Specifications. URL http://www.jcp.org. last accessed in March 2005.

[6] Concurrent Versions System (CVS) Homepage. URL http://www.cvshome.org/. last accessed in March 2005.

[7] CruiseControl Homepage. URL http://cruisecontrol.sourceforge.net/. last accessed in March 2005.

[8] Digital Object Identifier System. URL http://www.doi.org. last accessed in March 2005.

[9] Federation of European Publishers Homepage. URL http://www.fep-fee.be/. last accessed in March 2005.

[10] Filigrane Homepage. URL http://www.filigrane.org. last accessed in March 2005.

[11] Hivemind Homepage. URL http://jakarta.apache.org/hivemind/. last accessed in March 2005.

[12] J2ME Homepage. URL http://java.sun.com/j2me/. last accessed in March 2005.

[13] Java Virtual Machine Profiler Interface (JVMPI). Technical documentation. URL http://java.sun.com/j2se/1.5.0/docs/guide/jvmpi/jvmpi.html. last accessed in March 2005.

[14] JCoverage Homepage. URL http://www.jcoverage.com. last accessed in March 2005.

[15] JSR 124: J2EE Client Provisioning Specification. URL http://www.jcp.org/en/jsr/detail?id=124. last accessed in March 2005.

[16] JSR 177: Security and Trust Services API for J2ME. URL http://www.jcp.org/en/jsr/detail?id=177. last accessed in March 2005.

[17] kSOAP 2 Project Homepage. URL http://kobjects.sourceforge.net/ksoap2/. last accessed in March 2005.

[18] kXML Project Homepage. URL http://kxml.sourceforge.net/. last accessed in March 2005.

[19] Macrovision Homepage. URL http://www.macrovision.com. last accessed in March 2005.

[20] Maven-Plugins Homepage. URL http://maven-plugins.sf.net/. last accessed in March 2005.

[21] The Obje Software Architecture. URL http://www.parc.com/research/csl/projects/obje/. last accessed in March 2005.

[22] OMA - Open Mobile Alliance Homepage. URL http://www.openmobilealliance.org/. last accessed in March 2005.

[23] OMA Digital Rights Management V2.0 Candidate Enabler. URL http://www.openmobilealliance.org/release_program/drm_v20.html. last accessed in March 2005.

[24] Open Digital Rights Language (ODRL). URL http://www.w3.org/TR/odrl. last accessed in March 2005.

[25] Organisation Internationale de Normalisation ISO/IEC JTC1/SC29/WG11 Coding of Moving Pictures and Audio. URL http://www.chiariglione.org/mpeg/standards/mpeg-21/mpeg-21.htm. last accessed in March 2005.

[26] Picocontainer Homepage. URL http://www.picocontainer.org. last accessed in March 2005.

[27] Security Systems Standards and Certification Act Draft. URL http://cryptome.org/sssca.htm. last accessed in March 2005.

[28] Simian (Similarity Analyser). URL http://simian.dev.java.net/. last accessed in March 2005.

[29] Simple API for XML. URL http://www.saxproject.org/. last accessed in March 2005.

[30] SOAP Version 1.2 Part 1: Messaging Framework. URL http://www.w3.org/TR/soap12-part1/. last accessed in March 2005.

[31] Spring Framework Homepage. URL http://www.springframework.org/. last accessed in March 2005.

[32] Symbian OS Homepage. URL http://www.symbian.com/. last accessed in March 2005.

[33] Uniform Resource Identifiers (URI): Generic Syntax, IETF RFC 2396. URL http://www.ietf.org/rfc/rfc2396.txt. last accessed in March 2005.

[34] vCard profile, IETF RFC 2426. URL http://www.ietf.org/rfc/rfc2426.txt. last accessed in March 2005.

[35] XML Encryption Syntax and Processing. URL http://www.w3.org/TR/xmlenc-core/. last accessed in March 2005.

[36] XML Pull Parsing. URL http://www.xmlpull.org/. last accessed in March 2005.

[37] XML Schema. URL http://www.w3.org/XML/Schema. last accessed in March 2005.

[38] XML-Signature Syntax and Processing. URL http://www.w3.org/TR/xmldsig-core/. last accessed in March 2005.

[39] XrML. URL http://www.xrml.org/. last accessed in March 2005.

[40] XSL Transformations (XSLT). URL http://www.w3.org/TR/xslt. last accessed in March 2005.

[41] Systems Management: Software License Use Management (XSLM). Technical standard, The Open Group, March 1999.

[42] Digital Rights Management. Technical standard, CEN/ISSS (European Committee for Standardization / Information Society Standardization System), September 2003. URL http://europa.eu.int/comm/enterprise/ict/policy/doc/drm.pdf. last accessed in March 2005.

[43] Michel Barbeau. *Handbook of Wireless Networks and Mobile Computing*, chapter 27. John Wiley & Sons, Inc., 2002. ISBN B0001ZYOZQ.

[44] Edward Barrow. Tools and standards for protection, control and presentation of data: Rights clearance and technical protection in an electronic environment. 1996. URL http://www.library.uiuc.edu/icsu/barrow.htm. last accessed in March 2005.

[45] Dan Boneh and James Shaw. Collusion-Secure Fingerprinting for Digital Data. In *Advances in Cryptology - CRYPTO '95: 15th Annual International Cryptology Conference*, volume 963 of *LNCS*, pages 452–465. Springer-Verlag GmbH, 1995.

[46] Oliver Burn. Checkstyle Homepage. URL http://checkstyle.sourceforge.net/. last accessed in March 2005.

[47] Shigeru Chiba. Load-Time Structural Reflection in Java. In *ECOOP 2000 - Object-Oriented Programming: 14th European Conference, Sophia Antipolis and Cannes, France, June 2000. Proceedings*, pages 313–336. Springer-Verlag Berlin Heidelberg, June 2000.

[48] Mike Clark. JDepend. URL http://www.clarkware.com/software/JDepend.html. last accessed in March 2005.

[49] Mike Clark. *Pragmatic Project Automation: How to Build, Deploy and Monitor Java Applications*. The Pragmatic Programmers, 2004. ISBN 0974514039.

[50] Nick Cook. Copyright in transmitted electronic documents (CITED), 1990. URL `http://www.newcastle.research.ec.org/esp-syn/text/5469.html`. last accessed in March 2005.

[51] Ingemar Cox, Joe Kilian, Tom Leighton, and Talal Shamoon. Secure Spread Spectrum Watermarking for Multimedia. *IEEE Transactions on Image Processing*, 6(12):1673–1687, 1997.

[52] Dominik Dahlem, Ivana Dusparic, and Jim Dowling. A Pervasive Application Rights Management Architecture (PARMA) based on ODRL. In *Proceedings of the First International Workshop on the Open Digital Rights Language (ODRL)*, pages 45–63, 2004. ISBN 1-74064-500-6.

[53] Premkumar T. Devanbu and Stuart Stubblebine. Software engineering for security: a roadmap. In *ICSE '00: Proceedings of the Conference on The Future of Software Engineering*, pages 227–239, New York, NY, USA, 2000. ACM Press. ISBN 1-58113-253-0.

[54] Screen Digest. Mobile gaming gets its skates on. *The Register*, February 2005. URL `http://www.theregister.com/2005/02/09/mobile_gaming_analysis/`. last accessed in March 2005.

[55] Ivana Dusparic. PARMA - A Pervasive Application Rights Management Architecture. Master's thesis, Trinity College Dublin, Computer Science Department, Distributed Systems Group, October 2004.

[56] Ivana Dusparic, Dominik Dahlem, and Jim Dowling. Flexible Application Rights Management in a Pervasive Environment. In *2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE 2005)*, pages 680–685. IEEE Computer Society Press, 2005.

[57] Faultline. Mobile DRM levy hits operators where it hurts. *The Register*, January 2005. URL `http://www.theregister.co.uk/2005/01/19/mobile_drm_levy/`. last accessed in March 2005.

[58] Marc Fetscherin. Present State and Emerging Scenarios of Digital Rights Management Systems. *The International Journal on Media Management*, 4(3), February 2002. URL `http://www.mediajournal.org/modules/pub/view.php/mediajournal-90`. last accessed in March 2005.

[59] Norm Friesen, Magda Mourad, and Robby Robson. Towards a Digital Rights Expression Language Standard for Learning Technology. Technical report, IEEE, 2002. URL `http://xml.coverpages.org/DREL-DraftREL.pdf`. last accessed in March 2005.

[60] Kurt Gabrick and David B. Weiss. *J2EE and XML Development*. Manning, 2002. ISBN 1930110308.

[61] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995. ISBN 0201633612.

[62] Marie-Pierre Gervais. Towards an MDA-Oriented Methodology. In *COMPSAC '02: Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment*, pages 265–270. IEEE Computer Society, 2002. ISBN 0-7695-1727-7.

[63] Eric Giguere. Record Management System Basics. Technical report, SUN Microsystems, 2001. URL `http://developers.sun.com/techtopics/mobility/midp/ttips/rmsbasics/`. last accessed in March 2005.

[64] Susanne Guth, Gustaf Neumann, and Mark Strembeck. Experiences with the enforcement of access rights extracted from ODRL-based digital contracts. In *DRM '03: Proceedings of the 2003 ACM workshop on Digital rights management*, pages 90–102. ACM Press, 2003. ISBN 1-58113-786-9.

[65] Gael Hachez, Laurent Den Hollander, Mehrdad Jalali, Jean-Jacques Quisquater, and Christophe Vasserot. Towards a Practical Secure Framework for Mobile Code Commerce. In *Information Security: Third International Workshop, ISW 2000, Wollongong, Australia, December 2000*, volume 1975 of *LNCS*, pages 164–178. Springer-Verlag GmbH, 2000.

[66] Philippe Le Hegaret. Document Object Model (DOM), January 2005. URL `http://www.w3.org/DOM/`. last accessed in March 2005.

[67] Richard Hightower, Warner Onstine, Paul Visan, Damon Payne, and Joseph D. Gradecki. *Professional Java Tools for Extreme Programming: Ant, Xdoclet, Junit, Cactus and Maven*. Hungry Minds Inc, 2004. ISBN 0764556177.

[68] David Hovemeyer and William Pugh. Finding bugs is easy. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 132–136. ACM Press, 2004. ISBN 1-58113-833-4.

[69] Jason Hunter and William Crawford. *Java Servlet Programming*. O'Reilly & Associates, Inc., 2001. ISBN 0596000405.

[70] Renato Iannella and Peter Higgs. Driving Content Management with Digital Rights Management. Whitepaper, September 2003.

[71] Mehrdad Jalali. An integrated secure Web architecture for protected mobile code distribution. In *Proceedings of the IFIP TC6/TC11 International Conference on Communications and Multimedia Security Issues of the New Century*, page 6. Kluwer, B.V., 2001. ISBN 0-7923-7365-0.

[72] Rod Johnson and Juergen Hoeller. *J2EE Development without EJB*. Wiley Publishing, June 2004. ISBN 0764558315.

[73] Guy Kewney. Landscape fills with PDA smart phones. *The Register*, February 2005. URL `http://www.theregister.co.uk/2005/02/09/landscape_smartphones/`. last accessed in March 2005.

[74] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP 2001 - Object-Oriented Programming: 15th European Conference, Budapest, Hungary, June 18-22, 2001, Proceedings*, volume 2072, pages 327–353. Springer-Verlag Berlin Heidelberg, June 2001.

[75] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN 032119442X.

[76] Butler Lampson, Martin Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: theory and practice. *ACM Trans. Comput. Syst.*, 10(4):265–310, 1992. ISSN 0734-2071.

[77] C. Clemens Lee. JavaNCSS - A Source Measurement Suite for Java. URL `http://www.kclee.de/clemens/java/javancss/`. last accessed in March 2005.

[78] Rosanna Lee and Scott Seligman. *The JNDI API Tutorial and Reference: Building Directory-Enabled Java Applications*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 0201705028.

[79] John Lettice. Guilty until proven innocent - DRM the mobile phone way. *The Register*, July 2004. URL `http://www.theregister.co.uk/2004/07/15/oma_drm_for_phones/`. last accessed in March 2005.

[80] Yin-Ling Liong and Sudhir Dixit. Digital Rights Management for the Mobile Internet. *Wireless Personal Communications: An International Journal*, 29(1-2):109–119, 2004. ISSN 0929-6212.

[81] Umesh Maheshwari, Radek Vingralek, and Bill Shapiro. How to Build a Trusted Database System on Untrusted Storage. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)"*, pages 135–150, San Diego, CA, October 2000.

[82] Qusay H. Mahmoud. Understanding Network Class Loaders. Whitepaper, October 2004.

[83] Mark S. Manasse. Why Rights Management is Wrong (and What to Do Instead). World Wide Web Consortium, January 2001. URL `http://www.w3.org/2000/12/drm-ws/pp/compaq.html`. last accessed in March 2005.

[84] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 1st edition, October 2002. ISBN 0135974445.

[85] Vincent Massol and Ted Husted. *JUnit in Action*. Manning Publications Co., 2003. ISBN 1930110995.

[86] Vincent Massol and Timothy M. O'Brien. *Maven a Developer's Notebook*. O'Reilly, 2005. ISBN 0596007507.

[87] Open Mobile Alliance. Digital Rights Management Version 1.0, June 2004. URL `http://www.openmobilealliance.org/release_program/drm_v10.html`. last accessed in March 2005.

[88] Akito Monden, Hajimu Iida, Ken ichi Matsumoto, Koji Torii, and Katsuro Inoue. A practical method for watermarking java programs. In *COMPSAC '00: 24th International Computer Software and Applications Conference*, pages 191–197, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0792-1.

[89] Richard Monson-Haefel, Bill Burke, and Sacha Labourey. *Enterprise JavaBeans*. O'Reilly UK, 2004. ISBN 059600530X.

[90] Theodor Nelson. *Literary Machines*. Mindful Press, 1982. ISBN 0-89347-062-7.

[91] Andrew Orlowski. Symbian updates OS, toolchain. *The Register*, February 2005. URL `http://www.theregister.co.uk/2005/02/02/symbian_os_9/`. last accessed in March 2005.

[92] Shuping Ran, Paul Brebner, and Ian Gorton. The Rigorous Evaluation of Enterprise Java Bean Technology. In *ICOIN '01: Proceedings of the The 15th International Conference on Information Networking*, page 93, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-0951-7.

[93] Vladimir Roubtsov. Cracking Java byte-code encryption. URL `http://www.javaworld.com/javaqa/2003-05/01-qa-0509-jcrypt_p.html`. last accessed in March 2005.

[94] Nuno Santos, Pedro Pereira, and Luis Moura e Silva. A Generic DRM Framework for J2ME Applications. In *Proceedings of the First International Mobile IPR Workshop: Rights Management of Information Products on the Mobile Internet*, pages 53–66, 2003.

[95] William Shapiro and Radek Vingralek. How to Manage Persistent State in DRM Systems. In *Digital Rights Management Workshop*, pages 176–191, 2001.

[96] Aleksander Slominski. Design of a Pull and Push Parser System for Streaming XML. Technical report, Indiana University Computer Science Department, 2002. URL `http://www.extreme.indiana.edu/xgws/papers/xml_push_pull/`. last accessed in March 2005.

[97] Tony Smith. Smart phone shipments break records. *The Register*, February 2005. URL `http://www.theregister.co.uk/2005/02/01/mobile_devices_q4_04/`. last accessed in March 2005.

[98] RoleModel Software. J2MEUnit. URL `http://j2meunit.sourceforge.net/`. last accessed in March 2005.

[99] Joao Pedro Sousa and David Garlan. Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments. In *WICAS3: Proceedings of the IFIP 17th World Computer Congress - TC2 Stream / 3rd IEEE/IFIP Conference on Software Architecture*, pages 29–43. Kluwer, B.V., 2002. ISBN 1-4020-7176-0.

[100] Diomidis Spinellis. On the Declarative Specification of Models. *IEEE Software*, 20(2):94–96, March/April 2003.

[101] Douglas Stinson. *Cryptography: Theory and Practice (Discrete Mathematics & Its Applications S.)*. CRC Press, 2002. ISBN 1584882069.

[102] Mark Strembeck and Gustaf Neumann. An integrated approach to engineer and enforce context constraints in RBAC environments. *ACM Trans. Inf. Syst. Secur.*, 7(3):392–427, 2004. ISSN 1094-9224.

[103] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, November 2002. ISBN 0201745720.

[104] Matt Volpi. Nokia DRM tools. In *Nokia Media & Music Digital Rights Management Workshop 2003*, November 2003.

[105] Norm Walsh and Leonard Muellner. *DocBook: The Definitive Guide*. O'Reilly & Associates, Inc., 1999. ISBN 1565925807.

[106] Wireless Watch. 3g success hangs on handsets. *The Register*, February 2005. URL `http://www.theregister.co.uk/2005/02/07/3g_success/`. last accessed in March 2005.

[107] Mark Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–104, September 1991.

[108] Tom Wheeler. Improving Project Quality with PMD. URL `http://www.ociweb.com/jnb/jnbJun2004.html`. last accessed in March 2005.

[109] Michael Juntao Yuan. *Enterprise J2ME Wireless Applications*. Prentice Hall PTR, December 2003. ISBN 0131405306.

[110] Juergen Zimmermann and Gerd Beneken. *Verteilte Komponenten und Datenbankanbindung. Mehrstufige Architekturen mit SQLJ und Enterprise JavaBeans 2.0*. Addison-Wesley, Oktober 2000. ISBN 3827315522.