Attribute Enabled Software Development

(Illustrated with Java ME mobile software applications)

Product-Line Development
for Mobile Device Applications
with Attribute Supported Containers

Vasian Cepa

Contents

List of Figures

List of Tables

Acknowledgments

This work has been mainly supported by a scholarship of Deutsche Forschungsgemeinschaft (DFG) as part of the Graduiertenkolleg "Systemintegration für Ubiquitäres Rechnen in der Informationstechnik" (GK 749), headed by Prof. Dr. Dr. h.c. mult. Manfred Glesner.

This book has been based on a revised version of my Ph.D. thesis done at the Software Technology Group, at Darmstadt University of Technology and most of my acknowledgments go to the people that made the thesis possible. Doing a Ph.D. is a long project that one has to go through alone. However, no project of this size is possible without the help, valuable advice, and comments from a lot of other people. Especially my supervisor, Prof. Dr. Mira Mezini, provided continuous advice and reviewed parts of the material more than once. Her insistence that I care that the others understand what I say, has greatly contributed to the quality of this book. Finishing the dissertation would not have been possible without her continuous support.

The adventure of doing a Ph.D. started for me in 2000, when I went to the University of Siegen as part of a DAAD[1] project, made possible by Prof. Dr. Betim Çiço and Dr. Jochen Münch, while I was waiting to obtain a working visa for the USA (thanks to Dr. Patrick Ball). The visa took longer than expected, and my professor in Siegen, Prof. Dr. Bernd Freisleben, *somehow* convinced me to drop (in agreement with the employer - thanks to Marc Beneteau) the work possibility I had for the US, and start the research work toward a Ph.D. in Germany instead. I am grateful now to Prof. Dr. Freisleben for his support and for helping me to apply for a DFG[2] scholarship at the Darmstadt University of Technology (TUD). Without his initial advice and support my Ph.D. would not exist.

[1] Deutscher Akademischer Austauschdienst / German Academic Exchange Service, `http://www.daad.de`

[2] Deutschen Forschungsgemeinschaft / German Research Foundation, `http://www.dfg.de/`

I am also grateful to all my former colleagues at the Software Technology Group at the TUD, especially to Sven Kloppenburg, Dr. Michael Eichberg, Dr. Klaus Ostermann, Dr. Michael Haupt, Vaidas Gasiunas, Christoph Bockisch, and Thorsten Schäfer, for their comments, reviews, and discussions. Eniana Mustafaraj and Elvis Papalilo from the University of Marburg reviewed some of the early drafts. Oliver Liemert implemented most of the prototype container framework for J2ME as part of his master diploma thesis. The anonymous reviewers of my conference and journal papers, upon which most parts of my thesis were based, provided many valuable comments and insights. I also want to thank Prof. Dr. Uwe Aßmann for accepting to serve as co-reviewer of my thesis and for his comments on chapter §4.

While it is difficult to write a technical document that everyone can understand, I hope most of the readers will enjoy the topics addressed in this book. Usually a Ph.D. is evaluated for the new contribution that it brings in a very narrow area of science. However, on the personal level the real value of a Ph.D. is the gained overall knowledge. I used the time and the university environment not only to contribute, but also to learn new things every day. The papers and the books I have read, including a very interesting one from the co-reviewer of my thesis, Prof. Dr. Uwe Aßmann, the international conferences I have attended, the people I have met, the presentations I have seen, and the *Graduiertenkolleg* I was part of, have enabled me to obtain a deeper understanding of the technologies discussed in this book. I am also thankful to Prof. Dr. Alejandro Buchmann for his guidance during the Graduiertenkolleg workshops. I would like to thank the additional members of the thesis Examination Committee, Prof. Dr. Thomas Kühne, and Prof. Dr. Andy Schürr for their reviews and comments. Many thanks go also to Gudrun Jörs for helping me to organize a nice final buffet and for her administrative support during my time in Darmstadt.

Last but not least, I want to thank *Ana*, for her unconditional support during the extremely stressful and energy draining years of my PhD. I am not sure whether I would have survived finishing my work without her. She has been simply living for me, and there are no words that I can find to describe my endless gratitude to her.

Finally, I want thank my publisher Bettina Wahlen from VDM Verlag Dr. Müller e.K (VDM Publishing Hause) that made this book possible for a larger audience.

Würzburg, 2007

Chapter 1

Introduction

Computations across mappings are called transformations.

J. Greenfield, K. Short, S. Cook, S. Kent, Software Factories, Willey, 2004

Attribute Enabled Programming (AEP) is a wide-spreading technique that uses explicit attributes [BCVM02] directly in the programming language level to decorate code existing entities, in order to modify their semantics. AEP is supported in several general purpose languages, such as .NET (attributes) [Lib01], or Java (annotations) [JSR03]. AEP is also supported by modeling languages, such MOF, UML (tagged values, stereotypes) [Met02]. AEP enables developers to enhance the semantics of an existing language with embedded domain specific language constructs, without any need to change the language grammar. Developers can use AEP to customize a language to fit their needs, without having to worry about parsing and grammar rules.

AEP is not a mainstream methodology, rather it complements ubiquitously existing programming methodologies. It can be appended in the top of any existing programming model (methodology), such as, *object-oriented programming* (OOP), or *aspect-oriented programming* (AOP) [KLM+97]. Hence, the name Attribute Enabled Programming is preferred, and not something else, such as, Attribute Oriented Programming.

As AEP becomes more widely used, it can be easily overused and attribute annotations may fail to scale. Attributes are very easy to introduce, and depending on the automation as-

sociated with them, cheap to process. This could result in a large number of attributes used in unstructured ways within a project, or, in the general case, within a product-line made up of several related projects. The interpretation of many inter-related attributes could also become a bottleneck. Attribute dependencies further complicate the interpretation and add-hoc programming solutions cannot scale. A well defined development methodology that covers all the steps of *Attribute Enabled Software Development* (AESD) needs to be applied.

This book addresses AESD for mobile software applications and shows that a software abstraction, called attribute-supported generative container, when directly supported at the programming language level, provides an effective means to automate the development of product-line software that runs on mobile devices. This chapter presents the book in a nutshell. The need to automate the development of software for mobile devices is motivated by its role, extensive spread, and ever increasing complexity, in a world of ubiquitous computing. A short overview of the current technology for automation in the domain of mobile software is presented, justifying the need for advancing the technology, followed by a high-level presentation of the proposed technology. The chapter ends with a brief overview of the contribution of this book and with a preview of its structure.

1.1 Mobile Software in a Ubiquitous Computing World

The term *ubiquitous computing* (*ubicom* for short) was first coined in the beginning of the '90s in a series of articles by Mark Weiser [Wei91, Wei93, Wei94], while he was heading the Computer Science Laboratory at Xerox PARC. Ubiquitous means present or found everywhere, i.e., omnipresent[1], and implies transparency, i.e., the interaction with the computers is not explicit anymore. Computers become an integral part of the entire environment. Weiser says that *"The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."* [Wei91].

The ubiquitous computing ideas have emerged from the observation that the technology is becoming mature and cheap enough to be multiplied widely in the environment. The computer hardware is becoming less expensive and its size smaller as the number of transistors per chip grows. It could be afforded to have many specialized computers around: *"Ubiquitous computers will also come in different sizes, each suited to a particular task."* [Wei91].

The current trend of ubiquitous computing ideas is often labeled by the term *ambient intelligence* (AmI) [RVDA01, LMB+03], technically defined as a combination of *"Ubiquitous Computing, Ubiquitous Communication and Intelligent User Interfaces"* [RVDA01]. The *ambient* is the surrounding environment and often the term is used interchangeably with *environment*. While ubiquitous computing means more than one computer, ubiquitous communication means that computers do not live in isolation, but somehow communicate with each-other. Intelligent

[1] All definitions are based on Oxford English Dictionary (OED), `http://dictionary.oed.com`

user interfaces stand for personalized interaction with a ubiquitous computing environment in natural human ways, such as, voice (speech), or gestures. AmI is about combining different technologies toward a visionary goal, to enable various services and properties that the surrounding environment should have.

Ubiquitous computing has also been treated as a combination of:

- *natural interfaces*, another name for implicit and transparent human-computer interfaces,
- *context-aware applications*, related to the location, object identity, time history, user identification, etc. (who, what, where, when, why),
- *automated capture and access systems*, that enable recording and finding live experiences, e.g., meeting notes, and
- *everyday computing*, that addresses ubiquitous tasks that have a continuous timespan with no known end and that span through various concurrent activities [AM00].

There are two ways (Figure 1.1) which can be used in isolation but more often in combination to implement ubicom scenarios:

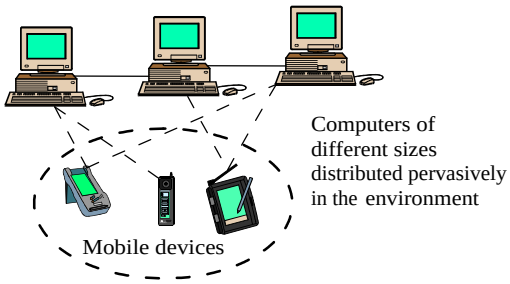


Figure 1.1: Pervasive and Mobile Devices

a. The technology can be *pervasive*, that is, the computers are extended and diffused throughout every part of the environment. In a pervasive environment, the computational resources will tend to be uniformly distributed among the objects. The objects will contain a combination of sensors, and / or *embedded* computers, and communication capabilities with humans (interface) or other computers. The objects will look like autonomous individuals, and every object will have its own processing unit, where it autonomously keeps track of the surrounding environment. That is, each object should have its own computer, its own sensors, and its own (unique) ways to communicate with and affect the rest of the environment. Depending on the context [SAW94], that is, the object location in the environment [HB01], each object

will create its own unique understanding of the environment and its history, and decide how to affect the environment in the future. In practice, some global view of the environment, along with the distribution of computational resources, makes it possible to optimize the coordination of object reactions (better context management). At present, it is also cheaper and easier to spread only sensors and the communication hardware, e.g., using the *smart dust* technology [KKP99, WLLP01], and keep some of the event processing centralized.

b. The technology can be *mobile*, that is, not stationary but movable around the environment, usually in the form of a personal computing device, or a *wearable* computer [Tho98]. It is the interaction with a ubicom environment, where the *mobile* view of the technology becomes important. Suitable forms of interaction are needed to control a ubicom environment [AM00]. Computers create a virtual model of the environment and operate in this model. The real world events should be converted to this virtual model, used to modify the state of the virtual model, and then the virtual model changes should be converted back to events than affect the real world. Human communication with a ubicom environment is part of this translation of the real world events.

   Mobile devices offer convenient and suitable ways to map human interactions in a ubiquitous environment and are often an integral part of ubicom scenarios. Using a mobile device is simpler that moving to reach a touch screen nearby. There is no need to stay in a queue to use a mobile device, as it would be the case with a shared device when there are other people in the same place. Using mobile devices for interaction, also does not conflict with using other forms of communication, e.g., voice and gestures, but can be used to augment them. There are also several other reasons why mobile computing and mobile applications are important for achieving ubiquitous computing.

   The non-uniform distribution of technology is likely to prevail. Not all environments will be equipped in the same ways. A mobile device that people can carry with, is a constant factor to rely upon, when designing applications in a non-uniform ubicom environment. It is cheaper to carry the technology around (e.g., a mobile phone), rather than replicate it in every corner (e.g., a public telephone cell every two meters). Even if a completely pervasive view of technology becomes possible and uniformly distributed, the mobile alternative pollutes the environment less with technology. Last but not least, a mobile device is a form of personal possession that can be carried always with. It may be used even if no networking is present. It makes sense to store data in a mobile device which are very personal, but difficult to remember, e.g., encrypted lists of passwords, financial records, and software that may be continuously needed.

   Mobile phones were the first mobile devices to have global success and wide acceptance. Based on this success, there is a trend to enrich the number of services that are offered to people via mobile phones, benefiting from an extensive existing user base. The demand for more functionality is followed by a competitive supply of better hardware [J2M02a] and software for mobile devices. Mobile phones and mobile computers (Personal Device Assistants

- PDAs) are becoming more powerful for less physical space. Both classes of devices are converging to a single device that has a telecommunication (networking) function and also serves as a personal computer device. This has opened a profitable market for third-party software for mobile devices supported by several platforms, e.g., J2ME [Jav05], .NET CF [Mic05], and BREW [QAL05], as well as frameworks [EBM05], followed by a need for more applications that should be delivered in time.

1.2 Automating Mobile Software Development

Mobile devices and their applications play an important and increasing role in ubiquitous computing. There are several non-functional ubicom challenges [BB02] related to software infrastructure, e.g., the way to build and organize the application functionality, seamless migration of logic among devices and different environments, device software modeling, and scalability of the solutions. To address these issues, the status of mobile software development needs to change from a craft, employed in a case by case manner, to a fully automated *product-line* specialized for mobile applications. A product-line offers the possibility to reuse in the future the common investment done in a series of individual products [Par76, CN02].

Mobile device applications share a lot of common non-functional features, e.g., screen management and data persistence[2], that can be factored out and made part of an automated product-line. The common functionality can then be requested declaratively and injected automatically in particular mobile device applications. Maximum automation can be achieved by focusing on a mobile application domain, and developing technology that takes into consideration the specific characteristics of the domain[3]. Mobile applications are easier to automate than their desktop or server counterparts, because they have limited variability, e.g., in the ways the user interface is composed, or in the possibilities available for data persistence. Despite the advances in hardware technology, there is a set of properties that remain specific for mobile software, e.g., the limited screen size, low memory usage, sporadic networking, and ease of use, that need to be specially addressed.

At the software application level, isolation from specific mobile device hardware and specific operating system software can be achieved by using virtual machine abstractions, e.g., J2ME Mobile Information Device Profile (MIDP) [Jav05] and .NET Compact Framework [Mic05]. The focus in this book is in the domain of mobile applications that run on mobile devices supported by *virtual machine* abstractions (Figure 1.2). As shown in Figure 1.2, several related mobile device applications can be managed by a product-line that addresses common repetitive technical issues. In a ubiquitous environment a mobile application also communicates with the environment as a client of one or more services, e.g., to access data in a central server-side

[2] Preservation of the data, this is, the ability to store and load back the stored data.

[3] A *domain* is a specialized body of knowledge, an area of expertise, or a collection of related functionality [CN02].

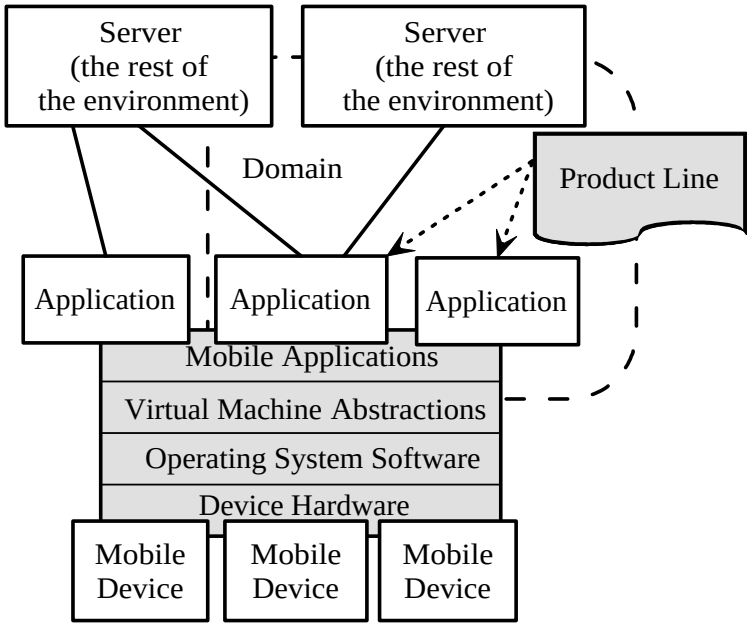


Figure 1.2: Mobile Device Applications Domain

location. Part of the server-side software found in the environment is closely related to the functionality of mobile device applications and may not be needed for other types of clients. This server-side part is represented inside the dashed box in Figure 1.2. It contains also repetitive technical concerns that need to be addressed by the product-line. A systematic and automatic approach for mobile application software development that addresses both client- and server-side issues is required.

1.2.1 Programming Models for Mobile Software

Mobile and especially embedded software has been always difficult to write and debug for several reasons. There is a gap between the machine where the software is developed, usually a desktop PC machine, and the actual device on which the software will run. This gap requires using various emulators for the actual devices in order to build and test the actual software. One more step, namely emulator testing and (remote) debugging, is added to the usual development cycle of desktop software. Various implicit coding conversions, based on the language or specific API[4] restrictions, must also be followed in order to write efficient software.

[4] Application Programming Interface.

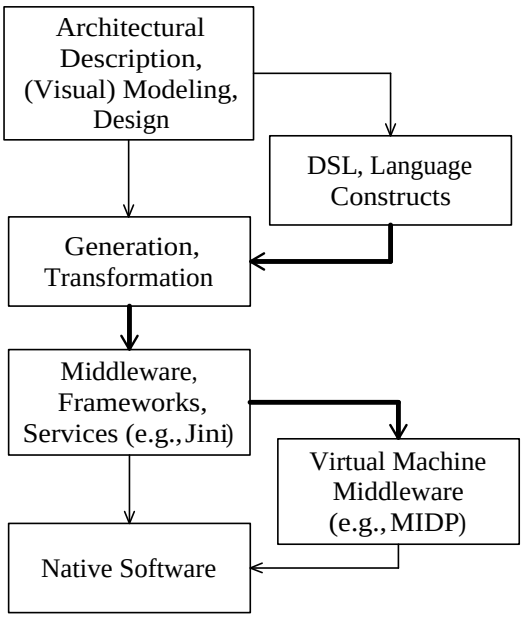


Figure 1.3: Programming Models for Mobile Software

Figure 1.3 shows a high-level view of the common techniques used to develop software for small devices, such as, visual modeling and domain-specific languages (DSL) [vDKV00]. Programming small devices can start at any node of the diagram in Figure 1.3. Not all systems use all the paths shown, and more than one path is possible. For example, the model of an application can be defined by using visual domain abstractions. This visual model is then either converted to a specific DSL, or is used to generate directly source code in a general-purpose language. The generated code targets directly the native software, or makes use of existing middleware frameworks. The application could run against the device hardware, or on the top of a virtual machine. Various forms of these programming models have been used in real-time embedded systems [Wir77, GMS94, FMG02], and other embedded software and frameworks [BPSV01, NFH+03].

Mobile device applications addressed in this book have less restrictive requirements than embedded systems, but more restrictions than other PC desktop software. Virtual machines, e.g., J2ME [Jav05], .NET CF [Mic05], and their programming models, such as MIDP [J2M02b], help to hide the hardware and software details for different classes of mobile devices. Virtual machines, however, do not offer high-level abstractions for application domains. As the focus on ubiquitous computing grows, so does the number of mobile applications that need to be developed and debugged. There are more and more applications for mobile devices that share

a lot of similarities, but also have their own peculiarities. Reusing only the virtual machine abstractions to hide OEM[5] specific device details is not sufficient any more. The commonality and variability of families of applications must be supported as well.

More than one programming abstraction of Figure 1.3, e.g., visual modeling, DSL and code generation, can be used to factor out and reuse the common functionality, and some of them are more declarative than the others. The approach presented in this book goes along the bold connection lines in the Figure 1.3. The goal is not to develop a new kind of middleware system, but rather to propose ways that automate the creation of third-party mobile software by relying on existing middleware, such as MIDP [J2M02b]. MIDP application development can be supported with abstractions that organize and reuse MIDP specific domain functionality. The interest will be in declarative representation of the domain abstractions at the source code level. Declarative constructs preserve the domain abstractions in the source code, and reduce the development and debugging time by hiding the details of a more complex underlying programming model.

1.2.2 Variation Mechanisms for Mobile Application Product-Lines

There are different ways to parameterize the common behavior characterizing a domain and reuse it in a specific application, known also as *variation mechanisms* [Bos00]. The automation of mobile device software product-lines requires generic software technology to support *iterative* (§2.1.1) product-line development. This means that the variation mechanisms should be:

a. Easy to introduce and maintain (low-cost). The iterative development of a product-line is important to support product-line evolution [Pus02].

b. Flexible to support design experimentation. The correct *domain abstractions* used to represent the *domain concerns* are often not completely known from the beginning and may change as the product-line evolves.

c. Declaratively supported to reduce accidental complexity [FPB87] and achieve transparent automation. The domain dependent concerns should be injected automatically in a specific mobile application. Domain dependent concerns are code-related *core assets* (reusable artifacts) for a domain [CN02].

d. Enable a clear separation of the common domain functionality and the application specific functionality and enable domain-specific optimizations. The domain concerns are usually *cross-cutting* [KLM+97], that is, they are needed in more than one place (components) within an application.

Several technologies [Bos00] can be used to support variation, e.g., inheritance, component libraries (object-frameworks [BCS00]), extensions (selection of variants), configuration,

[5] Original Equipment Manufacturer.

parameterization, templates, macros, generation [CE00], (embedded) domain-specific languages (DSL) [vDKV00], compiler directives, visual modeling and CASE[6] tools. All these variation mechanisms have their own benefits and drawbacks and need to be specifically evaluated for mobile software applications. The library solution is the simplest one, as it requires no further tool support other than those offered by the development language. It offers minimal automation for inserting the domain concerns into an application. Code generation is easy to implement, but pure generative solutions[7] are difficult to maintain and debug because of the programming indirection and lack of early static checking. DSL are more declarative and enable transparent automation, but have high start-up costs to be implemented. Visual modeling abstracts the domain concerns, but it does not support well iterative development of abstractions and traceability. Often it is not feasible to define every piece of behavior visually and source code artifacts are used instead[8]. A more elaborated discussion of the variation mechanisms of interest for mobile product-lines is provided in section §2.1.

1.3 Attribute Enabled Software Development

This book introduces variation mechanism that fulfills the requirements posed for mobile product-lines and does not have the problems encountered with other variability mechanisms. Mobile software product-lines can be implemented with programmable software container abstractions, backed up at the source code by lightweight domain-specific abstractions, encoded as explicit code attributes, which are interpreted by means of code transformation techniques. This section explains shortly what is meant by these terms.

1.3.1 Mobile Containers

A *software container* is an architectural abstraction that can be used to organize a product-line for a domain, by factoring out the common domain behavior into a set of *services* provided by the container. The container services are reused by all applications in the product-line. An application component inherits quasi-transparently the domain-wide features from the container it is deployed in. That is, when developing an application, the developer can focus exclusively on the features that are specific for the particular application at hand. The domain concerns are provided (*injected*) when needed, into the components of the application by the container implementation.

[6] Computer Aided Software Engineering.

[7] That is, generative techniques used alone, not in a combination with other variation mechanisms.

[8] For example, a UML-based CASE tool may generate only code stubs, whereas the method internals need to be filled out manually. The functionality of the methods could also be specified visually, but unless the functionality is made of well-known repetitive code, it is often easier to program using source code directly.

The container abstraction is known from enterprise server-side containers, e.g., COM+ [Gor00] and EJB [J2E03]. The idea can be applied to organize services in every domain of interest, especially to achieve automation of a mobile product-line. The container creates a well-defined boundary between the domain abstractions and the rest of an application. For a mobile application, the container provides a single centralized point of maintenance, that wraps the underlying services offered by the middleware (e.g., J2ME [Jav05]), making it easier to support mobile software evolution.

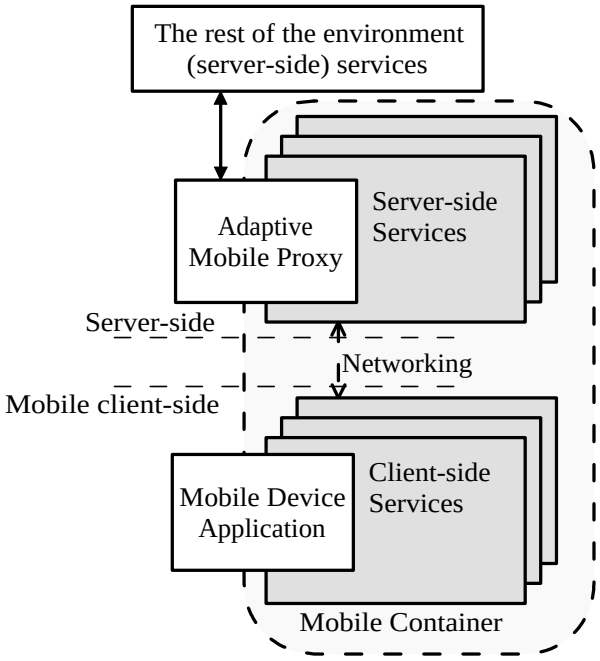


Figure 1.4: Mobile Container Architecture

The term *mobile containers* will be used in this book to refer to a special combination of client and server containers which is appropriate to accommodate the specific characteristics of the domain of mobile applications. A mobile application is a client application with *sporadic* network connectivity. As such, it is neither a thin nor a thick client[9]. It cannot be a thin client, because it should be able to do some processing and data storage on its own when disconnected. It cannot be a thick client either, because of the restricted resources of a mobile device. In order to save computing resources of the mobile device, as much of data processing as possible should happen on the server-side.

[9] A thin client does only interface processing on the client-side and delegates any other data processing to the server-side. A thick client does most of the data processing on the client-side.

A distinction will be done between the server-side functionality that is specific for mobile device applications, and the functionality that is common for other types of clients[10], as illustrated in Figure 1.4. The server-side functionality that is common to mobile clients can be factored out from the rest of the server-side services in the form of an *adaptive proxy* [FGBA96]. The mobile container is devised as a special kind of a software container [VSW02] that automates the organization and injection of domain concerns (a) in the mobile device client applications, shown in the lower part of Figure 1.4 and (b) in the intermediate application-level[11] proxy part between the server (environment) services and the mobile clients. The adaptive proxy stands also on the server-side (Figure 1.4). The server-part of a mobile container automates those aspects of the proxy that are directly connected with the functionality found on the mobile clients. It is the responsibility of the adaptive proxy to serve as a central connection point for representing the mobile clients and to communicate with the rest of the server-side services. The product-line handles the concerns of the both parts of the mobile container.

To support the domain variability, the container itself should be *programmable*, i.e., configurable, rather than providing a predefined set of services. The container can be used to inject any parameterized set of services into specific application components, using implicit or explicit architectural conventions. An open and extensible plug-in architecture with a configurable workflow is presented in this book (§5) to organize the product-line assets as container supported services. This enables software developers to specialize the product-line assets to fit their needs and to maintain them.

The inheritance of common properties from the environment is known as *environmental acquisition* [GL96, CF05] and could be supported by special language support. Some languages, e.g., Keris [Zen04], replace static linking with dynamic linking and can be used to support some form of environmental acquisition. The solution for modeling environmental acquisition presented in this book is *lightweight* in the sense that, it can be easily added to any existing language, making it suitable for iterative product-line development. The presented solution requires only a minimum extension to the existing language features, namely the ability to decorate the elements with attributes and support for interpreting the latter. The solution also makes the boundary between the application and the domain functionality explicit.

1.3.2 Lightweight Domain-Specific Abstractions

Domain-specific abstractions (DSA) are language abstractions, usually in the form of embedded domain-specific languages (EDSL) [Kam98], used to extend a general purpose programming language. DSA are used to add declarative support for specific concepts of a domain to a language, making it easer to write software for that specific domain. Having domain abstractions in code helps to preserve the domain architecture at the code level. The source code that con-

[10] E.g., powerful desktop clients.

[11] Not to be confused with lower level proxies used to represent objects remotely over a network.

tains the key abstractions of a domain declaratively is easier to understand. Declarative DSA blur the architectural gap [GSCK04] between a specific modeling step, and modeling directly at the source code level. DSA can be used to declaratively request the container-based services in the source code of a mobile application. As illustrated in Figure 1.5, declarative DSA can be interpreted and automatically connected to the container provided services.

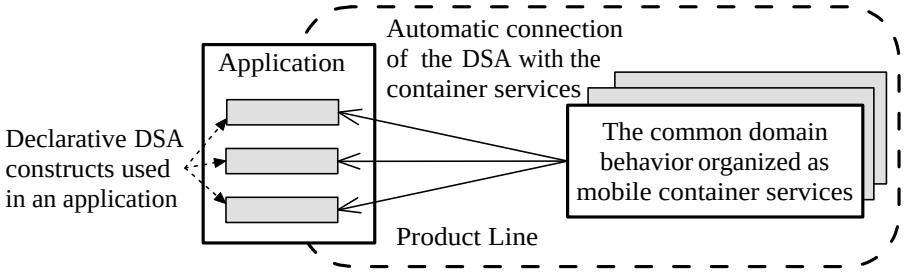


Figure 1.5: Connecting DSA with the Container Services

Explicit attributes [BCVM02], directly supported at language level, can be used as a low-cost mechanism to emulate DSA. Attributes[12] are a lightweight language extension that remove the need to implement explicit domain specific language abstractions [TS97]. Attributes can be used as a variability mechanism to model product-line domain abstractions directly in the source code of a specific mobile application (§3.1.1). Attribute-based abstractions are cheap to introduce and to modify in a general-purpose language. A language with support for attributes and attribute-driven transformations eliminates the need for other parsing and EDSL tools for implementing language abstractions to sustain mobile product-lines.

Several general purpose language technologies, e.g., .NET [Pro02] and Java [Jav04], already offer some support for attributes. The main support provided by these technologies is to enrich structural entities, e.g., classes and methods, with attributes and to preserve such decorations in the binary meta-data[13]. A more complete support for attributes and attribute transformations as part of the language technology is needed when attributes are used to emulate DSA. The abstract syntax tree (AST) created for the source code internally by the compiler, the AST-s created by various source processing tools (and API-s), and the AST obtained via reflection in languages that support meta-data, represent similar views of the same data structure, at various levels of detail[14]. The interfaces used to model these similar AST views are often different and the transformations done in one view are not easy portable to another view.

It is possible to unify all different AST representations, in a single data structure that pro-

[12] The terms *attribute*, *annotation*, and *tag* will be used interchangeably to denote the same concept, unless explicitly noted otherwise.

[13] Data describing the other data.

[14] For example, the details of the method body are not modeled in the reflection API.

vides support for different levels of detail. This generalized AST can be obtained either from source code or binary data, and can be used to implement attribute transformations for mobile product-lines. For attribute languages, this common representation will be called a *Generalized and Annotated AST (GAAST)*, and the languages that offer such an API as part of their technology will be called *GAAST languages*. The GAAST API, combined with ways to enable filtering nodes of interests, can be used as a common API to implement meta-programming attribute-driven transformations in a general-purpose language. If GAAST is part of the language, the investment on the transformation tools is protected as the language evolves.

Attribute enabled languages affect also the design [NV02] process. OMG Model-Driven Architecture (MDA) [Fra03] transformations, whose input models are represented as UML[15] class diagrams, can be directly supported in a GAAST language. Using attributes to sustain DSA at the language level preserves architectural decisions of the domain models. GAAST languages require only type mappings to support several MDA Meta-Object Facility transformations. In MDA, the software development could start with a platform independent model (PIM), and then be iterated toward a platform specific model (PSM), introducing platform specific details in each step. GAAST languages offer a common API to directly support such transformations and open new possibilities to design applications, that will be investigated in detail in chapter §3.

1.3.3 Attribute-Driven Transformations

Any invasive [Aßm03] system that can access the abstract syntax tree (AST) of a program can be used to implement static attribute-driven transformations for attribute-based DSA in a product-line. For example, GAAST languages offer all the necessary means to support attribute-based transformations. The implementation of transformers in the GAAST API level is still very general, and may result in non-reusable transformers which are difficult to maintain.

Programming at the GAAST-level requires following implicit coding rules to be able to reuse the transformation behavior. Many AST manipulations based on the GAAST-API are repetitive across different transformation operations. Transformer implementations that directly build upon GAAST lack a clear structure of the transformation process. This opens the need for specialized transformer technology to enforce reusable modular organization of attribute-based transformers. Having modular transformers facilitates understanding the attribute semantics, and helps using attributes successfully to model the domain variability.

Knowledge about the domain, modeled as a set of attribute-based DSA, and about the deployed programming language can be explored to introduce a *horizontal* and *vertical* modularization for attribute-driven transformers (Figure 1.6). The transformation process can be structured horizontally according to the domain assets modeled with attribute-based DSA. The knowledge about the specific domain assets can be used to guide the transformation workflow.

[15] Unified Modeling Language.

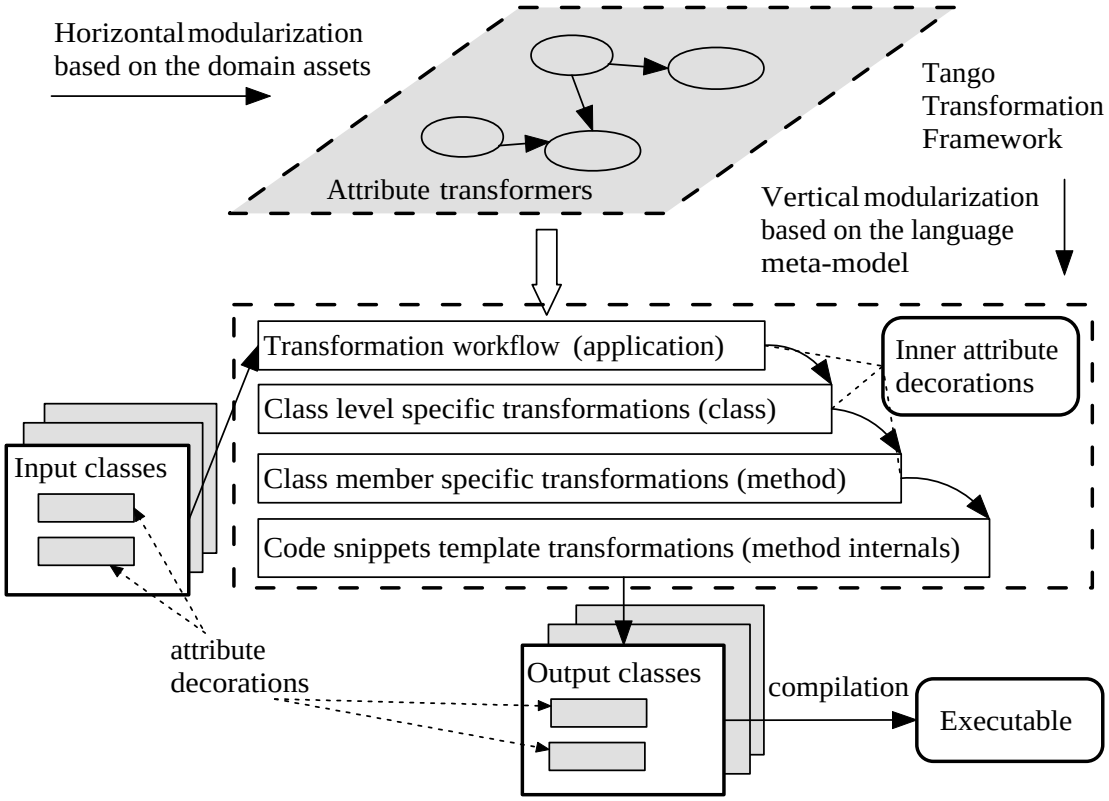


Figure 1.6: Layered Attribute-Driven Transformations

The workflow dependency graph can be partially computed automatically, based on local *before* and *after* dependency relations supplied by the developers that model the semantics of the interactions between the domain assets.

Knowledge about the structural nesting of the elements found in the language meta-model can be explored to organize the transformation strategy in vertical layers, as shown in the middle part of Figure 1.6. The approach for modular construction of attribute-based transformations taken in this book, embodied into the Tango transformation framework (§4), supports a hard-wired common OO language meta-model made of classes, fields, and methods, which in turn are made up of method code blocks. The hardwired meta-model enables a vertical layering that otherwise would not be possible with an arbitrary open meta-model. Some other frameworks [Vis01b, KHH+01] deal with this issue by defining parameterized transformation operations, e.g., selection and iteration, that can work with any AST element type, but have a single layer to organize the transformation process. The approach presented in this book defines also declar-

ative operations, but specializes them for each meta-model element type and organizes them in layers according to the language meta-model. The transformation process in Tango shown in Figure 1.6 contains operations specialized for each transformation layer, e,g., class level specific operations.

Attribute-based transformers do not only operate on AST nodes that are explicitly decorated with attributes, they can also decorate the AST as they process it with *inner attributes*, intended to support the transformation process (schematically shown in the middle right part of Figure 1.6). This technique is similar to saving the intermediate transformation results into declarative placeholders [vdBHKO02]. Unlike in a compiler, where the AST decoration with attributes is used only internally [WB00], attribute-driven transformers treat the explicit[16] and the inner attributes[17] uniformly. Given an attribute, a transformer cannot tell whether it comes from the source code, or from another previous transformer.

Inner attributes enable a declarative composition of transformers. A transformer declares (a) the set of attributes that it expects the AST elements to have, and (b) the set of attributes the elements will have after the transformation. Without relying on attributes for coupling transformers in a chain, each transformer needs to re-check all properties of the AST elements passed to it, before it processes them. When using attribute-based coupling, the transformers can rely on attribute decorations of previous transformers and do not need to revalidate all the semantics the input units are required to have.

Some attribute-driven transformer operations, e.g., checking attribute dependencies, are generic and can be factored out of any transformer. Such cross-cutting transformation concerns can be made declarative by using a way that is natural for GAAST languages. The decoration of an attribute with other attributes (often called *meta-attributes*) can be used to declaratively express generic operation semantics, e.g., the attribute dependencies. In the attribute dependency case, presented in chapter §4, attributes are decorated with appropriate forms of a dependency attribute when they are created. Decorated attributes can be used as normal attributes. The dependency information can be later checked and enforced automatically depending on the attribute usage context.

Chapter §4 explains how the mechanics of this proposal for organizing attribute-driven transformations can be made declarative and automatically enforced, by creating a language specialized for attribute-based transformers. Such a language makes the transformation operations more declarative and easier to write and maintain.

[16] Attributes used in source-code directly to express the DSA semantics.
[17] Attributes that express internal transformation semantics.

1.4 Contributions of this Book

Most of the contributions of this book are specific for mobile software and mobile software product-lines. Other contributions are of generic nature, and apply also to product-lines for other domains. This section only briefly mentions the introduced concepts to organize them in a single place. For more information see the referred chapters.

Architectural Contributions

- This book contributes a new architectural style for organizing the domain assets of a product-line as *container-managed services*, based on *attribute enabled software development*. An open container supported by attribute-based DSA can be easily extended with new services that model the domain concerns supporting iterative product-line evolution (§2, §5).
- A new kind of container abstraction for mobile product-lines, a *hybrid mobile container*, is another contribution. A mobile container is a unique combination of a client and a server container, optimized to automate product-lines for mobile device software. An enterprise server-based container extends from server to client [VSW02]. The reverse is true for a mobile container. The mobile container represents a client extension on the server-side, and takes over some operations normally handled by a client application and moves them to the server-side (§5).

Language Contributions

- This book contributes the notion of *Generalized and Annotated AST (GAAST) languages* used to unify common AST related API-s, supported by the language technology, to enable uniform source and binary transformations. GAAST creates a low-cost language workbench [Fow05] to support attribute-based DSA. As various transformation techniques are being continuously explored, having a unified mechanism for transformations as part of a language reduces the cost of building and maintaining several third-party tools (§3).
- The book helps to bridge the gap [GSCK04] between modeling and coding, by relying on *architectural support via low-cost DSA*. DSA preserve the architecture of product-line domain assets directly in source code. Low-cost attribute-based DSA make it possible to directly utilize the product-line domain abstractions at the language level (§3).
- This book also contributes to the domain of specialized languages for *attribute transformations*. Attribute transformation languages serve as domain-specific languages for attribute transformations that enforce modularity and enhance reuse (§4). These languages make use of the characteristics of the addressed domain to enable better organization of the transformation process, increase productivity, and automate traceability.

Transformation Contributions

- *Specialized attribute transformation engines* are introduced as a technique to support attribute-based DSA transformations in the context of a mobile container. Specialized attribute transformation engines are organized as a series of transformation layers. Inner attribute decorations make attribute transformer coupling declarative and enable transformer composition (§4).
- Another contribution is the enhancement of the *attribute transformation support with meta-attributes*. Attribute transformation concerns that cross-cut more than one transformer can be addressed declarative by expressing them as meta-attributes. Meta-attributes are used to extend an attribute enabled language, to support generic transformation operations, e.g., attribute dependency. Such generic operations are made native to a GAAST language by organizing them as meta-attribute decorations over the attribute definitions (§4).

In addition, several **practical contributions** have resulted from the concrete instantiations of the conceptual contributions:

- The *MobCon* framework [CM05b] is a mobile container for J2ME MIDP [J2M02b]. Several MIDP concerns are addressed and organized as a generative mobile container framework based on the technology developed in this work (§5).
- The *MobCon Transformation Engine (MTE)* is a generative GAAST-based transformation engine for Java. The generative attribute transformation engine of MobCon is extensible, and can be used to implement Java-based attribute containers for other domains apart of the J2ME MIDP (§5).
- The *Tango* framework [Cep04] is an attribute-based transformation language. Tango is a prototype implementation of layered attribute transformers with inner attribute notations. Tango enforces layering of the transformation process and offers transformation predicates specialized for each meta-level (§4).
- The *Attribute Dependency Checker (ADC)* [CM04] is a constrain checking engine for .NET attribute dependencies [Pro02]. ADC makes use of meta-attributes to declare attribute dependencies for any depth level of the structural tree, and to enforce them automatically (§4).

1.5 The Structure of the Book

Chapter §2 provides background information about current technology for software product-lines. Product-lines and various variability mechanisms are discussed. The container abstraction

is presented and an overview of invasive and non-invasive container implementation techniques is given. In the end, aspect-oriented techniques are compared with more specialized transformations.

Chapter §3 focuses on attribute enabled software development (AESD). The architectural properties of attribute enabled languages are presented by a Model-Driven Architecture (MDA) transformation case study, where the same model is mapped using representative ways onto language constructs. An overview of attribute support in .NET is given, followed by a generalization of these ideas in the form of a GAAST language. Next, a comparison of GAAST languages and other meta-programming approaches is made. The chapter ends with some heuristics about the right usage of attributes.

Attribute-driven transformations, and specialized languages to support them, are the theme of chapter §4. Different ways to make the attribute-based transformations modular are explored. The focus will be on the Tango language, specialized for attribute-driven transformers. Tango enforces a layered structure upon the transformer implementations. A declarative way to represent and enforce attribute dependencies based on meta-attributes is discussed in the end.

Chapter §5 builds upon the concepts of the previous chapters, and shows how a mobile container for Java 2 Micro Edition - Mobile Information Device Profile (J2ME MIDP) can be constructed. The GAAST ideas are used to develop a generative container framework for Java, and specialize it with services that are specific for MIDP applications, such as, data persistence, image adaptation, and user interface management. A medical application for X-Ray diagnostics is used to demonstrate the usefulness of the developed MIDP container framework.

Chapter §6 provides a summary of the main concepts and presents some ideas for future work. The related work is distributed uniformly in every chapter, to keep it near the point where the relevant technical details are discussed.

Most of the material presented in this book shares content with several conference and journal publications of the author which are referred to in the corresponding chapters.

Chapter 2

Organizing Mobile Product-Lines with Mobile Containers

The harder is to see the design in code, the harder is to preserve it, and the more rapidly it decays.

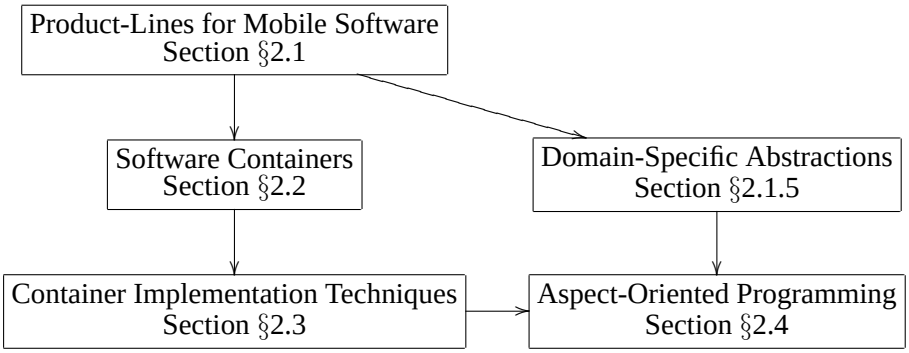
M. Fowler, Refactoring - Improving the Design of Existing Code, Addison-Wesley, 1999

This chapter serves two purposes:

- It introduces background information needed to explaining several technologies discussed in this book. Details are given about product-lines, variability mechanisms, software containers and their implementation techniques.
- It provides an extended overview of the software issues and problems addressed in this book. The focus is on automated product-lines with language support for domain abstractions. The material presented in this chapter serves as a starting point for the upcoming chapters.

While most of the information discussed in this chapter is also relevant for other domains, the focus will be on the importance of such technologies for supporting product-lines for mobile

device software. This chapter is organized as shown in the Figure below.



Section §2.1 justifies the need for bringing in more automation in the development of mobile software by employing product-lines. The preferred characteristics of a product-line for mobile applications are given in section §2.1.2. The section compares several variability mechanism focusing on object-oriented frameworks, visual modeling with Computer Aided Software Engineering (CASE) tools and domain-specific languages (§2.1.5).

Software containers represent an architectural pattern of interest that can be used to organize mobile product-lines. Containers transparently introduce functionality into a set of serviced components. The container abstraction is known from technologies, such as, EJB and COM+. An overview of these two technologies and how they use containers, is given in section §2.2. The benefits of generalizing the container concept as a means to automate arbitrary domains are then explained.

Section §2.3 addresses several technical issues related to implementation of the container abstraction. Background information about various possible implementation technologies is given, making a distinction between invasive and non-invasive transformations. These technologies are used to sustain programmable dependency injections, separating the serviced components from the container services. Invasive and non-invasive techniques are evaluated for mobile containers.

Aspect-oriented programming (AOP) techniques are compared with other invasive techniques for implementing domain-specific abstractions (DSA) in section §2.4. Section §2.5 ends this chapter with a summary and an overview of the proposed technology.

2.1 Reusability with Product-Lines

Factoring out the common functionality of mobile software and reusing it in specific mobile applications can be done with automated *product-lines* [Par76] specialized for mobile applications. A software product-line is defined in [CN02] as *"a set of software-intensive systems shar-*

ing a common, managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way". Other terms used often interchangeably to describe product-lines are *product (application) families* [Par76, JRvdL00] and *software factories* [GSCK04]. This section gives an overview of product-lines and discusses several mechanisms of interest for implementing product-lines that automate mobile software.

2.1.1 Two Views of Product-Line Development

A product-line reflects the experience gained by creating many applications that share some common set of characteristics, usually because they belong to the same domain. This common set of characteristics can be parameterized and can be factored out to represent the domain variability. This is also known as *variation management* [Har01]. The common functionality is reused [SB00] in every new application within the same domain. *Application families* [JRvdL00, Par76] are made up of several applications that share the reusable functionality provided by the product-line.

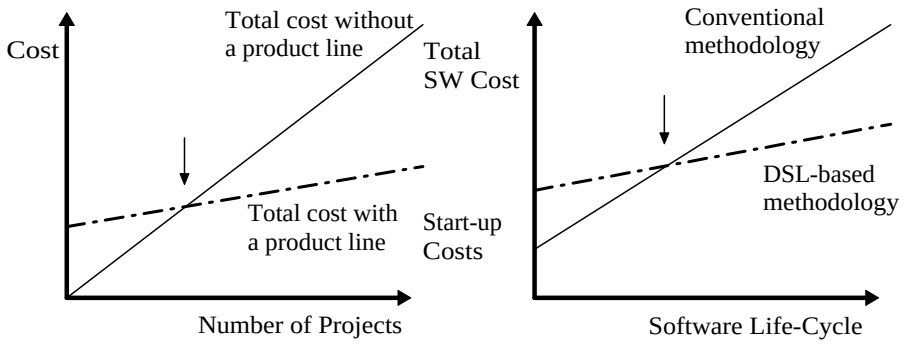


Figure 2.1: Product-Line Payoff

A product-line can be seen as up-front investment that pays itself off in the long term. This is showed often graphically as in case (a) of Figure 2.1, taken from [CN02], where the start-up costs of the product-line approach are bigger than those of an approach without a product-line. The product-line investment pays off after a given point. These empirical diagrams look similar for other technologies that require some up-front investment as in case (b) of Figure 2.1 that shows the payoff for domain-specific languages (DSL), taken from [Hud98].

An alternative view of a product-line is to consider it as an accumulation of investment, rather than an up-front investment. The more similar applications are built for a given do-

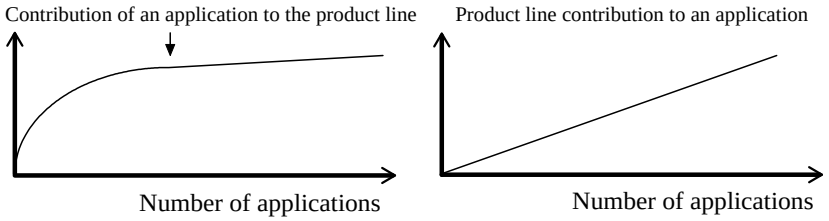


Figure 2.2: A Product-Line and its Applications

main, the more the previously implemented functionality will be reused in the new applications. Turning such an accumulation of investment into a really profitable product-line is a matter of discipline and architectural refactoring [Fow99, GSCK04]. There is a mutual contribution between the product-line and a specific application illustrated by the diagrams of Figure 2.2: (a) each specific application contributes (back) to the product-line, enhancing its variability; (b) the factored common product-line functionality is reused with proper variability parameters in a specific application implementation. A product-line is in an iterative development process, while new applications are created for the same family. In the long term, it is expected that after a *threshold* point, the product-line development changes very slowly when new applications are created. The threshold point is envisioned to be very close to the payoff point of the up-front investment view of product-lines.

The iterative view of a product-line is more suitable than the up-front investment view for the domain of mobile device software. Each ubiquitous computing scenario has its own peculiarities and contributes back in the common code base that is shared between all mobile applications, which are based on the same set of technologies for a given domain. The product-line assets are gradually structured as more scenarios are implemented.

2.1.2 Variation Mechanisms for Mobile Product-Lines

The common functionality of mobile applications can be reused when it is parameterized with regard to its variability. There are different mechanisms for organizing and reusing the common product-line functionality, known as *variation mechanisms* [Bos00, Har01], e.g., component libraries, inheritance and code generation. From the point of view of an iterative approach to mobile software product-line development and evolution, three general requirements can be identified. Some of these requirement are useful for any kind of software product-line, however, they are especially important for mobile applications.

- The variation mechanisms for mobile product-lines should *facilitate the introduction and maintenance of domain abstractions.* The creation of a product-line is an iterative process.

It should be easy to start creating a product-line, adding functionality to it at any time and maintaining it. No product-line is acceptable unless it can deliver in time [GSCK04]. If the costs of introducing reusable assets in a mobile product-line are high, the product-line will not be embraced in the early phases of the development. This will make it even more difficult to introduce a product-line later on, after some applications have been developed. The variation mechanisms should enable the evolution of the domain assets, as the underlying mobile technology (middleware) changes often, to respond to the market demands and technology innovations.

- The variation mechanisms for mobile product-lines should *enable stating declaratively and introducing automatically the domain functionality.* The integration of the product-line's generic functionality into a specific mobile application should be as much automated as possible. Automation accelerates the time to market, by reducing the debugging time (which is higher for mobile applications because of the indirection gap between the machine where the development is made and the device where the software runs).

  The used variation mechanisms should enable a declarative representation of domain assets, in order to preserve the design model within an application, and to avoid accidental complexity [FPB87]. Declarative constructs are important to ease the maintenance of a product-line during its evolution [Pus02], and to clearly define the boundary between the generic domain functionality and the application specific functionality. As noted in the discussion of case (a) of Figure 2.1, the product-line start-up costs are usually higher than the costs of applications without a product-line. Adding the costs of developing a declarative notation (case b), to the product-line start-up costs related to domain engineering, can result in an unacceptable overall cost for a project. As argued in [Hud98], it is not sure that the payoff point really comes in every case. Hence, declarative variation mechanisms with very low start-up costs are needed.

- The variation mechanisms for mobile product-lines should *enable domain-specific optimizations.* Mechanisms that balance between the increased level of abstraction and the introduced run-time performance are needed. For example, the indirection layers could be minimized. For example, in the J2ME MIDP [Jav05], the class meta-data are saved in the application's binary file. The number of classes affects the size of the application and should be kept in a minimum.

To summarize, the variation mechanisms (VMs) for product-lines for mobile applications should:

a. make the domain functionality explicit by means of declarative domain specific abstractions,

b. provide for stability of domain abstractions in the prospect of fast underlying technology changes,

c. support automated integration of domain and application functionality,

d. have low start-up costs and require minimal additional (language) technology,

e. enable domain-specific optimizations.

Chapter §1 listed several variation mechanisms for product-lines discussed in [SB00, Bos00, Har01], e.g., inheritance, code generation, and compiler directives. An overview of the other techniques can be found in [GSCK04]. The list of the variation mechanisms given in [SB00, Bos00, Har01] is extended in this section to include visual modeling and domain-specific languages (DSL), as more declarative mechanisms. The remainder of this section examines OO frameworks, visual modeling and DSL.

2.1.3 Object-Oriented Libraries and Frameworks

Object-oriented (OO) libraries and frameworks are interesting variation mechanisms because they enable implementing the domain variability with techniques already found in OO languages. OO libraries are the native way to factor out and reuse functionality in an OO language. The domain variability can be expressed, by using method parameters, overloaded methods, or forms of OO inheritance, e.g., template polymorphism in C++ [Str97].

While OO libraries and frameworks use the same underlying OO language variability mechanisms, the difference between OO libraries and frameworks becomes clear when implementation flexibility is considered. OO libraries allow a lot of freedom in design and usage. The coding and the architectural conventions in an OO library are implicit. It is easy to extend an OO library with new domain assets that do not follow the conventions. The implicit design rules are often not enforced. This results in non-uniform libraries, where the reusable components may follow different design patterns. Non-uniformity of the design makes it difficult to reuse OO library components in a product-line. It makes also difficult to refactor the product-line as it evolves. Patterns [GHJV95] help to clarify common designs, but they are not a replacement for architecture [SSRB00]. It is not uncommon to find more than one different implementation of the same pattern within the same system, resulting from the incoherent design.

In order to access the required domain functionality assembled in an OO library, each application has to go each time through several steps, e.g., component library initialization, object(s) instantiation, selection of required interfaces, and passing of required parameters. This adds accidental complexity to the application code. Component library calls with various parameters are also difficult to maintain when the library functionality and interface change as the product-line evolves. When domain abstractions are implemented as libraries, e.g., as a library that contains the functionality common to *web services* [ACKM04], additional classes and interfaces need to be introduced. The more generality is needed, the more indirection the OO implementation will add [ACKM04].

Frameworks perform better than OO libraries, when used to organize product-lines. Frameworks place design restrictions in the way the applications are built, and define clear specializa-

tion points for the developers to follow [GSCK04]. The design restrictions of a framework show in the form of well-defined (a) *specialization* points, e.g., how to create an enterprise bean in EJB [EJB03] by deriving it from a specific priorly-defined based class (§2.2.2), and (b) well-defined *extension* points, e.g., the (missing) possibility to create a new type of bean in EJB. The clear specialization and extension points enforce a uniform structure in a product-line and make it easier for developers of the product-line to maintain it. Developers benefit from the clear specialization points that enable reusing the architecture of the framework in a new application.

Although OO frameworks have been used successfully to organize product-lines [BCS00], they cannot define proper abstractions to represent the domain concerns, which makes it more difficult to reason about the domain by looking at the code of an application. Using OO frameworks may result in more coding, because the domain variability should be expressed in pure OO language mechanisms. For example, in EJB 2.0 [EJB03] the variability mechanisms are based on OO constructs, such as, interfaces and required method names. The complexity of the programming model in EJB 2.0 motivated the need for EJB 3.0 [EJB04], which uses a more declarative way to express the domain variability. OO frameworks are not declarative and offer few automation. OO frameworks are often combined with other variability mechanisms, e.g., visual modeling (§2.1.4) and code generation [CE00], that contribute respectively to a more declarative domain representation and more automation.

2.1.4 Visual Domain-Specific Modeling

The most successful approaches in automating embedded software have been those based on visual meta-modeling and code generation from graphical models. Any Computer Aided Software Engineering (CASE) tool can be used to model an application in a domain of interest, and generate code stubs from that model. Visual modeling has been successful for mobile and embedded applications even before the popularity of virtual machine abstractions. The reason is that code generation can be optimized for a given device, and only recently generic virtual machines that run efficiently on small devices could be afforded.

The way CASE tools are used to model software has been gradually standardized under the Object Management Group (OMG)[1], as Model-Driven Architecture (MDA) [Fra03]. The most important standard behind MDA is the Meta-Object Facility (MOF) [Met02]. MOF defines the structure of a domain in a layered way. The lower layer represents the real information about the objects of a domain, also known as the M0 layer. The next layer, M1, models how this real domain information is represented in a computer. M1 is the lowest practical level for working with a domain. The M2 layer models the data structures used in the M1 level. This is also known as the *meta-layer*, and M2 models are known as *meta-models*.

While M2 is enough to describe any M1 model, often M2 itself needs to be described.

[1] `http://www.omg.org`

The motivation behind this is to enable different M2 representations to be interchanged between different CASE tools. For this reason, at the top of the MOF stands a *meta-meta-model*, known as the M3 layer. The M3 layer is very general and could be used to describe any meta-model, no matter the domain or the methodology the meta-model uses. Because of this generality, the M3 model is hardwired in the MOF standard and no further layers are needed. The M3 model is known as the *MOF model*. Everything in the MOF model derives from a single element, known as the *ModelElement*.

The MOF standard defines the MOF model and various mappings (representations) of the MOF model into various technologies, e.g., UML, CORBA, Java, and XML. The MOF model can be used not only to model data items and relations between them (the *modeling viewpoint*), but also to traverse the model graphs and obtain information about the model (the *data viewpoint*). Both these views are tightly interconnected and are part of the MOF standard. Other standards are currently under standardization by OMG that enable the transformation of the MOF models. While MOF is a generalization of UML[2] modeling [OMG03], UML 2.0 will be structured as a special case of MOF. MOF is intended to be a common ground for exchanging models between different visual CASE tools. The ideas behind MOF are quite generic, and MOF can also be applied to textual representations of models, including source code.

MOF is not directly interesting for domain modeling. Everything can be represented as a MOF model. Such a level of abstraction can be useful to prove theoretical properties of graphs [Men99], but only when the models are specialized for a given domain, they become useful in practice. The specific abstractions of a domain are important, because they embody the domain knowledge. They are used to reason about a specific domain better than the generic graph algorithms. The more specific a model becomes, the more useful it is for a given domain. For example, UML profiles [OMG03] enable UML specializations for various domains.

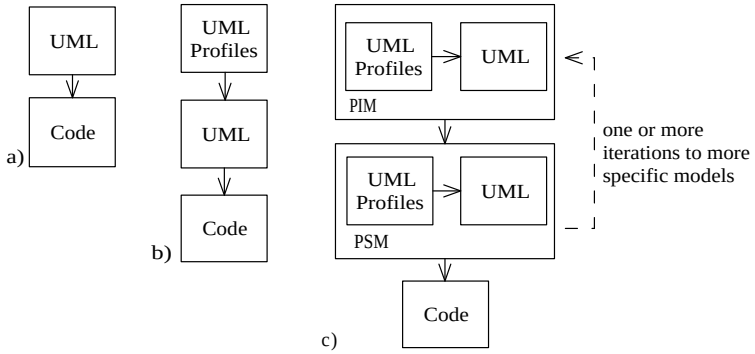


Figure 2.3: MDA Development

[2]Unified Modeling Language.

Figure 2.3 shows three different ways, how a visual CASE tool can be used to model an application. Although only UML [OMG03] is denoted in the boxes of this diagram, any (meta) model language can be used. The simplest case (a) shows generating code from a visual model which maps directly to the implementation language. Case (a) has been used in the past to generate embedded software. The visual model contains a predefined set of user-defined element classes. Case (a) is usually not very interesting because it supports only an one-to-one mapping into code and hence no automatic injection of the domain concerns is possible.

Case (b) of Figure 2.3 uses custom additions to the generic meta-model, in this case through one or more UML profiles, which helps to factor out and to reuse the domain functionality. For example, the UML profile for Enterprise Java Beans (EJB) [UML03, Met04] *"defines a set of UML extensions that capture the structure and semantics of EJB-based artifacts"* [UML03], mainly by utilizing stereotypes and typed values. The EJB profile captures the semantics of a specific domain, in this case of EJB [EJB03], in the form of predefined profile entities that can be reused to model applications in this domain. Rather than having to define applications in terms of object-oriented (OO) classes, a designer relying on the EJB profile can directly use EJB concepts, such as, ≪EJBEntityBean≫ or ≪EJBPrimaryKey≫. Case (b) represents how most existing CASE tools work.

The third case (c) generalizes case (b) by splitting the transformation of a model to code into several configurable transformation steps to intermediate models. Case (c) stands for the generic MDA scenario, where the created model (known also as *Platform Independent Model* - PIM) is refined into a more specific model (*Platform Specific Model* - PSM), before some executable equivalent is generated. This could require more than one iteration, moving to a more specific model in each iteration, whereby each model uses its own specialized profile. The case (c) implies also that the details of the PSM are automatically inserted into the PIM using appropriate transformations. While case (b) can be used for a product-line, it is the automated model of case (c) that could be of most interest for automated product-lines that require more than one mapping step, in order to postpone some of the design decisions to a latter phase. That is, case (c) supports *progressive refinement* [GSCK04] of the application architecture, from a very abstract design model to more specific models, known also as a stratified design [AK03, AK01].

There is no agreement in the MDA community on what is the best way to define PIM to PSM transformations. Often, only the structure of a model is defined graphically in a CASE tool, that is, components, their interfaces and relations. Although it is possible to define method implementations graphically (e.g., by using UML Actions), in practice it is often easier to associate pieces of code directly with the structural model elements, which means that pure visual models are rarely used.

An example of a commercial meta-modeling tool, used to model mobile and embedded software, is MetaEdit+ [Met03b] from MetaCase. MetaEdit+ has been used in several Nokia [Nok04] projects, e.g., graphical user interface modeling for TETRA (TErrestrial Trucked RA-

dio) devices [Met03a]. The main rationale behind MetaEdit+ is that domain-specific visual languages and visual domain-specific abstractions help to automate software development [PT02]. MetaEdit+ enables the creation of meta-models for specific domains, and using the elements of these meta-models to model specific domain problems. The domain elements and their relations are presented graphically. Source code artifacts can be associated with parts of interests, and various predefined or customized reports (including source code) can be generated from the models. Model transformations are not supported directly in MetaEdit+, and it basically works based on the model of case (b) of Figure 2.3.

Another example of a CASE tool is the Generic Modeling Environment (GME) [LMB+01]. GME, like MetaEdit+, uses a proprietary meta-meta-model and offers limited support for MOF, as just another meta-model. GME is a university research tool, developed based on the concept of Model Integrated Computing (MIC) and MultiGraph Architecture (MGA) [SK97]. While the terminology used in the GME documentation is sometimes peculiar, most of the MGA concepts can be directly mapped to the MOF terminology and MGA could be considered to be a custom case of MOF.

Compared to MetaEdit+, GME is limited only on defining meta-models, model constraints[3], and graphical models based on these meta-models. Code generation or other reports should be implemented as Visual C++ add-ins, or generated by doing a traversal of the model graph, by using various script languages that can access the GME COM components API. The transformation of the models is supported via graph transformations [Agr04] using other tools outside GME. Graph transformations support the case (c) of MDA development of the Figure 2.3. Graph transformations are based on writing transformation (graph rewrite [Sch94b, BS99]) rules as left-hand and right-hand side subgraphs. The left-hand subgraph, when found in a graph, is replaced with the right-hand subgraph. The specification of rules can be done graphically in a CASE tool, e.g., GME, or by using more formal grammar rules [RS97].

Matching of the left-hand side rules in a graph can be simplified by sequencing the order in which the rules are applied [Agr04], or by restricting the form of the left-hand side of a rule [RS97]. Rule matching can also be simplified by starting on predefined nodes, known as part of the sequence rules for a given domain. Milan [BPL01] is a framework based on GME, for the simulation of embedded system designs, in particular System-on-Chip (SoC) designs. Apart from GME modeling components, Milan also supports various simulations of the designed systems, using other third-party software. Another GME based system is GRATIS [VL02]. GRATIS supports high-level visual modeling for TinyOS [HSW+00], a small operating system for embedded devices. GRATIS helps to manage the event-based component communication in the TinyOS, reducing the development errors that result from directly maintaining the component interaction files.

Some other approaches, such as, SmallComponents [Voe03b], use predefined UML pro-

[3]The UML Object Constrains Language (OCL) is used.

files to generate the entire OS required in an embedded device, by including only those parts that are interesting for an application. SmallComponents-like approaches can be useful for small embedded devices where the entire system is generated at once containing a set of selected pre-defined applications. A comprehensive list of existing CASE tools can be found in [Jec04], whereas a list of preferable features for such tools can be found in [Cho04].

Based on the criteria for mobile product-lines listed in section §2.1.2, there are several reasons why visual modeling in isolation is not very useful as a variability mechanism.

- *Partial automation.* Ideally, a visual model can be used to generate code in more than one target language. The final target code is, thus, not very important to a CASE tool. In practice, this requires that the entire functionality is defined visually, which is not preferable in complex application domains, where a lot of specific application functionality must be coded manually. For this reason, CASE tools can easily generate stubs of code for a language, but leave most of the method implementation details to be filled out manually. Various techniques [Voe03a], e.g., the template method pattern [GHJV95], can be used to keep the manually added code separately from the generated code.

- *Difficult evolution.* Creating product-lines is an iterative process. The domain abstractions are not known from the beginning and they normally evolve as the product-line matures. The final generated code for such abstractions may not be optimal, or completely known from the beginning. Maintaining evolving abstractions visually in a CASE tool could result in additional costs [War94], because the meta-model and all the models based on it must also be maintained.

  CASE tools are better suited when the domain abstractions are nearly frozen and well-known. Language abstractions can better support iterative development of product-lines if their implementation and maintenance costs are low. Once the product-line stabilizes, it is usually easy to map domain abstractions, implemented by the programming language(s), to visual modeling elements.

- *Loss of design information.* Another issue using CASE tools alone, without appropriate support at the language level, is that source code is treated as an end product, similar to the binary code generated by a compiler. Despite comments added by a CASE tool in the generated code it is difficult, or impossible, to preserve the visual model architecture in an easy understandable way. It could be argued that the source code is not important, as long as the model is present. The model hides the details of the code that are not important for a domain. However, given that code must be often edited or completed manually, it is difficult to follow changes and debug it. Domain-specific abstractions, when supported directly at the programming language level, help to bridge this architecture gap.

2.1.5 Domain-Specific Modeling with Language Abstractions

This section discusses ways to support domain-specific modeling with language abstractions, rather than with visual abstractions and CASE tools. *Domain-specific abstractions* (DSA) refer to abstractions that are added to, or directly supported by a language, or that are an inherent part of the language, in order to support domain-specific modeling. For example, in the domain of web services special keywords, such as, `webservice` and `webmethod`, could be used to denote special treatment of web service constructs by the compiler, e.g., to automatically generate WSDL[4] files. Such keywords can be used to define a `TravelAgent` class, as a web service component, as illustrated in Figure 2.4.

```
1 webservice TravelAgent {
2   ...
3   webmethod GetHotels(){...}
4   ...
5 }
```

Figure 2.4: A Domain-specific Extension to Implement Web Services

There are several benefits from using declarative DSA as a variation mechanism for mobile product-lines:

- *DSA preserve the architecture of the domain in source code.* Consider the example of the web service abstraction (Figure 2.4) as a part of a product-line. When web services are supported declaratively in the language, a single keyword `webservice` can be used in every place the web service functionality needs to be reused. The source code with the declarative domain abstractions is easier to understand. The code is not considered as the end-product of some generator, which is difficult to modify by hand, but as a means to express the domain design. DSA help to trace the architecture down to code, as well as, to reconstruct the architecture from source code [GSCK04]. DSA support automation and reduce accidental complexity[5] in a product-line. While DSA have the same explicit programming model as OO libraries (§2.1.3), they automate the instantiation and usage of the domain abstractions in source code.

- *DSA can be used as an alternative to visual modeling.* It is easy to support a set of DSA, that are in place for a domain, with CASE tools. For example, an icon in a CASE tool can represent a `webservice` which will be mapped directly onto the corresponding DSA construct in the source code. Declarative DSA blur the distinction between an explicit

[4] Web Service Description Language. (http://www.w3c.org)

[5] Complexity related to the particular solution, not inherent in the solved problem itself. The later is known as *essential complexity* [FPB87].

modeling step, and directly modeling at the source code level. Working directly with the code, to refactor it using DSA, reduces one step in going through a visual modeling tool, but at the same time supports the same level of abstraction.

- *DSA implementations can be optimized using generative techniques.* The DSA processing tools have access to the abstract syntax tree (AST), and can directly inject the domain code into the AST locations where the DSA are found. To the end user, the abstractions are still declarative. It is hard to achieve this with object frameworks [BJMvH00, BCS00], where OO inheritance and composition are exclusively used.

There are several ways to support DSA:

i. New languages can be defined from scratch, whose elements directly correspond to abstractions in a particular domain. Such languages are called *Domain-Specific Languages* (DSL) [vDKV00]. For example, Maple[6] is a computer algebra system specialized for calculating specific mathematical formulas. Such specialized languages are very effective in some domains. However, often it makes sense to reuse as much of an existing host language as possible when adding DSA.

ii. *Embedded Domain-Specific Languages* (EDSL) integrate DSA into a host language [LM99, Kam98]. The host language is often a general-purpose language, e.g., Java, which is augmented with declarative constructs to support one or more domains of interest. An example of an EDSL, is SQLj [SQL03], an embedded SQL engine for Java.

iii. Another way to support DSA is indirectly via *general-purpose software abstractions*, which are first added to a host language in order to (a) make it extensible to support EDSL-like constructs, or (b) to support generic meta-modularization[7] mechanisms not found in the original language. An example of an general-purpose software abstraction for supporting extensibility is the support for *attributes* in .NET [Pro02], or *annotations* added recently to Java. An example of a generic meta-modularization mechanism that can be used to support EDSL is *Aspect-Oriented Programming* [KLM$^+$97] as supported, e.g., by AspectJ [KHH$^+$01, Lad03] (§2.4). The general-purpose software abstractions are not specific to any particular domain and can be used to support several EDSL within the same host language.

In the following, several drawbacks are discussed, which prevent DSA supported by the first two alternatives (DSL and EDSL) from being widely accepted as a variability mechanism for product-lines. Attribute-based DSA are discussed in section §3.1, while AOP-based DSA are explained in section §3.5.3.

[6] http://www.maplesoft.com/

[7] These generic modularization mechanisms need to access the meta-model of a language to enable factorizations that are not possible with inheritance and composition alone.

- *High start-up costs*. DSA realized with the first two approaches have the highest start-up costs compared to other variability mechanisms, e.g., component libraries, and require careful planning of the expected variability. Adding the DSL start-up costs [Hud98], schematically shown in case (b) of Figure 2.1, such as defining grammar extensions and interpreting declarative constructs, to the product-line start-up costs devoted to domain engineering [Har01] and variability representation [Har01], may result in an unacceptable payoff point for a product-line. This motivates the need for technology that reduces the DSL introduction costs in iterative product-lines.

- *Difficult to evolve.* DSA are also difficult to maintain in order to support product-line evolution. As with every software system, it is very probable that the proper declarative abstractions cannot defined clearly from the beginning. The development is more iterative in the early phases of the product-line. The cost of changing DSA is usually higher than, e.g., that of maintaining libraries or object frameworks. The implementation of the abstractions and their representation needs to be modified, which affects the language grammar and its parsing. The DSA technology should support experimentation during the iterative product-line development.

- *Accidental costs.* DSA frameworks introduce additional costs not only for implementing the DSA itself, but also for educating the developers to use the abstractions. Last but not least, DSA introduce unnecessary external dependencies of the product-line on third-party frameworks, needed to implement the DSL additions, which may increase the product-line maintenance costs in the long term. There is also a lack of standard parsing tools and DSL supporting API-s in languages, which renders the implementation of reusable DSA difficult. The DSA technology should be part of the programming language supported by the language vendor.

2.2 Software Containers

Chapter §3 shows how attribute-based DSA can be used as a low-cost DSL alternative to represent domain concerns in source code. Selecting a variability mechanism is not enough to create a product-line. A structured way to organize the domain assets of the product-line is needed. An *architectural abstraction* [Har01] could be combined with one or more variability mechanisms, in order to support product-line creation and evolution. The product-line architecture [Har01] defines and enforces an implementation methodology to represent and to organize the product-line assets [CN02]. For example, when OO inheritance is used as a variability mechanism, a well-defined hierarchy of the domain assets could be created.

The product-line architecture should define clear extension points for the developers that extend the product-line (§2.1.3). The DSA constructs are connected with the domain assets implemented in the form of OO libraries (§3.1.3). However, when the DSA become easily to im-

plement, there could be a lot of unstructured DSA constructs added to support the product-line. The unstructured DSA interpretation can become a bottleneck for the product-line evolution.

Several architectural styles [Har01] have been explored for product-lines, e.g., data-flow architectures and virtual machine architectures [Har01]. The interest in this book is (a) in low-cost mechanisms that support the iterative view of mobile product-line development and (b) in automated reuse of product-line assets in specific applications. Following the current trend of component-based development [SGM02], supported by today's general-purpose OO languages, such as Java and .NET, this book focuses on component architectures [Dol99, SGM02, CE02, Aßm99] based on *software containers*. Containers place a clear distinction between the domain functionality, organized as low-cost DSA, and the rest of the application specific functionality.

The container abstraction is usually related to server-side enterprise container implementations, e.g., COM+ [Gor00] and EJB [J2E03], and could be described by a combination of software patterns [VSW02]. The container abstraction can be used to organize product-line assets for any application domain. A software container serves as a high-level architectural abstraction, used to organize common domain functionality as services and introduce them nearly transparently to an application[8]. This section starts with an overview of two existing enterprise container implementations, COM+ and EJB. The remainder of the section explains how containers can be used in arbitrary domains, and why is it beneficial to use them to organize mobile product-lines.

2.2.1 Microsoft COM+

COM+[9] [Gor00, Ewa01] is one of the first wide-spread enterprise containers. It uses an underlying component model based on Microsoft DCOM[10] [DCO02] technology, which allows components to reside nearly transparently in any networked machine. COM+ is mainly intended to facilitate implementation and maintenance of big enterprise applications. Some of the COM+ services, e.g., catalogs, that organize and maintain sets of components, can be used to support any distributed application. A COM+ application can be configured either programmatically, using any (script) language that can access COM objects, or via GUI tools.

A COM+ application consists of two types of DCOM components: (a) *serviced* components that use the services offered by the COM+ container, and (b) *client* objects that interact with the serviced components. Each COM+ component has its own identity and can be manipulated individually if needed. The idea behind COM+ (as with any container) is that application components that use the COM+ services, are developed by leaving out the exact details of ser-

[8] Unlike most real-world containers, a software container offers some kind of support (services) to the object found in (managed by) it.

[9] The term COM+ is used here synonymously with Microsoft Transaction Service (MTS).

[10] DCOM stands for Distributed COM, whereas COM itself stands for Component Object Model. Both are proprietary component models.

vice configuration to COM+. The following discussion of COM+ services is based on [Gor00].

One of the basic services offered by COM+ is a fine-grained security model based on *roles*. Each role has a well defined set of permissions that specify, which components and component resources, e.g., databases, the role is allowed to access. Users are assigned to roles and this determines their permissions.

Transaction management is another service offered by COM+. A transaction is a sequence of operations that is treated as a single logical operation, and enjoys atomicity, consistency (in the terms of invariants), isolation (no side effects), and durability (introduced changes are not lost). The underlying DCOM technology supports distributed transactions relying on a distributed two-phase commit protocol [CDK01]. Users can also define custom atomic resources, using Compensating Resource Manage (CRM) service, to allow any kind of created or acquired resource to take part in a transaction.

COM+ helps to manage the internal state of component instances by distinguishing between *stateless* objects, *cached state* objects, and *persistent* objects. The internal state of stateless objects is not important and does not need to be preserved. The cached state objects can have their state preserved while the application is running in a shared property manager (SPAM)[11], indexed by string keys. The state of persistent objects can be stored in persistent storage directly or in a relational database.

Object state is related to object life-cycle management. COM+ manages the life-cycle of objects in order to transparently support scalability[12]. To handle a large number of sporadic clients for an object, COM+ uses Just In Time Activation (JIT). JIT creates an object instance only when it is needed, and passivates it during the idle time to save the system resources. This service is complemented by object pooling, where several instances of a component are kept and reused to re-personify[13] serviced objects every time a specific object is needed by a client. To achieve pooling, a client does not use COM+ objects directly, but via a *moniker*, which is a DCOM abstraction of an instance handle. A moniker can be used to represent any resource. Database connections are also COM objects, so they can be pooled using the same mechanism too.

An important feature for an application is the ability to synchronize its own activities. COM+ deals with synchronization by supporting different threading models known as *apartments*. A component is activated (invoked) always within a specific thread model. The apartment model is specified when the component is deployed in a COM+ application by setting the right apartment type as part of the configuration. Access to multi-threaded apartment (MTA) objects is synchronized by COM+. A related COM+ concept is an *activity*, which represents a set of objects that are used by a single client (similar to a session). Synchronization can also be

[11] Shared Property Application Manager.
[12] COM+ can also handle load balancing by distributing requests in a cluster of several machines.
[13] This is similar to *virtual instances* in Java EE [VSW02].

controlled at the level of the individual activities.

Another related service is Microsoft Messaging Queue (MSMQ), a publish-subscribe event service often used by COM+ applications. MSMQ can be used alone without COM+, but it is often used for asynchronous message-based object invocations inside COM+. COM+ applications can also use lightweight directory services[14] to locate resources and other components via Microsoft Active Directory service.

One of the problems with using COM+ in the past has been its complex programming model [Ewa01]. This complexity comes from the fact that DCOM and COM+ add several layers of abstraction, while preserving the underlying details. COM+ *serviced* components are DCOM components, usually written in C++. DCOM introduces an interface declaration language (IDL) and several implicit programming conventions. COM+ adds another layer of explicit and implicit conventions to DOM that must be known and followed to use COM+ successfully. The preservation of all these details of the programming layers made sense back in the early days of COM+ when the technology was not mature. It was useful to be able to experiment with the technology in order to understand the properties of its abstractions and their implementations. Having access to the implementation details enables also manual optimizations for software that has to run in slow machines. Such a detailed programming model with little automation can work only for relatively few and small applications.

In last ten years the situation has quickly improved. As the computer hardware becomes much cheaper than software and there are faster machines, there are also more and larger applications. Language-based abstractions, such as those of .NET [Pro02] language families, supported by virtual machines are not considered slow anymore. Long experience with COM+ and DCOM have revealed the repeated coding idioms that could be abstractions on their own. With .NET, [Pro02] Microsoft introduced a new, simpler and less restrictive programming model for DCOM [Box00] and COM+ [Low01], based on meta-data and explicit attributes. The COM+ support in .NET is part of the `System.EnterpriseServices` namespace [Low01].

2.2.2 Java EE and Enterprise Java Beans

Similar to COM+, Java Enterprise Edition (Java EE) [J2E03] is a container-based technology for enterprise server-side applications. Java EE is based on Enterprise Java Beans (EJB) [EJB03] component model, relying on Java [Jav04] component model (beans and remoting)[15]. Unlike DCOM, the Java component model is simpler to use. Java itself (unlike C++) belongs to the family of languages that maintain meta-data as part of their binaries[16]. This simplifies the programming model, e.g., the discovery of the interfaces implemented by a component, is directly

[14] Lightweight Directory Access Protocol (LDAP).

[15] These are also proprietary component models.

[16] The Java binaries (*.class files) are actually pseudo-binaries that require a virtual machine to be interpreted and cannot be executed directly.

supported by the language. Java technology (and Java EE) is an example of the second wave of enterprise containers and technologies used to build them, where more abstraction layers can be afforded as the hardware becomes faster.

While Java EE and COM+ (MTS) offer more or less the same set of services, the terminology is often different. Java EE, for example, uses the term *role* to denote different stages of an application development and life-cycle, e.g., enterprise bean provider, application assembler, deployer, container provider, whereas COM+ does not define any specific terminology for the developer roles. The components are called *beans* in Java EE.

The programming model for the container services is different too. Java EE distinguishes between various types of beans, such as, stateless, session, entity (persistent), and message-driven (event based). Unlike COM+, where the stereotype of a component is specified as part of its configuration attributes, in Java EE components state their type explicitly based on required interfaces specified as part of their implementation and other implicit[17] coding rules, e.g., required method names.

Services offered by Java EE include, for example, security, transaction management, naming, message-based communication and timer service. There are slight differences between these services and the corresponding COM+ services. For example, security in COM+ is not concerned with user authentication, since this is handled by the operating system, whereas Java EE deals with user authentication itself, because it is intended to work in multiple different server operating systems. Naming (JDNI) in Java EE corresponds to Active Directory services in COM+, whereas message-based communication via message-driven beans is similar to MSMQ. A Java EE bean can use the timer service to register callbacks to be called at a given time or periodically. The timer counters are persistent and callback registration can be treated as a transactional resource (as with CRM in COM+). Object pooling and database connection pooling are also handled by Java EE. Java EE uses the term *virtual instances* [VSW02] for handles to the pooled objects (in COM+ a handle is implemented by a moniker).

The most interesting aspect of Java EE is the evolution of its programming model. EJB, up to version 2.1 [EJB03], makes use of required interfaces and pseudo-syntactic rules (coding conventions), such as naming rules for method names. Creating a bean implies defining one or more required interfaces. Furthermore, beans that need some form of life-cycle management, must implement required callback methods, e.g., `ebjActivate()`. Most of these actions are repeated with few variations and could be automatically generated. It is unclear why EJB did not define from the beginning proper abstractions, to deal with such tedious programming tasks, that can be easily automated and made transparent[18]. Third-party tools, e.g., xDoclet [xDo03], appeared to deal with such inconveniences of the EJB programming model.

[17] That is, many Java EE component coding rules are not part of the language syntax and cannot be enforced explicitly by the Java compiler.

[18] When EJB appeared its programming model was significantly easier to use compared to COM+. The .NET programming model for serviced components is now simpler than EJB.

EJB is also moving toward a simpler programming model, based on attributes [BCVM02]. The latest EJB specification [EJB04] uses predefined attributes to decorate component implementations, which are now *plain old Java objects* (PoJo-s) with very few implicit coding rules.

2.2.3 Using Containers Beyond the Enterprise Domain

Since the term *container* was made popular by COM+ [Gor00] and EJB [J2E03], many people use the term synonymously with enterprise containers. Enterprise containers, such as, COM+ and EJB, are often referred to as *heavyweight containers* [POM03], which means:

a. COM+ and EJB are container implementations for a hardwired problem domain. The containers are defined exclusively for enterprise server applications, usually involving relational databases as back-ends. These container implementations cannot be reused in other domains.

b. Even for the enterprise domain, COM+ and EJB containers offer only a predefined set of services. EJB and COM+ containers cannot be easily extended to introduce new kinds of services into the domain [POM03].

Heavyweight container implementations are tightly coupled with the specific problem they solve. The container as an abstraction is domain independent and open to accommodate new kinds of services. In particular, the container abstraction can be used to conveniently implement automation of application families (product-lines). The functionality offered by a product-line (the reusable assets of the product-line) can be seen as a set of services offered by the product-line container abstraction. The specific application components managed by the container (managed components), known also as *serviced components* [Low01], use the container services as shown in Figure 2.5. The container service variability parameters are given as part of the deployment specification of an application component in the container. The variability parameters are shown in Figure 2.5 in the form of a deployment interface. The actual implementation may vary depending on the container. The serviced components use the services provided by the container. The connection between the container services and the serviced components is known as *service injection* [Low01].

The container abstraction enables a uniform treatment of the services modeling the assets of a specific domain, and the generic services modeling cross-cutting concerns (§2.4) that may be shared by many applications of different families. The developers of a product-line can manage generic concerns, such as data persistence, within the same container framework as the specific assets identified in a particular domain, e.g., image adaptation required by mobile image processing applications. The container abstraction enables the organization of an *extended* view of a product-line (Figure 2.6), whose assets include generic concerns used in the same way, as the assets of a specific application family. An extended product-line is made up of more specific intersecting sub-families that share common assets with each-other, having a network of domains

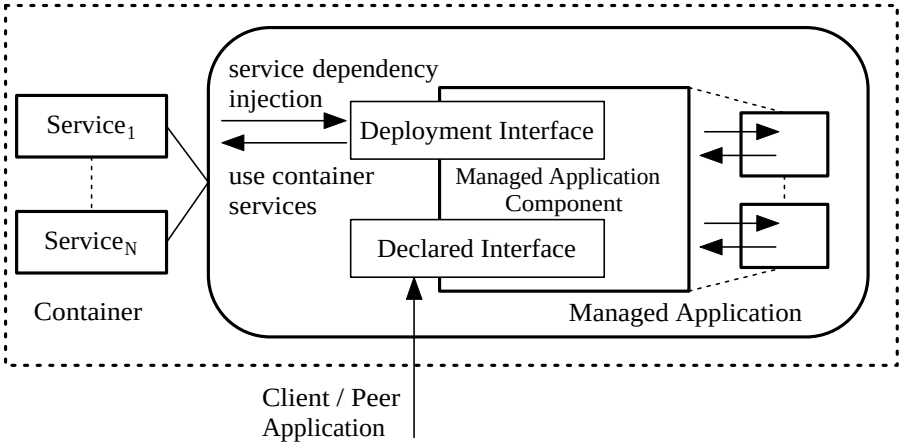


Figure 2.5: The Container Abstraction

[CE00]. The shared set of services is represented by the intersection of the different sub-family (assets) eclipses in Figure 2.6. The generic concerns are needed by most of the application sub-families within the extended product-line, whereas the application family specific assets are needed only within a family of applications. The uniform treatment of the services enables better share and reuse of the specific services developed for a family, in another family or set of such.

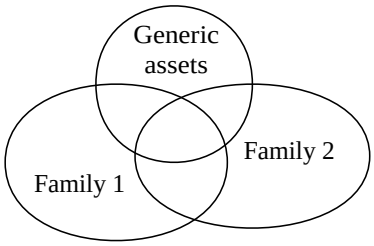


Figure 2.6: The Extended Product-Line

Using the container abstraction to organize a product-line, increases the transparency of reusing the domain services, and results in a better separation of concerns [Par72], because of:

- *Inversion of control.* The container provides a software indirection for injecting dependencies. The *dependency injection* pattern [Fow04], also known as *inversion of control* (IoC)[19], implies that a component does not explicitly obtain the services it needs but

[19] IoC is also expressed as the `Hollywood Principle`: *"Don't call me, I'll call you."* [Joh03]. See [Fow04] for

rather, declares its dependency on them by following certain conventions. The declared services are then provided transparently to the component, as necessary, by the environment [GL96, CF05]. Dependency injection *"moves the responsibility for making things happen into the framework, and away from application code"* [Joh03].

The container connects the product-line services and the application-serviced components in a transparent way. When a service component library is used directly, the exact interface of the service components should be known to the application and be referenced explicitly (§2.3). This is problematic, not only when the interface of the library changes, but also when the library is maintained and several versions of it appear. When inheritance is used, e.g., in a framework, the fragile base-class problem can be encountered [Mez97a, SGM02, SDN04], as the base classes evolve. The container abstraction separates the need for a service, from the need to explicitly reference the implementation of the service. The exact service references are introduced transparently to the managed application component.

- *Improvement of modularization.* The concerns addressed by a product-line are often of a cross-cutting nature. They involve functionality that is needed in more than one place in an application, and negatively affect the overall modularization [MO03, Ost03]. The transparent dependency injection, offered by the container, allows developers to focus on those concerns that are specific to application components. The cross-cutting service concerns of the product-line are then introduced automatically by the container.

- *Automation of the development process.* The container abstraction automates the development of applications in a product-line. Product-line automation significantly decreases the development and maintenance costs. The maintenance is improved because the container offers a centralized point to isolate specific middleware services from an application. The container abstraction offers a structured and centralized *object transaction monitor* (OTM) [Dol99]. This makes it easier to maintain and port mobile application families when the underlying middleware changes.

- *Implementation architecture.* The container abstraction is a high-level architectural abstraction. It does not replace the usual techniques for product-line implementation, e.g., libraries and code generation. It rather serves as an umbrella for all possible technologies that can be used. One or more technologies can be combined together, to implement the container abstraction in a way that is convenient for a certain domain.

- *Separation of developer roles.* As stated in the Java EE specification [J2E03], the container abstraction helps to divide developers in clear distinguishable categories, according to the application development roles: (i) developers that implement the container functionality; (ii) developers that implement services for the container; (iii) developers that implement

a discussion as why the term *dependency injection* is preferable to IoC.

managed applications using the container; (iv) developers that deploy the managed applications into the context of a container. These roles are clearly distinguishable and could be used to properly structure all developers that work with a product-line in an organization.

The container automation has also a few drawbacks depending on the implementation:

- *The need to learn new coding conventions.* Explicit or implicit coding conventions must be followed by a component implementation if it is going to be used inside a container. These conventions should be learned and properly used by the programmers. For example, an object of a managed virtual instance [VSW02] must not contain methods that return the `this` pointer directly. This coding convention is required, because otherwise a client would always obtain a different physical reference to the same logical object.

- *No specialized language support for the coding conventions.* Cheeking implicit coding conventions automatically may be difficult. Explicit coding conventions can be automatically checked, but must be introduced and learned. Explicit coding conventions could be in the form of required interfaces, e.g., *EJBMessageBean* in EJB [EJB03], or based on some form of DSA, e.g., using additional keywords. Implicit coding conventions are harder to check automatically than the explicit coding conventions are, but require no special syntax to be supported.

- *Traceability problems because of indirection.* Another related problem is testing and debugging. The indirection introduced by the container makes it difficult to understand the overall functionality of a managed component. This makes it more difficult to debug and test the implementation of the managed components. Nevertheless, debugging can be supported with proper tools and built-in container support for traceability (§4.1.9).

2.2.4 Mobile Containers

This book introduces the use of containers to organize mobile software product-lines for J2ME MIDP [Jav05, Mob02] (§5). The container abstraction needs to be specialized in order to reflect the characteristics of the MIDP [Mob02] domain. The mobile software is usually a combination of server-side software and client-side software on the mobile device. Because of limited capabilities of mobile devices, it is desirable to execute as much of the application functionality as possible on the server-side. Ideally, mobile applications should be thin clients. This is not possible for most mobile applications, because a mobile device usually has only sporadic network connectivity. The mobile applications should be able to function even when no network connectivity is available, or to keep at minimum the on-line time, to reduce the connection costs.

Support on the server-side is also needed for mobile applications. For example, images need to be resized to adjust their resolution (or HTML pages need to be converted to WAP

[WAP03] format, in a WAP-portal), before they are sent to a mobile device. The data need to be prepared on the server-side, before are sent to a mobile device. Data adaptation can be done transparently from the rest of server-side applications, by using some form of the adaptive-proxy pattern [FGBA96], before the data are sent to a mobile device application.

To address mobile software product-line issues, this book introduces the notation of a *mobile container*, which is a special kind of container for managing mobile client applications (§5). A mobile container helps to organize services needed by a mobile client application and can be used to support product-lines for mobile applications. The implementation of the offered services is split into a server-side part, and a client-side part running on the mobile device. The client-side implementation must be optimized to reduce the number of abstraction layers resulting from the container indirection. For a managed application running on the mobile device, the container services are transparent. It is the container's responsibility to coordinate the client and the server parts and to provide the requested services to the client applications.

Chapter §5 elaborates more on how mobile containers can be structured and about the technology that can be used to build open *container families*, which can be customized to support product-lines for more than one domain. The container implementation introduced in chapter §5 is based on a generic Java-based framework, and is open for adding new services for arbitrary domains. The implementation is then specialized with a set of service plug-ins for J2ME MIDP [Jav05] applications.

2.3 Container Implementation Techniques

The ultimate goal of a container is to provide services transparently to its managed components. Ideally, the components are developed in isolation from the environment [GL96, CF05] where they are used. The component implementation does not need to know anything about a specific container environment. The component, nevertheless, needs to specify what kind of services (other components) it expects from the environment.

The lightweight coupling of a component with the container environment can be achieved using additional levels of abstraction. Instead of providing a component with direct access to the needed services, as in case (a) of Figure 2.7, an agreed convention based on some form of *inversion of control* [Fow04] abstraction can be used, as shown in the case (b) of Figure 2.7. In case (b), the managed component only declares the services it needs, e.g., by using DSA constructs. The managed component does not have a direct knowledge of any particular service implementation. The container contains the functionality to provide the requested services to the component. The container implementation could provide only a predefined set of services, or the container can have programmable logic to be extended to provide arbitrary services (§5).

The most interesting point in a container implementation is, thus, how to define abstractions for *programmable dependency injection*, i.e., how to define dependencies of a managed

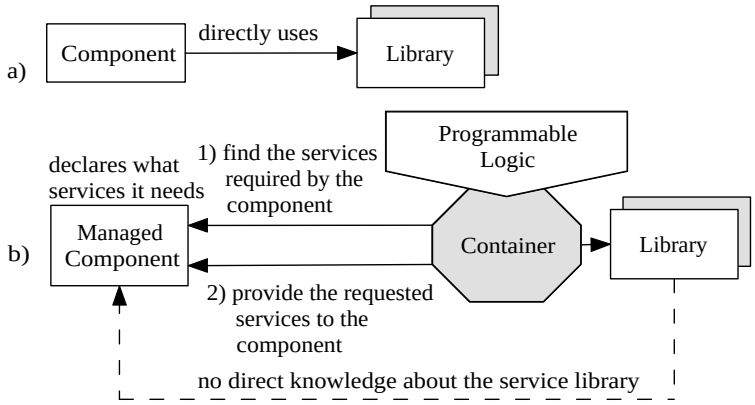


Figure 2.7: Container Service Dependency Injection

component programmatically after the component has been created. Based on the terminology defined in [Aßm03], this section distinguishes between *invasive* and *non-invasive* container implementation techniques and gives some examples for each case. Each of the discussed implementation techniques can be seen as a fine-grained variation mechanism.

2.3.1 Non-Invasive Container Implementation Techniques

Non-invasive container implementation techniques are based on pure object-oriented (OO) mechanisms mainly, interfaces, composition, inheritance and reflection. They do not need any special language support, and can be easily implemented in any existing OO language[20]. In a non-invasive technique, there are no components / units more privileged than the others with respect to their power to access and modify other components / units. The entire component access goes through the declared (public) interfaces and uses only standard OO techniques, i.e., non-invasive techniques respect the OO encapsulation.

There are three main ways to inject component dependencies using non-invasive composition techniques: *constructor injection*, *setter injection*, and *interface injection* [Fow04]. Almost all implementations of these techniques require reflective introspection support, i.e., internally they use some form of a reflection API. The discussion of non-invasive techniques in this section is based on [Fow04]. To demonstrate the *constructor injection* and the other techniques, an example of a Java MIDP based game score class [Jav05] will be used, shown in Figure 2.8. The example is made of a `GameScore` class, whose internal state needs to be saved in the local memory of a mobile device.

[20] Techniques for non OO languages also exist, but these languages will not be considered.

```
1  interface Serializer {
2      public void Save(string id, byte[] data);
3      public byte[] Load(id);
4  }
5
6  class RawSerializer() implements Serializer { ... }
7
8  class GameScore() {
9      public GameScore(Serializer sr) {     ...      }
10
11     public void Save() {
12         sr.Save(this.GetId(), this.GetData());
13         ...
14     }
15     ...
16 }
```

Figure 2.8: GameScore Serialization Example

The `Serializer` interface (lines 1 - 4) in Figure 2.8 declares operations for saving any data in the device memory, that can be represented as byte arrays. The `RawSerializer` class, in line 6, implements the actual serializing functionality. To keep things simple, it is assumed that the `GameScore` class contains two methods `GetId()` and `GetData()`, which return a string ID for the object (the key), respectively its data as a byte array (the value). An automatic implementation of the `GetData()` method will be shown in section §2.3.2.

```
1  class GameScore() {
2      private Serializer sr = new RawSerializer();
3
4      public GameScore() { ...      }
5
6      public void Save() {
7          sr.Save(this.GetId(), this.GetData());
8          ...
9      }
10     ...
11 }
```

Figure 2.9: Direct Dependency Access

To demonstrate the constructor injection, the `GameScore` constructor expects an object implementing the `Serializer` interface. The `GameScore` class can be connected directly to `RawSerializer`, as shown in Figure 2.9, which is equivalent in functionality and simpler to implement. This method would tie the `GameScore` implementation to the `RawSerializer` class. The connection between `GameScore` and `RawSerializer` is explicitly coded. The programmer decides about the exact implementation of serialization to use, at the time the

`GameScore` class is coded. Thus, the direct connection is the least flexible dependency method.

```
1 MutablePicoContainer pico = new DefaultPicoContainer();
2 pico.registerComponentImplementation(RawSerializer.class);
3 pico.registerComponentImplementation(GameScore.class);
4 GameScore gameScore = (GameScore) pico.getComponentInstance(GameScore.class);
5 gameScore.Save();
```

Figure 2.10: Constructor Dependency Injection with PicoContainer

- *Constructor injection*. Figure 2.10 shows how the `GameScore` class of Figure 2.8 and the `RawSerializer` instances could be coupled programmatically using PicoContainer [Pic03]. PicoContainer assumes that the classes to be connected already define constructors for the appropriate service parameters. The `pico` container object contains all dependency information required to connect the `GameScore` and `RawSerializer` instances. To connect `GameScore` with a new `Serializer`, for example `RemoteSerializer` that can save data to a server when there is a network connection present, only the `RawSerializer.class` need to be replaced with the new `RemoteSerializer.class` in the container. The PicoContainer indirection assures that `GameScore` will never know directly about a particular `Serializer` implementation it is using. Having a `Serializer` interface is not a requirement for PicoContainer. It can also connect instances of classes directly by examining their interfaces and finding the appropriate services based on the method signatures by using reflection.

- *Setter injection* is similar to constructor injection, but makes use of the assumption that component classes implement appropriate setter methods for service components they depend on. In the example of Figure 2.8, this could be accomplished by having the `GameScore` class define a `setSerializer(Serializer sr)` method. A container implementation that relays mainly on setter injection is Spring Java / Java EE Application Framework [Spr04]. There are several semantic differences between setter and constructor injection approaches. The setter injection provides more flexibility:

  - An object can be created even though not all dependency objects it requires are available. This is not possible with the constructor injection, where all dependencies must be known when the object is created.
  - When using the setter methods, dependencies can be changed during the lifetime of an object. Often, a certain service needs to be attached to different existing instances and this is not possible with constructor injection. This can be useful, for example, when a virtual instance [VSW02] in an EJB container needs to be connected with a previously saved object state. With constructor injection, a new object must be always created, whereas with setter injection existing objects can be reused.

There are also some drawbacks that result from the flexibility of the setter injection:

- Given an existing component instance, it cannot be guaranteed that the instance is fully and properly initialized in the context of a container, or whether it needs more dependencies to be injected. Repetitive checks need to be performed every time an object instance is used to make sure that the dependencies are properly configured.
- An explicit search must be carried out by the container to find the appropriate setter methods and decide for the right service objects to be passed, based on the setter method signature. This requires scanning the component interface for all possible methods, finding whether there are any setter methods, and matching the signatures of setter methods to the service components. Hence, the setter injection takes in average longer than the constructor injection. In the later, only the constructors need to be checked when new instances are created.

The setter injection is an example of what will be called a *pseudo-syntax* convention to add custom semantics to a language (§3.2.2) without changing its grammar. A pseudo-syntax construct is not an explicit part of the grammar syntax of the language. It is a *coding convention* and is not checked by the compiler. In this case, the *set* prefix is used as an implicit *syntactic pattern* to define the setter injection.

☞ *Interface injection* is a generalization of the setter injection. Instead of having required constructors or setter methods in the game score example, an interface `InjectSerializer` with a single method `injectSerializer(Serializer sr)` is defined, and the `GameScore` class will implement it. Based on the `InjectSerializer` interface, an interface injection container implementation can couple the object and the service[21].

The interface injection pattern has the same benefits and drawbacks as the setter injection. However, interface injection improves the search by making the injection points explicit. The container needs to search only for the implemented interfaces and their methods to find what services need to be provided to a given component. No pseudo-syntax checks are required, such as, the check for the *"set"* method prefix. Adherence to pseudo-syntax constructs cannot be checked at the compile-time. Hence, employing pseudo-syntax constructs as a mean of interfacing with the dependency container is more error prone that using interfaces for the same purpose. Pseudo-syntax checks rely also extensively on string pattern matching, which is a resource consuming task when done repeatedly at run-time.

When interfaces are used to denote custom semantics by tagging class implementations, they are known as *marking interfaces* [NV02]. The interfaces that require trivial method implementations just for the sake of service injection will be treated as marking interfaces here. In an invasive container (§2.3.2), a marking interface does not need to contain methods that must be implemented in the classes that use the interface. An invasive container

[21] The reader is referred to [Fow04] for an example of a framework that uses interface injection.

can modify a component decorated with `InjectSerializer` interface to properly inject the required serialization service. In non-invasive containers marking interfaces may contain methods.

Marking interfaces can be used to add semantics to components in languages that do not provide explicit attributes, such as, Java 1.4 [Jav04, Att02]. They can be used to declare dependencies, as in the case of injection pattern, or other properties that a component must have[22]. Marking interfaces have also some problems. As chapter §3 discusses, the number of marking interfaces can grow exponential in a system, in the case when interfaces are used to support component variability.

There are also several other non-invasive techniques, less flexible than the ones explained above. They are used less frequently to implement the dependency indirection in a container. Only *service locator* [Fow04, JH04] and *mix-ins* [BC90, Ern99, FKF98, Dug00] will be briefly considered in this section.

☞ *Service locator* makes service acquisition explicit. The idea is to locate the services of interest explicitly by consulting a (well-known) object registry. The registry location could be hardwired in the component implementation, or it can be indirectly passed to it using any of the above dependency injection techniques. In the service locator case, the `GameScore` example would contain code to get the serializer object directly from the object registry as in Figure 2.11. Any directory (LDAP[23]), trading, or naming service can be used as a locator repository for objects. The locator must find the required service object and properly initialize it. The requested registry object path (e.g., a relative path *"LocalSerializer"*, or an absolute path *"jndi://mobcon/services/LocalSerializer"*) can be hard coded in the application, or read from a configuration file.

```
1 | ServiceLocator locator = ...
2 | LocalSerializer s = (LocalSerializer)locator.loadService("LocalSerializer");
```

Figure 2.11: Constructor Dependency Injection with a Service Locator

The service locator offers less automation than the other techniques presented above. Finding services is explicitly encoded, and the kind of service the programmer is asking for, must be exactly know [JH04]. Furthermore, service locator adds accidental complexity to the component's implementation. The service locator is sometimes used as a

[22] A well-known example is the use of the *Serializable* interface in Java. Another example is the *Null Object* pattern [Fow99], where a class defines specific functionality for the null case by using inheritance from a Null (marking) interface.

[23] Lightweight Directory Access Protocol.

complementary technique to other dependency injection mechanisms in a system. For example, JNDI [LS00] is used in Java EE to access secondary application resources, e.g., images and Java Server Pages (JSP).

☞ *Inheritance and Mix-ins* can also be used to emulate indirect dependencies [Mar96]. Mix-ins [BC90, Ern99, FKF98, Dug00] factor out properties that cross-cut more than one class [Rie96]. In the particular case of the containers, a mix-in can contain the context for a container service. Mix-ins can be implemented using multiple-inheritance, or with templates in the languages that support them [SB02, CE00], e.g., with C++ templates, or Java generics.

The inheritance based techniques are usually static[24], i.e., static linking (deployment) of components is required. For this reason, inheritance composition techniques are usually less flexible than other non-invasive techniques. No known container implementation exists that relies exclusively on inheritance techniques. The mix-ins technique appears often in different forms in different systems [BST+94], which is why it was discussed as an separate alternative here.

2.3.2 Invasive Implementation Techniques

If a container is considered to be a strict component system in the OO sense, then only the non-invasive techniques make sense. They respect the integrity of individual components and operate in accordance with the OO encapsulation rules. A container in this case contains non-invasive code to connect the application components with the container predefined services (other well-known components). The container services are just other components. A well-designed container (non-invasive, or invasive) is usually extensible, i.e., new service components may be added.

Another view of a container is as a transformation system[25], much like a compiler / transformer. In this case, the components used inside the container environment need to be somehow transformed, so that the expected dependencies are injected into their inner implementation. The container in this case is a programmable transformer, whose input consists of the components, container services (other well known components), and a specification how to connect the first two with each-other. The specification can be given explicitly in some particular syntax, or it can be implicitly inferred from the implementation of components. The details of the specification are used to create *the glue code* [Hal02] that connects the components with services. The

[24] There are also exceptions, e.g., Rondo [Mez97b].

[25] The term *transformation* is used here to include any generation or transformation technique that requires some form of AST, or equivalent program structure, processing. Sometimes, the same technique can be supported with generative techniques, or with run-time support. No distinction will be made between the two techniques. The run-time support will be considered to be a specialized optimization.

output of such a transformation is a system, where the components and the services are properly connected with each-other by conventional OO mechanisms.

Invasive techniques support the transformation view of a container. Invasive techniques require either special language features[26], or external frameworks that provide means to mutate the internals of a component, surpassing the usual OO techniques. Invasive source code or binary mutations may not be directly supported by the target language, in which the original component is written. Invasive techniques do not respect the OO encapsulation and depending on the implementation, they can change a component arbitrarily.

An invasive technique can introduce changes to the component internals, or just read the component internals to introduce non-invasive changes (§4.1.2). That is, the invasive techniques can be *read-only* or *read / write*. Read-only invasive techniques can be treated as non-invasive. However, the read-only invasive techniques will be classified as invasive here, because they require access to component internals which are not visible through the declared interface of the component. An example is the use of the Reflection API in Java. By default, the Reflection API returns only the public members of a class, and is non-invasive by the definition. The private methods can be accessed by changing member access permission by using the `ReflectPermission` class. This latter case will be treated as a form of the read-only invasive access to a class.

A distinction is made in [Aßm03] between invasive techniques that use *explicit* and *implicit hooks*. A *hook* is defined as an accessible point of the component interface or implementation that can be used to modify the component and to inject code into it. Explicit or declared hooks are hooks that are defined explicitly by the programmer to indicate that the code associated to them will change. An example would be the decoration of a method with an attribute, or a special predefined prefix in its name. Implicit hooks can be any accessible point of a component, even those that are not foreseen by the original developer, e.g., a method call at run-time.

Accessible points of a component depend on the technology used to access the description of the component's implementation. Not all technologies are equally expressive. For example, method internals cannot be accessed using OO reflection alone. Therefore, the code inside a method cannot serve as a hook when the reflection technology is used. The term *hook* will not be used here anymore, because it is also used to name some special callbacks in event-driven applications. The term *hook* in [Aßm03] is more or less used with the same meaning as the term *joinpoint* in Aspect-Oriented Programming [KLM+97], which is more specific and will be used in this chapter. Joinpoints will be further discussed in section §2.4.

Any system that supports some form of access and manipulation of the abstract syntax tree (AST), e.g., JTS [BLS98], can be used to implement invasive containers. Invasive techniques will be illustrated here by an example which uses syntax similar to COMPOST [Aßm03].

[26] Such as, the ability to access and (arbitrarily) manipulate the source AST, or the run-time meta-model of the program.

☞ *Invasive Example.* The service dependency injection in the `GameScore` example of Figure 2.8 could be easily implemented with invasive techniques. The AST of the `GameScore` class needs to be traversed to add / modify the serialization service variable and make it point to the requested service. Given that this case is trivial, the example will rather show how the `GetData()` method of the `GameScore` class could be implemented with invasive techniques. This method returns a byte representation of a `GameScore` object's state.

The example of the `GetData()` method implementation is more interesting, because it shows how the invasive techniques can be used to introduce functionality inside a method. This is impossible to realize directly with non-invasive techniques. Part[27] of the implementation of the `GetData()` method is illustrated in Figure 2.12, by using pseudo-syntax similar to the one found in COMPOST [Aßm03].

```
1  CompilationUnitBox box =
2      compositionSystem.createCompilationUnitBox("GameScore.java");
3  List<FieldBox> fields = box.findField("*");
4  //<<check if there are fields>>
5  MemodBox method = compositionSystem.createMethodBox("GetData");
6  //<<specify method parameters, scope and return type>>
7  //<<add code to the method to init an in-memory binary writer>>
8  for(FieldBox field : fields)
9  {
10     method.addString("binaryWriter.Write(" + field.Name + ");\n");
11 }
12 //<<add code to method to properly return bytes from binaryWriter>>
13 box.addMethodBox(method);
```

Figure 2.12: The Invasive Implementation of the `GetData()` Method

All the member fields of `GameScore` class are accessed, and based on them the code for a new private method `GetData()` is generated. `GetData()` serializes the data of each field in a memory binary stream. The example of the Figure 2.12 accesses first the Java class (GameScore.java) and implicitly parses it in lines 1 - 2. Then, it builds a list of all fields found in the input class (line 3). It creates a new method node (line 4) for the `GetData()` method that needs to be added. The details of the method node, such as its return type and the parameters, are specified in lines 6 - 7 (not shown). The example traverses over all the fields which are found in the fields list. For each field, code to serialize its contents into a series of bytes is added to the new `GetData()` method (lines 8 - 11). Finally the created method is added to the class (line 13)[28].

[27] To keep the discussion simple, it is assumed that the `GameScore` class includes only simple primitive types. Object serialization should follow the containment dependencies to properly serialize all field objects. Checking of the special cases is also omitted from the code.

[28] The code which saves back the class after its modification is omitted.

A similar method `SetData(byte[] data)` that reverses the serialization process could also be generated at the same time.

While a Java source code unit (*.java) is shown in this example, the input could also be a compiled Java class (.*class). The class file (bytecodes) can be processed either at load-time or at run-time if the particular invasive system has the support for that.

As illustrated by this example, even for relatively small tasks, invasive compositions require tedious coding. This is an inherent property of API-based generation systems [Voe03a] due to the programming indirection. Rather than writing the source code directly, code that generates the code is written. The indirect code has also to deal with each special case. Code templates and textual pieces of code can be used to shorten the amount of code that needs to be written programmatically and debugged indirectly. The `GetData()` method could also be created from a predefined source code template. The template would then be bounded [Aßm03][29] to the specific code generated to serialize each field. Suitable predicates, which are used to control the AST node selection and iteration, as those in [Vis01a], can also reduce the needed work.

2.3.3 Non-Invasive versus Invasive Techniques

Most software composition problems, such as the dependency injection, can be solved either with non-invasive or invasive techniques. However, there are several differences between the non-invasive and invasive techniques with regard to:

- *Automation* - invasive techniques help to achieve more automation, because they can be customized for the problem at hand. Invasive techniques reduce better the accidental complexity of the solution.
- *Flexibility* - invasive techniques are more flexible for making changes inside components. Invasive changes are either not possible with non-invasive techniques, or require many explicit indirection layers to be implemented.
- *Dynamicity* - non-invasive techniques enable changing the binding of the components with services at the run-time. Invasive techniques are static: they usually operate over the AST of a program. Invasive techniques could also be implemented dynamically when appropriate run-time support is provided.
- *Ease of the implementation* - non-invasive techniques require only OO mechanisms which are found by default in many languages. Invasive techniques require meta-language extensions or external frameworks that need to be maintained explicitly as the core language changes.

[29] That is, template parameters will be replaced by the specific code for the use-case.

Depending on the particular properties of the specific systems, sometimes dynamicity is preferred to flexibility and automation, and sometimes using existing language support is preferred to using external frameworks. For mobile containers, the declarative language constructs are supported with attributes, so that the static invasive techniques make more sense. Maximum automation is needed and the generic dynamicity of non-invasive compositions is not required in the addressed MIDP [J2M02b] systems (§5). Section §2.1.2 listed several preferable properties of the variability mechanisms for mobile product-lines. Table 2.1 summarizes how these criteria are supported by invasive and non-invasive techniques, in the context of attribute enabled mobile product-lines.

Criteria	Invasive	Non-Invasive
Explicit programming model, possibly DSL based	+	-
Enable automation	+	-
Low start-up / evolution costs	-	+
Domain-specific optimizations	+	-

Table 2.1: Techniques for Attribute Mobile Containers - Invasive versus Non-Invasive

As shown in Table 2.1, invasive techniques can be used to support an explicit programming model with DSA at the language level. The focus is in the invasive interpretation of explicit joinpoints modeled with attributes. Invasive techniques can be used to automate attribute-based DSA transformation and also to process implicit joinpoints (§2.4). Explicit attributes, combined with invasive techniques, help to reduce the accidental complexity of the programming model. The last line of Table 2.1 shows that, for mobile containers, invasive techniques also support better the optimizations, e.g., specific selection of service code, or reduction of the abstractions being used.

The only benefit of non-invasive techniques with regard to the criteria in Table 2.1 is that they do not require any special language support and have lower start-up costs. That is, non-invasive containers are easier to implement. However, non-invasive techniques do not support an explicit declarative programming model. This means that more lines of code need to be written with non-invasive techniques. Therefore, non-invasive techniques offer less automation than invasive ones. Appropriate lightweight attribute-based DSA technology can help to reduce the startup costs of invasive DSA techniques and to preserve the DSA automation benefits.

Invasive techniques have several liabilities that need to be addressed:

- *Lack of language technology support.* Invasive techniques may not be directly supported by the language. Source-to-source transformers require access to the AST of the component code. For languages that support binary meta-data, such as Java [Jav04], transformations at binary level can be used. Some equivalent of source code AST is preserved

and need to be accessed and manipulated. Not all OO languages offer such AST access as part of their libraries. Third-party parser tools need to be used. These tools should be explicitly maintained when the target language changes. Third-party meta-programming frameworks increase the overall cost of the container implementation[30], and could result in higher costs for a product-line, as compared to the non-invasive techniques.

- *Transformation side-effects.* The flexibility of invasive techniques raises new problems for debugging and traceability. Transformation side-effects[31] may not be predictable in all cases. Despite of the flexibility of invasive techniques, developers need to be conservative in the way they use them (the so-called *sound* compositions in [Aßm03]). Explicit extension points, used in frameworks and explained in section §2.1.3, can be combined with the invasive techniques to narrow the range of transformation possibilities in a product-line and reduce the transformation side-effects.

2.4 Aspect-Oriented Programming and Product-Lines

The transformations of attribute-based DSA (§3.1) that are presented in this book can be supported with any generic invasive meta-programming system. This section discusses Aspect-Oriented Programming (AOP) [KLM+97] as a generic technique that can be used to support invasive compositions. AOP is used to implement horizontal[32] transformations [GSCK04] within the same system meta-level, usually within a single programming language. Arbitrary meta-program generators and DSA can be used to implement horizontal, but more often vertical transformations, which cross more than one meta-level (§3.5.3). There is an overlap of various program transformation technologies that affects also attribute-based transformations. Attributes can be used to carry either horizontal, or vertical semantics and serve as a bridge that enables AOP engines, which can access attributes, e.g., AspectJ [Lad03, KHH+01][33], to support vertical transformations that can be expressed with attributes.

This overlap has several consequences. More than one technology can be utilized to achieve the same transformations. All AOP examples given in [Lad03], can be equally solved with any invasive transformation framework, even though some more explicit context book-keeping data may be occasionally needed. For example, the AOP techniques could be explained in terms of hooks in the COMPOST [Aßm03], a generic invasive composition system. It is, thus,

[30] For example, third-party tools used to parse Java code or manipulate Java binaries have difficulties moving from JDK 1.4 to JDK 1.5. JDK 1.5, introduced a lot of new syntax and the binary meta-data were slightly enriched. The Java vendor (Sun) does not support any transformation interface for Java binaries.

[31] For example, joinpoint matching with pointcuts in AspectJ [KHH+01] may match also program elements that were not intended as the system evolves.

[32] Horizontal transformations have both the domain and the co-domain in the same MOF [Met02] level. Vertical transformations have the domain and the co-domain in different MOF meta-levels.

[33] With support for Java 1.5 annotations.

often not clear when to choose one technology or another. There is also no clear boundary on the limits of what can be achieved with specific technologies, such as AOP.

This section elaborates more about the position of AOP for supporting generation. AOP techniques will be compared with DSA in section §3.5.3. This section considers AOP from a technical point of view as an invasive transformation technique. While the focus will be on attribute transformations, much of the discussion in this section is general and applies not only to mobile product-lines.

2.4.1 Introduction to AOP Techniques

Aspect-Oriented Programming (AOP) [KLM+97] comprises a set of technologies aimed at better modularization of cross-cutting concerns found in software systems. This section discusses AOP from a technical perspective. The interest will be in the transformation mechanisms used by AOP and not in the modularization issues [Ost03]. The discussion of AOP technology is based mainly on the AspectJ programming language [Lad03, KHH+01][34]. The scheme in Figure 2.13 illustrates the relations between the main components found in AOP terminology.

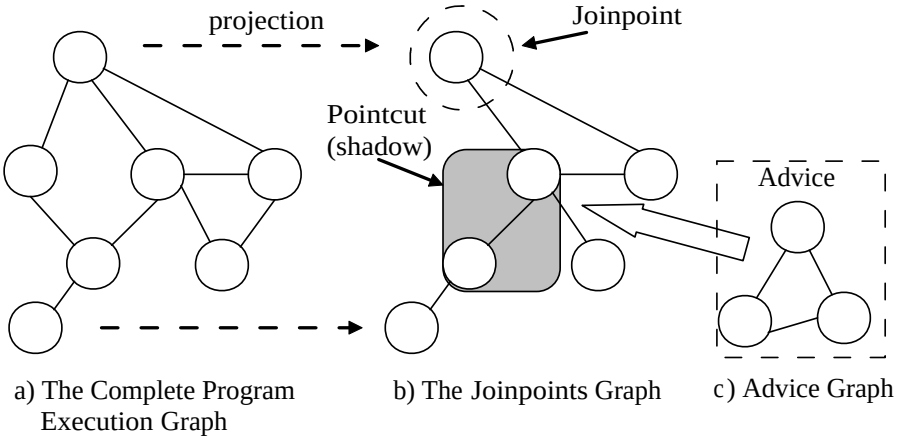


Figure 2.13: Illustration of Components found in the AOP Terminology

- A program is seen in AOP as an *execution graph* [MK03]. A node in this graph represents a possible state of the program and an edge represents a transition from one state to another. The node set of the execution graph of a program includes theoretically any state in

[34] AspectJ is Java dialect with AOP extensions. The AspectJ's predicate model has been influential to other AOP languages.

a program execution, with all possible paths to that state from the other states as shown in part (a) of Figure 2.13.

Building the complete execution graph for any arbitrary program and any possible input is unfeasible. The term *execution graph* is often used in AOP to mean some abstraction of the entire execution graph, consisting of some representation of the Abstract Syntax Tree (AST) and eventually augmented by information from additional static or dynamic analysis, e.g.: data flow information, control flow information, and other forms of execution history. Collecting the additional information, e.g., execution history, may require special run-time support[35].

- Accessible points of the execution graph are called *joinpoints* and are defined as *"any identifiable execution point in a system"* [Lad03]. As shown in part (b) of Figure 2.13, joinpoints are a subset of the execution graph accessible in a particular representation of it[36]. A joinpoint model describes the execution graph nodes based on the context where they are found, shown as edges inside the dashed circle in part (b) of Figure 2.13. That is, the execution nodes are identified not only by their own characteristics, e.g., a method signature, but also from the context where they exist, e.g., where the method is being called.

  As mentioned in section §2.3.2, the accessible points of a program graph depend on the technology used to describe it. Some systems, such as AspectJ, place further restrictions on accessible joinpoints and distinguish between all possible joinpoints (accessible for example thought AST) and *exposed* joinpoints [Lad03]. For example, AspectJ's joinpoint model does not expose loops. That is, loops are a possible, but non-exposed joinpoint.

- An execution graph is a state machine. To modify the execution graph, there should be a way to select sets of nodes of interest and modify, replace, or delete them. There are two ways how this can be done and both are explored in AOP:

  i. The execution graph is totally known[37]. That is, the entire state machine graph, or at least the wanted parts to change, are known. In this case, it can be spoken of *selection* of nodes and of explicit *search* over the execution graph. Depending on the execution graph representation, many transformations are possible in this total view. For example, often the AST is used as a representation of the total execution graph.

  ii. The execution graph is partially known at run-time. Transformations over the total view of the execution graph cannot be done in real-time. This could be a problem for those graph transformations that rely on information that is available only at

[35] An optimization used in AspectJ [KHH+01] is to simulate part of execution to collect some run-time data from the static AST to avoid the overhead of run-time support, for example, when implementing *cflow*.

[36] A projection in time and space of the execution graph.

[37] This is of course not possible all the time, especially at run-time.

run-time. In this case, the execution graph can be seen as an automata, and the transformation process can wait for patterns of transitions to appear, or *events*. Every time an event happens, a respond to it in real-time with predefined *event handlers (callbacks)* can be programmed.

The real-time processing of events[38] requires some form of run-time support which is often not acceptable, because of the additional overhead, or proprietary technology additions. Using partial evaluation [JGS93] (partial execution) techniques, many real-time (dynamic) events can be equally expressed statically, by using the information found in total static models, such as the AST, the way AspectJ does. The total view of the graph is preferable, as it removes the need to maintain the execution context. The real-time partial view requires maintaining the execution context explicitly[39].

AOP systems logically use the real-time event-based way for reacting to sets of nodes of interest. Practically, most events can be statically implemented. For this reason, some of the early AOP systems, such as AspectJ, speak about *virtual events* over the program control flow [Lad03]. Newer implementations, especially those of dynamic AOP, such as Prose [PGA02], react to actual events in the program execution and the distinction between virtual events and event-driven (triggered) systems is blurred.

The event patterns of interest, over the execution graph, are made declarative by using specialized predicates. A predicate model could support the composition of primitive predicates. Predicates can be made part of a general-purpose language, and are known in AOP as *pointcuts*. For example, pointcut predicates, such as *cflow*, are made part of the AspectJ syntax. The selection criteria in a pointcut is based on all characteristics of a joinpoint, this includes a node in the execution graph and its context.

- Every time an event of interest is matched by a pointcut predicate, some action of interest (event handler) can be executed. This is known as *advice* in AOP and is shown in part (c) of Figure 2.13. The execution graph regions matched by a predicate are also known as the *pointcut shadow* [MK02].

  The advice contains code to be executed for the matched nodes. The modifications that the advice code can do to the execution graph can include any graph rewriting [Men99] technique, depending on the model used to present the execution graph.

  The process of injecting the advice code into the pointcut shadows is known as *weaving*. Depending on the AOP system the weaving process can be either static or dynamic.

[38] That is, responding to the events as soon as they happen with the allowed timespan.

[39] This is analogous to the difference between DOM and SAX parser models for XML documents (http://www.w3.org/) [McL01]. A *Dynamic Object Model (DOM)* parser processes entirely an XML document and builds a total tree of its nodes, so the node tree can be explicitly searched and modified using XPath or XQuery. A *Simple API for XML (SAX)* parser, on the other hand, generates events (that can be processed as necessary) every time it encounters a node tag in a XML document. In the SAX parsers case, the context, where a node is found with regard to the other nodes, needs to be maintained explicitly when needed.

- The combination of pointcuts and advice is known as an *aspect*. Depending on the particular system, an aspect may contain also other elements. For example, AspectJ aspects contain compile-time directives for errors, introduction (invasive changes in the structural hierarchy of a program), and can declare new methods and fields.

2.4.2 AOP as a Generic Invasive Transformation Technique

The above discussion about AOP shows that AOP engines are generic transformation engines, that can be used to carry out a great range of invasive transformations. AOP techniques have been used to support product-line variability [MO04, LRZJ04]. The possible AOP transformation are limited only by the joinpoint model supported by a specific AOP engine implementation. The use of generic AOP engines, such as AspectJ, is preferable for program transformation because of:

- *Language integration*. AspectJ is tightly integrated with the Java language. This enables a great range of compiler-based static checking to find errors in the aspect code[40].
- *Declarativenes*. AspectJ offers a set of declarative context-enabled predicates (pointcuts) to select nodes of interest[41]. The pointcuts not only ease the implementation, they also build a common vocabulary to speak about node selection.
- *IDE support*. Statically checked generative tools with IDE[42] support simplify the implementation of program transformations. AspectJ is tightly integrated to Java, and well supported by development tools, such as, Eclipse AJDT [Ecl05].

These features make any generic transformation system, such as AspectJ, preferable, because the cost of developing custom transformation systems is often high and cannot be justified. There are also several liabilities:

- *Specialized transformation engines can explore better the domain characteristics.* Generic transformation engines, including the AOP ones, are not always the best option. Generic transformation engines could be used to implement transformations in those systems where it makes no sense to invest in a more specific transformation technology. However, invasive transformation frameworks specialized for a narrower purpose are better suited in the long term than any generic transformation engine. An example, where a specific technique is better suited than any generic technique, would be to add OO support to

[40] AspectJ is not the only generic generative framework that is statically checked.
[41] AspectJ users are not explicitly aware that aspects rely on the meta-model of a program to manipulate it. This is explicit in other systems with meta-programming capabilities.
[42] Integrated Development Environment.

ANSI C. Starting with a C `struct` construct, additional abstractions around it could be defined, using a combination of C code and generative transformations as in [Sch94a], to make the `struct` construct behave as a C++ class. Any invasive transformation tool that allows us to access the AST of a C program can be used for this purpose. However, the complexity of the solution would make any generic tool based implementation complex. Specialized generative techniques for this problem, as in [Sch94a], work better in ANSI C case. And it is even better, when the OO abstractions are made part of the language, as they are in C++.

- *Limitations in transformation capabilities of generic systems.* Another issue with generic transformation engines is their transformation limitations. The limitations are unfortunately not always clearly stated. For example, the joinpoint model of AspectJ cannot be used to enforce capital name conventions [TBG03]. These limitations exist on purpose in AspectJ. They make its programming model simpler and AspectJ was originally intended to make various modularization factorizations over the meta-model of a Java program easier. Supporting very detailed joinpoint models is possible, but would remove much of the declarativeness of the pointcut notations used.

- *Limited support for vertical transformations.* AOP engines, such as AspectJ are tightly connected to the meta-model of the language they target. They cannot support new keywords or new language constructs. This makes some generic AOP systems, such as AspectJ, to offer limited support for implementing arbitrary DSA constructs. Section §3.5.3 returns to this issue and explains in more detail the relation between AOP and DSA.

Chapter §4 develops a transformation technology specialized for interpreting DSA emulated with attributes. Having a special transformation technology for this domain, enables developing modular attribute-driven transformers that are difficult to structure and enforce with more generic transformation technologies. It makes sense to invest on an attribute-driven transformation technology, as the problem is relatively complex, very specific, and the attribute-driven transformation technology can be used to support open container frameworks to organize assets of product-lines for more than one domain.

2.5 Chapter Summary

More mobile device applications can be build and debugged in time, when product-lines are introduced. Iterative mobile application product-lines automatically reuse the common functionality of the mobile application families. Several variability mechanisms can be used to support mobile product-lines, e.g., OO libraries, frameworks, visual modeling with CASE tools, and domain-specific abstractions (DSA).

DSA support the architecture of a mobile product-line at the language level, blur the architectural distinction between visual CASE tools and programming language constructs, and allow for domain-specific optimizations. The cost of introducing DSA remains still very high, and low-cost implementation mechanisms need to be explored.

Software containers offer an convenient architectural pattern to organize product-line assets. Containers clearly separate the application specific functionality from the cross-cutting domain functionality. Containers can be used to transparently inject the domain services into a specific application. A mobile container is used to support product-lines for mobile device applications. The assets of a mobile product-line are organized as container services, having both a server and a mobile client part. DSA, combined with the container abstraction, can be used to create open container families for supporting iterative product-line development.

There are several invasive and non-invasive approaches to implement the dependency injection of services in containers. Non-invasive techniques are easier to implement, but offer less automation. Invasive techniques are better suited to support attribute-based DSA in mobile containers. Static invasive techniques, in the context of a container, can be used to bind the attribute-based DSA to the product-line services.

The need for several technologies was identified, that will be explored further in the following chapters:

- *Lightweight language extensions based on attributes.* Attribute enabled language technology can be used to emulate embedded DSA. Attributes support iterative product-line development with minimum start-up costs. Language technology that directly supports attribute-based transformations is needed (§3).
- *Attribute-driven transformation support.* Any generic invasive programming technology can be used to implement and support attribute-based abstractions. Specialized transformations could help to better modularize attribute-based transformers and make them declarative. Attribute-based transformer technology enables the reuse of transformation components and declarative composition of attribute-based transformers (§4).
- *Open container families.* Extensible containers are needed to organize the common mobile device application functionality. Mobile containers are specialized to address the peculiarities of mobile applications and to organize the mobile product-line assets, such as, the need for data adaptation (§5).

Chapter 3

Attribute Enabled Software Development

We don't usually consider a statement to be data at all, since it cannot be read, written, or manipulated.

R. A. Finkel, Advanced Programming Language Design, 1996

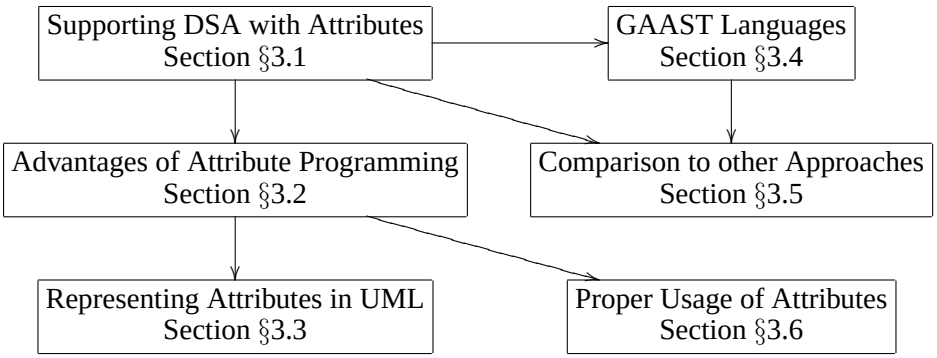
This chapter[1] explains the main ideas behind attribute-enabled language technologies[2] and their usage to sustain domain specific abstractions (DSA). Section §3.1 portrays how DSA can be supported with attributes, and how the domain variability can be modeled as nested attribute-families. The advantages of *attribute enabled programming* (AEP) are discussed in section §3.2, by analyzing several scenarios for mapping of Model-Driven Architecture (MDA) UML class diagram models onto source code artifacts.

Section §3.3 investigates several ways to support the attribute-based software design in UML class diagrams. The language technology behind AEP is discussed in section §3.4, stressing the importance that this technology be part of the core language, and supported by the lan-

[1] This chapter shares content with references [CM05a, CK05].

[2] The term *language technology* is used to mean the complete language system: the grammar, the binary representation, and the run-time system.

guage vendor. Related approaches are addressed in section §3.5. Attribute programming is compared with other ways to support EDSL constructs, namely extensible grammars / compilers and meta-programming systems. Finally, section §3.6 explores the proper usage of attributes to clarify the power and some of the pitfalls of the AEP.



3.1 Supporting DSA with Attributes

Based on the third alternative for implementing domain specific abstractions (DSA) explained in section §2.1.5, namely generic language extensions, this book presents technology that addresses the DSA drawbacks mentioned in section §2.1.5, based on *explicit attributes* [Hed97, BCVM02]. Explicit attributes are a lightweight language extension which reduces the grammar processing costs, and the accidental [FPB87] costs of the DSA implementation. Using a *attribute* (or *tag*) to denote an additional property about an existing entity is intuitive. When Knuth [Knu90] describes the idea of using attributes as tags that carry out semantics related to grammar productions, he notes that the idea has been around for some time. Since then, attribute grammars have evolved and matured and they are used in different ways to develop software [Paa95].

An example is the tag definition and usage in the Meta-Object Facility (MOF) [Met02] and the Unified Modeling Language (UML) [OMG03][3]. In MOF, tags derive from the class `ModelElement`, i.e., any element can be decorated with custom defined tags (Figure 3.1). Tags carry no domain-specific semantics for the MOF or UML itself. Their semantics are meaningful only to the modeler of the profile, who introduces the tag definitions. Tags serve as hints to model transformers and generators [Fra03], and enable the association of arbitrary semantics with model elements of interest, without having to change the meta-model of a given model. In terms of MOF, tags enable modeling in different vertical layers simultaneously. There is some functionality available to modify the meta-model M_{i+1} in the layer M_i, providing similar functionality in the layer M_i, as found directly in the upper meta-model layer M_{i+1}.

[3] UML can be described in terms of MOF.

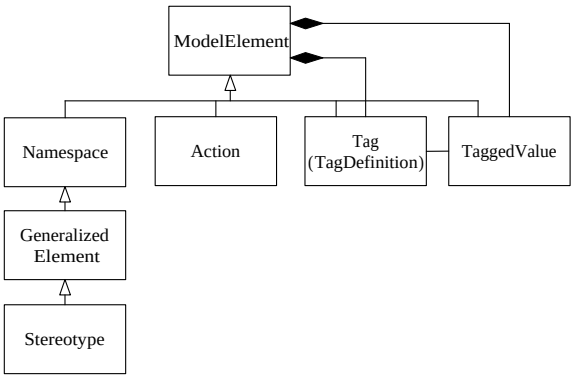


Figure 3.1: The *Tag* element in MOF / UML can be used with any *ModelElement*

This book focuses on the usage of *attributes* [Pro02], known also as *annotations* [JSR03], or *tags* [Met02], as a form of graph labeling [DJ90, Plu95, Men99] at the programming language level[4]. In some programming languages, attributes are *explicitly* present as part of the grammar rules, and enable modification of the semantics of the language constructs without changing the grammar [TS97]. The interest is in programming languages that enable a programmer to define any number of attributes to decorate selected elements, promoting an explicit attribute programming model [BCVM02]. Examples of languages that support the annotation of program elements with attributes are the .NET [Pro02] platform with its common language model, and Java 1.5, whose annotations support is described in JSR 175 [JSR03]. In other languages, where support for attributes is missing, e.g., J2ME MIDP [J2M02b], attributes can be emulated with special comments, as is the case with JavaDoc [Pol00] comments in Java 1.4 (§5).

Unlike other approaches, where a limited number of predefined attributes is used to replace custom keywords [TS97], explicit attributes can be introduced freely, in any number, in a language that supports them. They are used directly at the source level and hence the qualification *explicit*. This is different from other approaches, where attributes are used in the inner parts of a compiler to save intermediate processing results [vdBHKO02, WB00]. Attributes enable the decoration of program entities with custom notations, whose semantics are defined by the programmer. Attribute decorations are explicitly used by the developers and represent semantics that make use of an arbitrary context, unlike in an attribute grammar [Paa95, WMBK02], where attributes are used inside the parser to help with the evaluation of the grammar rules, having with a well-defined context propagation. Attributes can be parameterized (§3.1.2) and help to drive program transformations.

Attributes can be used to emulate DSA at the language level. In a language that sup-

[4]Different names are used for tags, such as attributes [Pro02] or annotators [JSR03].

ports attributes, e.g., .NET C# [Pro02], the same web service language extensions of Figure 2.4 (§2.1.5) can be coded by introducing two custom attributes, as illustrated in Figure 3.2. The `TravelAgent` class is decorated with a `[WebService]` attribute, whereas its public methods that constitute the web service interface contain a `[WebMethod]` attribute. Introducing new attributes is supported directly by the .NET compilers. There is no need to deal with any grammar modification issues, making it easier to extend a .NET-based language, such as C#, with domain-specific constructs.

```
1 [WebService]
2 class TravelAgent {
3   ...
4   [WebMethod]
5   public void GetHotels(){...}
6   ...
7 }
```

Figure 3.2: A Web Service Class with two Inter-depended Attributes

Unlike some other forms of EDSLs, such as SQLj [SQL03], which introduce islands of alien code into a host language, the methodology presented in this book is restricted to less expressive language extensions in the form of new parameterized attribute-based keywords, that fit in the overall design of the host language. While less expressive than full-fledged EDSLs, attribute-based language extensions are expressive enough to support declarative DSA for iterative product-line variability, as will be explained in section §3.5 and demonstrated in chapter §5. The attribute technology makes it possible to benefit from the declarative nature of DSA in order to preserve the domain abstractions, whereas at the same time, attributes keep the start-up costs of DSA extensions at a minimum. These features makes attribute-based DSA a variation mechanism of choice for iterative mobile product-lines.

3.1.1 Supporting Domain Variability with Attribute Families

There are several techniques to identify the core assets and to present the variability of the requirements for a domain [KE02]. One of the most widely used is Feature-Oriented Domain Analysis (FODA) [KCH$^+$90, CE00, BHST04]. FODA models variability as feature trees with required, optional, or alternative sub-trees. Starting with a feature diagram, it is possible to identify components of a system and produce design models of how the final system may look like [ZJF01]. Feature representation facilitates also the representation of the domain concerns with declarative programming constructs [vDK01]. Modeling of the domain features with attributes could follow the structure of the feature diagrams.

The idea is to model the top features of a domain, that would be handled by an attribute-based container, by using single attributes. For example, a single attribute can be used to repre-

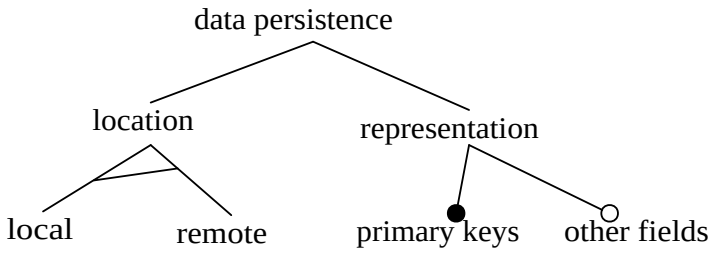


Figure 3.3: Feature Representation of Data Persistence

sent a top-level feature, such as the need to persist the data in a mobile application. A simplified (non-complete) feature model of data persistence is shown in Figure 3.3. The data may need to be stored locally in the device, or remotely on the server-side. The possible representation model of the data needs to be decided for each component, e.g., to identify the fields that will serve as primary keys. Similarly to a feature model, attributes can be grouped in tree-like name spaces, where each name space is used to model different parameters of a single product-line asset. Sub-attributes are added as necessary to follow the feature model, and obtain a tree-like view of the modeled domain concerns. Attribute name spaces will be called *attribute families*, and a C++ [Str97] namespace-like dot notation will be used to organize them. The parameterization of attribute families is done by using nested attributes, or attribute parameters.

For example, consider the attribute-based implementation example of a *GameScore* component in a mobile application[5], shown in Figure 3.4. The code in Figure 3.4 is a declarative specification of the functionality encoded in Figure 3.5[6]. The fields in Figure 3.4 are decorated with explicit attributes in the form of special `'[]'` comments. The following attribute families correspond to the different generic domain assets (§2.2.3) that have been used in the code of Figure 3.4:

- `[property]` - adds accessor / mutator methods for a field (lines 3, 10, 17, and 20).
- `[validate]` - adds validation code for fields that have mutator methods; *min, max* show the required range for an integer or the required length ranges for a string field (lines 2, 8, 9, 15, and 16).
- `[dp]` - adds data persistence methods to the component, and enables retrieving the data

[5]The example is based on the standard *GameScore* example which comes with J2ME MIDP 2.0 [J2M02b] documentation, explained in chapter §5. The examples of Figure 3.4 and Figure 3.5 will be reused in chapter §4 to explain attribute-driven transformations.

[6]The code in Figure 3.5 is one possible result of mapping the code in Figure 3.4 to a concrete implementation for a concrete mobile device.

```
1  [dp]
2  [validate]
3  [property]
4  public class GameScore {
5
6    [dp.pk]
7    [dp.pk.sort("asc")]
8    [validate.min(0)]
9    [validate.max(100)]
10   [property.accessor]
11   private int score;
12
13   [dp.pk]
14   [dp.pk.sort("asc")]
15   [validate.min(4)]
16   [validate.max(32)]
17   [property.both]
18   private String playerName;
19
20   [property.both]
21   private String Comment;
22 }
```

Figure 3.4: Input Code

```
1  public class GameScore {
2    private int score;
3    private String playerName;
4    private String Comment;
5
6    public getScore() { return score; }
7
8    public String setPlayerName(
9      string value)
10   { playerName = value; }
11
12   // ...
13   public byte[] toRecord()
14   {
15     ByteArrayOutputStream baos =
16         new ByteArrayOutputStream();
17     DataOutputStream outputStream =
18         new DataOutputStream(baos);
19     outputStream.writeInt(o.getScore());
20     outputStream.writeUTF(
21         o.getPlayerName());
22     outputStream.writeUTF(
23         o.getComment());
24     return baos.toByteArray();
25   }
26   // ...
27 }
```

Figure 3.5: Output Code

records sorted based on the primary keys (presented as an attribute sub-family `dp.pk`) (lines 1, 6, 7, 13, and 14).

The organization of attribute families is similar to organizing other language abstractions into name spaces and enables the reuse of attribute names with overloaded semantics, reducing the total required vocabulary. For example, all attributes related to the persistence of data in a device start with the `db` prefix (lines 1, 6, 7, 13, and 14 of Figure 3.4). Attribute parameters, e.g., `"asc"` in line 7 of Figure 3.4, are used to support the variability of the specific attributes inside an attribute family. Attribute families provide a convenient method to organize the domain functionality. All domain assets are organized as a tree of attribute families. Similar to language name spaces, attribute families can contain nested sub-families that group the variability of similar concepts, e.g., `db.pk`, groups the variability parameters related to the primary keys (lines 6, 7, 13, and 14 of Figure 3.4).

Based on the domain features, the architectural space of a product-line can be designed and its programming model can be decided. In [CE00], the domain features are divided in four classes: concrete, aspectual, abstract, and grouping. As explained in section (§2.2.3), the

container-based view of a product-line enables a uniform treatment of generic and specific assets and their features. The programming model can be represented by attribute families. Attributes enable a uniform representation of all the feature groups distinguished in [CE00]. For example, the concrete implementation of the data persistence feature of Figure 3.3 will be a *concrete* component as part of the product-line services. The instantiation of the data persistence feature is, however, *aspectual*. The specification of the primary keys needs to refer to a particular component implementation. Attributes connect the feature-based variability, implemented as part of the product-line services, with the fine-grained feature variability code written manually by the developers.

Ideally, attribute families enable the presentation of the domain features declaratively, without giving any clues about the implementation details of a given feature. In practice, the level of abstraction modeled by an attribute family is as good as the feature model used to create it. Attribute can also quickly reflect the evolution of a feature model. As explained in section (§3.4), attribute-enabled languages offer a lightweight mechanism to support the evolution of the declarative constructs in code.

3.1.2 Attribute Parameters

Attribute parameters are a convenient mechanism to specify the domain variability, modeled by using attributes. This section discusses how attribute parameters can be included formally in the overall attribute model. The discussion hitherto has assumed that attributes have the following EBNF [GJ90] form: `attributeName := (paramterName = parameterValue)*`. An attribute is identified by a name and by any (optional) parameters that it takes.

A distinction can be done between *structural* attribute parameters that are known at compile time (usually static constants), and *behavioral* parameters whose value can be determined (dynamically) only at run-time. When attributes are used for generation, they can contain only compile time evaluated parameters. Run-time evaluated parameters usually need some form of run-time support. Behavioral parameters are similar to introducing additional method / constructor arguments, and can be handled as such in the systems that need them.

When speaking about attributes, their structural parameters will be implied without making any special assumptions about them. Using parameters is only a convenience in using tags for annotations. While parameters help to express attribute variability more declaratively, they do not make annotations more powerful as summarized formally by the following theorem:

Theorem 3.1.1 *Tags with explicit structural parameters can be always replaced with tags without parameters in a given program.*

Proof: Let T be a tag and π its parameters vector. It should be proved that (i) there exists a discrete function F: (T, π) $\rightarrow$ T', that returns a new tag T' based on the tag T and its parameters

π and (ii) that F is a finite function, that is, it has a finite co-domain. Indeed:
(i) Let F: (T, π) $\rightarrow$ T' be a function constructed by expressing all the parameters $\{\pi\}$ as strings that are joined together with some string separator (e.g: '$'), and appended to the tag T, for example T' = T$$\pi$. The function F returns a unique tag for any tuple (T, π). If '$' is escaped inside T and π, then F has also an inverse function F^{-1}: T' $\rightarrow$ (T, π).
(ii) The function F is a total function over $\{\pi\}$. If $\{\pi\}$ is an infinite set then so is F. But the number of structural elements in a given program is finite. Thus, even though π could be infinite, only a finite set of its values are used in a given program. So, F can be replaced with a finite set of partial functions over π (QED).

3.1.3 Connecting Attribute DSA with Product-Line Assets

Representing DSA with attributes in code removes the costs related to the grammar modification. Attribute-based DSA should, however, be interpreted and properly connected to product-line services. For example, the code in Figure 3.6 shows the attribute-decorated `GameScore` class of Figure 3.4. The used attributes state that the `GameScore` objects should be made persistent in the memory of a mobile device. The persistence requirement is expressed declaratively as a set of attributes, e.g., `dp`[7] in the `GameScore` class definition.

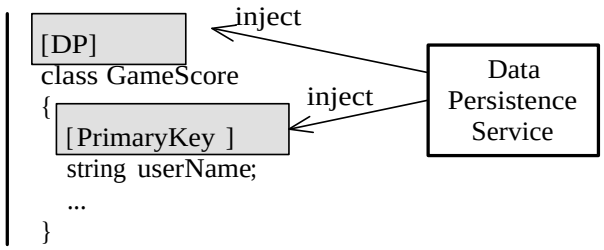


Figure 3.6: Connecting Attributes with Services

The declarative attribute-based DSA notation needs to be connected to the actual data persistence code (service). The connection can be implemented in a variety of ways, for example by enhancing the compiler to detect and process the attributes, or by using some other way to organize services, such as, as component libraries or templates, and then bind them to the attribute-decorated code as required. Given that the focus is on easy to implement and to maintain DSA mechanisms, extending the compiler is not an option.

Section §2.2 already discussed how product-line assets can be organized as services that are transparently managed by a software container abstraction. Chapter §4 develops a technology to implement modular attribute-based transformers, which helps to develop low-cost

[7] Data Persistence Object.

attribute-based DSA transformations and connect them to the product-line services. Attributes can be interpreted before, or after compilation, or at runtime. Attribute interpretation requires to somehow be able to access and eventually modify / transform some AST-like source- or binary-level representation of the annotated program. For instance, tags used in MOF / UML can be interpreted when the model is transformed to another more detailed model. JavaDoc attributes used in Java 1.4 can be interpreted when the source code is processed. .NET attributes, or Java 1.5 annotations are saved as part of the binary meta-data. This enables the manipulation of binaries after compilation, or the run-time interpretation of tags by using the Reflection API. The run-time interpretation requires that the original annotated program is structured in such a way, that it can be compiled without interpreting the tag semantics. For example, in .NET a method body cannot use a variable that will be introduced by an attribute interpretation, because such a method cannot be compiled.

This chapter concentrates on language technology for supporting and facilitating the construction and maintenance of attribute-based DSA transformers. Before doing so, in order to complete the attribute-based software development (AESD) discussion, the advantages of AEP will be summarized, and ways to integrate attributes in the early phases of software development will be considered.

3.2 Advantages of Attribute Programming

This section compares the benefits of having access to explicit attribute support at the language level, with other techniques to model domain abstractions in source code, namely marking interfaces and pseudo-syntax marking. A typical model mapping scenario from *Model-Driven Architecture* (MDA) [HS01, Fra03] is used. The goal of MDA is to increase the level of abstraction of software development. MDA enables software developers to specify ”what” a software solution should provide, rather than ”how” to realize the desired solution in terms of the technicalities of a particular implementation platform. The ”what” is specified in a so-called *Platform Independent Model* (PIM). Based on the chosen technology, there are different operations that can be used to realize a PIM, resulting in different *platform specific models* (PSMs) of the same PIM. A PSM can be a ready-to-run implementation, or it may act as a lower-level PIM for further refinement to a new PSM that can be directly implemented.

Obviously, it is desirable to have an abstract PIM and to automate its translation to a given PSM implementation. A fully automatic transformation of any abstract model is not possible. Additional PSM specific directives need to be provided by applying marks from a specific profile[8] to PIM elements. This implies a commitment to some kind of specific technology for solving the problem. The profile denotes the domain-specific notations, by using specialized forms of marking, e.g., tagged values and stereotypes. Marking represents concepts of the PSM

[8] UML profiles modify the UML M2 model, i.e., they bring extensions to the M2 level.

in the PIM and indicates how the PIM elements are to be transformed [MDA03].

For illustration, Figure 3.7 shows a simple web service with two methods, (i) to log in and authenticate a client (user) (`Login`), and (ii) to enable the client to access some user specific data (`AccessUserData`). The simple web service class, named `WebService1`, is modeled by the means of UML profiles. Three stereotypes ≪`webservice`≫, ≪`namespace`≫ and ≪`uniqueid`≫ are defined. The last two can also be represented as tagged values connected to the ≪`webservice`≫ stereotype. In addition, two free tagged values, `enableSession` of type `boolean` and `transactionOption` of type `enumeration`, are used to decorate the methods of this web service.

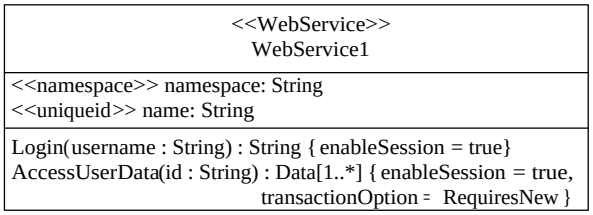


Figure 3.7: Modeling using UML Profiles

Compared to a more abstract counterpart that does not contain any of the above tags, the model of Figure 3.7 is technology dependent. Several technology commitments are made by using the profile. First, it is decided to expose the component as a web service. Second, the technical concerns needed by the service methods are explicitly enumerated, e.g., session and transaction management. However, at this point there is not yet any commitment made on how the session and transaction management will be implemented. The example only states the need for such technical services by means of the specialized UML profile, but has not yet committed to a particular web service platform.

In the MDA terms, the model in Figure 3.7 is still a PIM for a more specific PSM. Given such a marked PIM, some transformer T_1 is applied to produce a lower-level PSM, as shown schematically in Figure 3.8. In MDA the commitment about the technology is done in stages rather than at once. The rationale behind this stratified commitment [AK01] is that a model of a higher-level stage can be mapped to more than one possible model of a lower stage. A staged commitment makes it possible to exchange the lower-level models, while preserving the investment on the higher-level models.

Assuming that the target PSM is expressed in a programming language, the transformer T_1 knows (a) how to map marks to corresponding language constructs, and (b) how to map types used in the model, to types of the target programming language, e.g., a *String* in the modeling language may map to a character array in the target programming language. Type mapping is usually easy to handle automatically and will be not addressed any further. By selecting a given

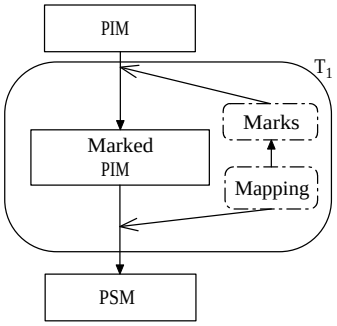


Figure 3.8: MDA PIM-to-PSM Transformations

language, a further commitment is made on what the final software will be like. Selecting the target language says still nothing about how issues, such as sessions and transactions, will be implemented by tag interpretation (eventually in a later stage). Concerning the tag mapping, in the case when the target PSM (Figure 3.8) is expressed in a concrete programming language, it could be distinguished between:

- mapping the tags to language constructs, and
- interpreting the language constructs to insert the specific concern's functionality.

A specific transformer may combine all three stages (mapping types, mapping marks to language constructs, and interpreting the latter) into a single pass. For instance, in addition to type and tag mapping, $\mathtt{T}_1$ (Figure 3.8) may also interpret the tags. In this case, $\mathtt{T}_1$ commits to concrete session and transaction implementations and produces an executable PSM.

However, it makes sense to separate tag mapping from tag interpretation, when the transformation of a PIM to an executable PSM is not fully automatic. This is the case when modeling is used only for defining the high-level architecture of an application. For example, in an EJB [MH00] application, it is preferred to model beans and their interactions by means of UML constructs. It is easier to write and maintain complex business functionality directly in Java rather than to model `for` loops and similar constructs using the UML action language [OMG03][9]. In the web service example, issues, such as sessions and transactions, will be handled automatically. Programmers may still fill-in manually the functionality for logging and retrieving data by implementing the methods `Login` and `AccessUserData`.

As the focus is on programming language support for tag mapping and interpretation, the interest will be in lightweight mappings of tags to language constructs that do not require

[9]The UML action language might be well suited to model embedded systems, where full automation of the transformation would make sense.

domain-specific additions to the target language. Such mappings are preferable because of lower costs for mapping arbitrary custom profile elements to a general-purpose language. The remaining of this section compares three widely used approaches for handling the mapping of tags to programming language constructs and for the interpretation of language constructs, namely: *marking interfaces*, *pseudo-syntactic marking* and *attribute mapping*[10]. The comparison is done along the following dimensions:

1. *Preservation of the PIM.* Preserving the architecture of the marked PIM in the source PSM is important, because it helps (a) to reverse engineer the source code PSM and (b) to understand the original PIM architecture by looking at source code alone. Figure 3.9 shows an equivalent textual model of the web service of Figure 3.7 in an extended[11] HUTN[12] notation [Hum02]. The OMG HUTN standard is aimed at defining textual equivalents of MOF / UML diagrams which can automatically be generated. The tags of the web service example are modeled as extended adjectives in terms of HUTN. It is desirable that the source code PSM preserves the PIM structure to the same degree as the HUTN representation.

```
1 webservice "WebService1"
2 {
3   namespace namespace : "www.st.tu-darmstadt.de"
4   uniquieid name : "Simple Service"
5
6   enableSession Login(username : String) : String
7   enableSession transactionOption.RequiresNew
8     AccessUserData(id : String) : Data[1..*]
9 }
```

Figure 3.9: Extended HUTN Model

2. *Complexity of the Programming Model.* As already mentioned, parts of the code need often to be filled-in manually by the programmer in the generated PSM code. The structure of the PSM produced by tag mapping directly affects how the programmer interacts with such code. It is preferable to keep the programming model simple.

3. *Interpretation of Mappings.* Interpretation is the next step after marks are mapped into language entities. Different kinds of mappings result in different techniques of interpretation. The interest will be in how easy it is to interpret language constructs resulting from tag mapping by considering the native support that the language technology offers for this purpose.

[10] A given tool may use any combination of these approaches.

[11] The introduced extensions address modeling profile elements. The current version of HUTN specification does not address any extension mechanisms for HUTN in order to keep the language simple [Ste99].

[12] Human-Usable Textual Notation.

4. *Extensibility of the Profile.* Extending a profile is often a requirement rather than an option. It is preferable to have means which facilitate the introduction of custom extensions to profiles. To illustrate the discussion, suppose that a new traceability tag named `log` is added to the custom web service profile of Figure 3.7. This tag, when used with a method, will generate code that logs all method invocations. In the discussion that follows, the tag will be added to the `Login` method. Logging enables to register which users used the service, at what time, and which users failed to authenticate.

3.2.1 Mapping Marked PIMs to Marking Interfaces

To implement mapping of the PIM of Figure 3.7 in Java 1.4, interfaces are often used as a means to emulate marking code elements at the language level [NV02, Fra03]. When the PIM of Figure 3.7 needs to be implemented in Java 1.4, it could map to the Java model consisting of the classes and interfaces of Figure 3.10.

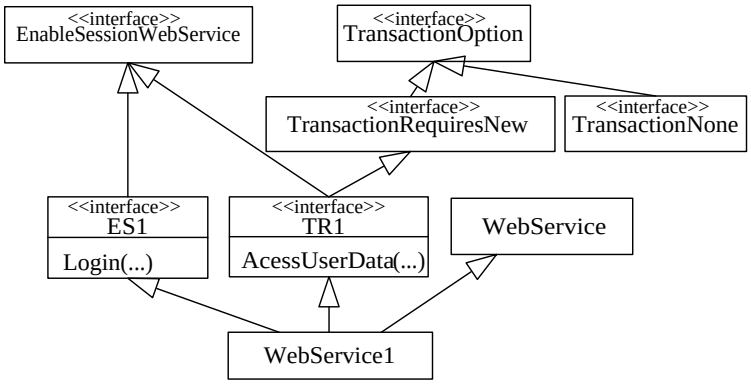


Figure 3.10: Mapping by Means of Marking Interfaces

For each tag and stereotype in Figure 3.7, a marking interface is introduced in Figure 3.10. Multiple-value attributes are modeled as specialized interfaces that derive from other base marking interfaces. For example, Figure 3.10 assumes that the multi-value tag `transactionOption` can only have two possible values, `RequiresNew` and `None`, which are modeled as children interfaces of the `TransactionOption`. The stereotypes can also be modeled as marking interfaces, or as specialized (prototype) classes, e.g., `WebService` in Figure 3.10. The mappings for the ≪`namespace`≫ and ≪`uniqueid`≫ stereotypes are not shown in Figure 3.10 and it is assumed that they are used in the implementation of `WebService`.

To emulate the tags of a given PIM method in a given class, a specialized interface for that method needs to be created. The specialized method interface derives from the basic inter-

faces that model the respective tags. For example, the interface `ES1` inherits from `EnableSessionWebService` to make explicit that the method `Login` is marked by the tag emulated by the `EnableSessionWebService` interface. In the same way, the interface `TR1` containing the method `AccessUserData` is derived from `EnableSessionWebService` and `TransactionRequiresNew` to reflect the fact that this method is marked by both `enableSession` and `transactionOption=RequiresNew` in Figure 3.7. This way, the attributes of a method can be found by looking at the interface it belongs to.

Discussion:

1. *Preservation of the PIM* - The mapping of Figure 3.10 does not preserve the modular structure of the PIM in Figure 3.7. From the model in Figure 3.10, it is hard to guess the clear and concise structure and semantics expressed by the original PIM. The corresponding PSM code contains an explosive number of marking interfaces. This makes the PSM model more difficult to understand and hinders reverse engineering to the original PIM. The corresponding Java code of the UML model of Figure 3.10 will also be much more verbose compared with the textual HUTN representation of Figure 3.9.

2. *Complexity of the Programming Model* - Even though the example is extremely simple and several simplifying assumptions were done, the resulting PSM (Figure 3.10) gets complex and verbose. Cross-interfaces need to be introduced, which inherit from base ones. The resulting PSM already mixes the business logic model with the model for implementing technical concerns. The technical concerns crosscut the modular structure of the business logic in Figure 3.10 and dictate the inheritance structure of the resulting program. The developer responsible for implementing the PSM in Figure 3.10 cannot ”escape” some issues related to the implementation of the technical concerns. The developer has to know that interfaces, such as `ES1`, exist and will be handled by a suitable concern automation environment.

3. *Interpretation of Mappings* - Marking interfaces are not easy to interpret. Extracting the tags of a given method requires finding the interface where the method is declared, and retrieving the interfaces from which this interface inherits. When the interpretation is done at the source-code level, the full inheritance hierarchy must be resolved. Using compiled pseudo-binaries that contain meta-data, e.g., Java bytecode along with the Reflection API, makes it easier to resolve marking. Reflection relies on the existing virtual machine mechanisms to resolve the inheritance hierarchy. In order to extract knowledge about the marking interfaces hierarchy, the transformer that performs the mapping of marks must know the meta-model abstractions of the marked PIM.

   The overhead of tag extraction in terms of both complexity and performance of the interpretation logic cannot be avoided because the mapping of model tags to programming language constructs does not preserve tags as first-class values. First, as indicated above, the transformer that maps tags to language constructs is complex and needs to know the

tag meta-model of the marked PIM. Second, the tag interpretation transformer needs to basically ”undo” this work in order to extract tags from the marking interfaces resulting in a double overhead.

4. *Extensibility* - To add a new `log` tag to the method `Login`, a new additional interface `Log` need to be created in the language representation. In addition, the `ES1` interface needs to be modified so that the web service class inherits from the new interface, in order to make explicit which new attributes the `Login` method has. If `log` were to be added to the other methods, the other interfaces need to be modified in a similar way. This would make the PSM model even more complex. Hence, the marking interfaces approach does not scale well.

3.2.2 Mapping Marked PIMs to Pseudo-Syntactic Elements

One way to decrease the number of the resulting marking interfaces is to rely on coding conventions, such as those used in JavaBeans [Har97], J2EE EJB [Jav02a], or COMPOST [Aßm03]. Pseudo-syntactic marking uses string prefixes / suffixes to decorate the names of the marked elements according to some accepted convention based on the marked PIM. This approach will be called *pseudo-syntactic marking*, because it adds new syntax to a language without really adding new keywords.

Some systems, e.g., COMPOST [Aßm03], rely on pseudo-syntactic marking for expressing class and member annotations. Other systems, e.g., JavaBeans [Har97] and J2EE EJB [Jav02a], use a mixture of marking interfaces, required coding conventions, and annotations defined in so-called bean info objects (in the XML deployment descriptor). J2EE EJB implements class-level annotations by means of predefined classes / interfaces from which the annotated classes inherit. For methods and attributes, JavaBeans and J2EE EJB use pseudo-syntactic marking.

```
1 WebService1 implements WebService {
2    String sessionLogin(String username) { ... }
3    Data[] session_transactionAccessUserName(String id) {
4      Transaction t =
5        context.getTransactionFactory().NewTransaction();
6            ...
7    }
8 }
```

Figure 3.11: Using Pseudo-Syntactic Marking

For illustration, Figure 3.11 shows how the web service of Figure 3.7 could be mapped with pseudo-syntactic marking. The ≪`WebService`≫ stereotype is mapped to the prede-

fined class `WebService`, from which `WebService1` inherits[13]. The `session` and `session_transaction` are used as required method name prefixes for methods `Login` and `AccessUserData` respectively. For analogy, in the EJB [Jav02a] model there are predefined types, such as `SessionBean` or `EntityBean`, as well as coding conventions, such as `ejbPassivate` or `ejbActivate`.

Discussion:

1. *Preservation of the PIM* - The original PIM is preserved better in this approach compared to the approach based on marking interfaces. However, pseudo-syntactic marking does not fully preserve the original PIM structure. Coding conventions and implementation restrictions imposed by a component model pollute the PSM, such that the details of original PIM get blurred.

2. *Complexity of the Programming Model* - Coding conventions for method prefixing reduce the exponential number of the emerging cross-derived marking interfaces. Compared to a PSM expressed by marking interfaces only, a PSM expressed by pseudo-syntactic marking hides some of the details for implementing technical concerns. Pseudo-syntactic marking abstracts over the way the technical details are concretely realized by a certain framework model. The syntactic marks still need to be processed. Pseudo-syntactic marking also introduces new complexity at the programming model [POM03]. The programmer has to be aware of the coding conventions and implementation restrictions encoded in the framework in use, which cannot be directly enforced by the compiler.

3. *Interpretation of Mappings* - Pseudo-syntactic marking is more difficult to parse than marking interfaces because the transformer must use string operations on the code element names in order to extract tags. This can also cause performance overhead when done repetitively at run-time, due to the need for introspection. For example, a bean implementation in EJB needs to be introspected after it is compiled so that the container can generate and add glue code [Hal02], e.g., to handle its passivation methods based on their name prefixes. As with marking interfaces, the overhead paid for tag extraction is introduced because tags were not preserved as first-class entities in the first place during mapping model tags to language constructs. No direct support is offered by the language technology for pseudo-syntactic mappings.

4. *Extensibility* - Again, consider adding the new tag `log`. The session concern can be represented as a new special prefix with pseudo-syntactic mapping, and the name of the method `Login` will be `log_sessionLogin`. Such a schema is more fragile than marking interfaces because the new name could easily contradict with existing names, and may require a more careful code overview to avoid errors.

[13] A pure pseudo-syntactic marking approach would have used a `webservice_` prefix.

3.2.3 Mapping Marked PIMs to Attribute-Enabled Languages

As discussed in section §3.2, the transformation of a marked PIM to a source code PSM can be staged into (a) mapping marking elements into language elements and (b) interpretation of such language elements. An attribute-enabled language helps with both (a) mapping marking elements to language constructs and (b) building transformers that do the interpretation. To deal with the issues present in other language representations, e.g., the preservation of PIM and the complexity of the programming model, a new element can be added to the MDA model of Figure 3.8, represented by the gray box in Figure 3.12.

The transformer T_2 performs the mapping of model tags to language level tags. The T_2 mapping is straightforward, in the sense that tags are basically preserved and only written in a different syntax, since tags are first-class values in an attribute-enabled language. For illustration, Figure 3.13 shows a possible mapping of the extended HUTN notation for the web service of Figure 3.9 in .NET C# language[14]. .NET C# is considered an attribute-enabled language due to its explicit support for tagging in the form of attributes. The attribute-driven transformer in Figure 3.12 is responsible for implementing the semantics of the tags. Compared to the interpretation of tags in the approaches discussed in the previous sections, the attribute-driven transformer does not need to do any tag extraction as attributes are full status elements. Implementing attribute-driven transformers is natively supported by attribute-enabled languages (§3.4).

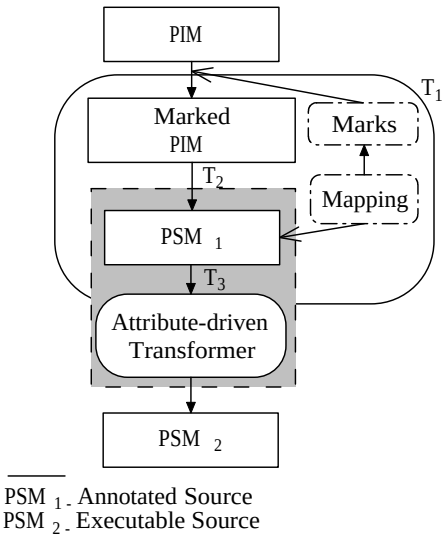


Figure 3.12: MDA Attribute-enabled PIM-to-PSM Transformations

[14]The web service example was deliberately chosen to be a simplified version of the web service modeling semantics provided in the `System.Web.WebServices` .NET namespace.

```
1  [WebService(namespace="www.st.tu-darmstadt.de",
2    name="Simple Service")]
3  class WebService1 {
4    [WebServiceMethod(enableSession)]
5    string Login(string userName)
6    {// TODO: add implementation code here}
7
8    [WebServiceMethod(enableSession,
9      transactionOption = TransactionOptions.RequiresNew)]
10   ArrayList AcessUserData(string id)
11   {// TODO: add implementation code here}
12 }
```

Figure 3.13: Mapping to .NET C#

Discussion:

1. *Preservation of the PIM* - Converting the HUTN representation of Figure 3.9 to C# is straightforward. Apart from type and syntax mapping these models are actually equivalent. The transformer for the attribute mapping does not need to access the PIM meta-model. The transformer T_2 can work at the M1 level. This is different from the case of transformers responsible for tag mapping in the marking interfaces approach. Marking interface transformers need details about the meta-model to do the transformation.

   With the attribute representation, the model structure is preserved in source code, hence facilitating reverse engineering to the original PIM's architecture. The annotated source can sustain the full design architecture better than pseudo-syntactic marking without having to process method names, and without having to invent many method prefixes or suffixes.

2. *Complexity of the Programming Model* - The use of annotations simplifies the programming model because marks are still explicit. Having attribute-driven transformation support in the language framework provides the means for processing the language-level PIM. In other words, the part of the transformation concerned with introducing the details of "how" to realize the specified "what" is pushed entirely down to the language level.

   The series of commitments in the MDA-based development could start by choosing the target attribute-enabled language. In an attribute-enabled language decorations based on attributes are used directly in code. This would be an alternative design to a graphical UML profile modeling step. Making programs look like designs improves the programming model, given that it decreases the intellectual gap to the domain concepts.

3. *Interpretation of Mappings* - The interpretation process in the marking interfaces and pseudo-syntactic marking approaches may require third-party tools which can parse and modify the input. An attribute-enabled language serves as a unified framework that does not require any additional third-party tools for parsing the input AST (§3.4).

Having the attribute processing be integral part of the language also simplifies the development of the transformers. Instead of relying on external systems to introduce custom parsing extensions to the language, the language is designed from the beginning to support such kind of transformations. The developer can focus on specific needs of the transformation, e.g., how to integrate transaction management, and not on the *technicalities of the transformation itself*, e.g., on how to parse, access and modify the introduced linguistic abstractions for modeling transaction processing.

Shifting the transformation at the language level helps to achieve tool unification. The target development language is also the transformation tool. The transformation model supported by an attribute-enabled language is the only framework a developer must learn in order to deal with the transformation issues.

4. *Extensibility* - Attribute-based mappings are modular because they directly preserve the PIM architecture and are easy to customize and extend. New issues can be dealt with at any time, by defining appropriate annotations and by introducing corresponding processors to the language framework. The `log` tag example maps directly to a new attribute in this representation, and the mapping does not conflict with any of the other existing attributes. Nevertheless, still care is needed about the order of the transformation during the interpretation (§4).

This section concludes with a short discussion of some of the limitations of an attribute-enabled model-driven development (MDD):

- Only the mapping of UML class diagrams with specialized profiles is directly supported by attribute-enabled languages. UML class diagrams map easily to an OO language. The attribute-enabled MDA process was illustrated by the means of the general-purpose object-oriented language C# [Pro02], whose meta-model maps directly to the class-based web-service model. The transformation process may not be as easy when the UML model elements cannot be directly mapped to the target OO language. Other UML constructs cannot be mapped directly to the source code of a OO language. That is, attribute-enabled languages do not address the complete UML-based modeling of MDA.
- The integrated attribute processing greatly facilitates, but does not automate the implementation of the transformers. While it enables the transformer programmer to concentrate on the semantics of the technical concerns to be integrated rather than be concerned with issues, such as tag extraction, the technical concerns semantics and interactions among transformers still need to be taken care of by the programmer.
- Only the PIM structure is preserved in the language level PSM. Other more fine grained models of the method internals (e.g., using UML actions) will be lost, so only structural reverse engineering is possible.
- The target language must have support for attribute-driven transformations.

3.3 Representing Explicit Attributes in UML

Attributes play an important role in representing custom design-related meta-data [NV02, KM05, Lad05, Aßm03], and can be used to expose joinpoints [KLM+97] (§2.4). These usages of attributes require an appropriate representation of attributes at the design time and therefore in UML [OMG03] diagrams. Several UML extensions using stereotype-like notations have been proposed to deal with specific joinpoints in OO class diagrams [Don02, BGJ99, HKC05], but none of them deals directly with explicit attributes.

While attributes are similar to MDA MOF [Met02] tags and UML [OMG03] tagged values or stereotypes, the effect of using attributes explicitly in a programming language, e.g., with Java 1.5 [JSR03] annotations and .NET [C#02] attributes, opens new perspectives to design and program software. UML tagged values and stereotypes model only a subset of design possibilities that can be covered with explicit attributes. This section elaborates on the issue of representing attributes in UML, comparing different possible alternatives that can be used to model different design scenarios. The interest is to find convenient ways to represent explicit attributes in the UML class diagrams.

There are several issues with regard to representing attributes in UML. The first issue comes the used the terminology. In UML (and MOF) the term *attribute* is used to denote class fields: in accordance with OO terminology, a class contains methods and *attributes*. The term is also used with a generic meaning in the UML documentation, synonymously with the term *property*. The term *annotation* does not have any strict meaning in UML. It is used in the UML documentation mainly to denote text notes, i.e., comments. The best fit for a corresponding term for .NET *attributes* and Java *annotations* in UML (and MOF) are *tagged values*. Tagged values are one of the three extensibility mechanisms available in UML (the two others being *constraint* and *stereotype*).

> ☞ *A tagged value is a keyword-value pair that may be attached to any kind of model element (including diagram elements as well as semantic model elements). The keyword is called a tag. Each tag represents a particular kind of property applicable to one or many kinds of model elements. Both the tag and the value are encoded as strings. Tagged values are an extensibility mechanism of UML permitting arbitrary information to be attached to models.* (UML 1.5 Section 3.17.1 [OMG03])

The UML specification [OMG03] defines in section 3.17, a standard notation for representing tagged values and other custom properties in class diagrams (which describe the static structure of the system). Properties of an element are written after all other data for that element have been specified.

> ☞ *A property (either a meta-model attribute or a tagged value) is displayed as a comma delimited sequence of property specifications all inside a pair of braces ({ }). A property*

> *specification has the form name = value where name is the name of a property (metamodel attribute or arbitrary tag) and value is an arbitrary string that denotes its value. If the type of the property is Boolean, then the default value is true if the value is omitted.* (UML 1.5 Section 3.17.2 [OMG03])

There are, however, several problems with the property notation for modeling .NET attributes and Java annotations:

- Tagged values can only represent a single *(key, value)* pair. .NET attributes and Java annotations can take more than one *(key, value)* parameter. For example, an `[Author]` attribute may require name and department parameters (e.g., `[Author(Name=Vasian,-Department=TUD)]`). The parameters can also be named or positional. Coding this information as a special escaped string, in the value part of a tag, makes it error-prone and difficult to parse.
- In .NET, an attribute is a class, whereas in Java it is a special form of an interface. That is, a .NET attribute or a Java annotation exists as a separate class diagram element in a UML design. By representing an explicit attribute as a tagged value, it is impossible to distinguish between a tagged value and an attribute by looking at the notation only. A special notation can be used, e.g., an '@' prefix, for those keys that denote attribute names. However, this convention is not supported by default in the UML standard.
- .NET attributes and Java annotations can take complex type values as parameters, such as arrays of constants, or other attributes. The UML tagged value notation can be become overloaded with all this information.
- A .NET attribute and a Java annotation can be used to decorate not only a class but also inner class elements, e.g., fields, methods, method parameters, and the return type. The UML property notation can be used in all these cases, but the diagrams may become overloaded with information.
- .NET attributes and Java annotations are used here to represent domain-specific abstractions. In this context, attributes serve as a kind of parametrized stereotype, and have the same weight in the design, as a stereotype. The property notation, which is specified after all the other information for an element, does not give any visible clue about the relative weight of the attributes in the design.

There is more than one possible alternative for presenting attributes in UML class diagrams[15], with tagged values being the first candidate. The remainder of this section discusses

[15]The issue of presenting attribute definitions in UML will not be addressed, given that an attribute is similar to a stereotype. As attributes are first-class entities in Java and .NET, attribute definitions can be easily mapped to definitions of a specific ≪attribute≫ stereotype notation, used to decorate attribute classes in .NET and annotations interfaces in Java.

several alternative UML presentations of explicit attributes, and presents criteria for analyzing the benefits and drawbacks of each notation accordingly.

3.3.1 UML Alternatives for Explicit Attributes

Several UML-based representations could be used to include explicit attributes in class diagrams. While all these notations use UML standard mechanisms, they all extend the UML notation in one or more ways. The intention is to enumerate the most useful possibilities, and analyze the benefits and drawbacks of them. The different UML alternatives are compared based on the following criteria:

Standard Compatibility: How good does the notation relate to the overall UML standard. Notations that are as near as possible to the UML standard are preferred. Related to standard compatibility is *tool support* (integration in existing tools): Existing UML tools [Jec04] can easier support minor extensions that fit into the overall UML designs, rather than major changes.

Visibility: How good does the notation emphasize the importance of attributes in a UML design. Notations that emphasize the attribute semantics are preferred. Explicit attributes are often an important part of the design, and their relative weight should be properly expressed in the UML class diagrams.

Clarity: How clear or verbose the notation is. The less verbose notations are preferred. Verbose notations can be difficult to manage when they are used to decorate internal class elements, e.g., method parameters. Verbose notations could also result in cluttering of the UML diagram with too many elements.

Usability: How easy it is to use or reuse a given notation, e.g., being able to have a single definition of an attribute in a diagram and then reuse it via associations. Reuse is preferred, as it means less maintenance efforts. Related to reuse is *ease of use*, that is how convenient a notation is to use, in order to model an explicit attribute.

Representation Structure: The preferred notations sustain a more structured attribute representation, compared to those that simply represent attributes as strings. Structured notations are less error prone and easier to manipulate automatically.

The main possible alternatives for presentation of explicit attributes in the UML class diagrams are discussed next. The example of Figure 3.14, modified from [Lad05], will be used to illustrate the various alternatives. The example shows how a bank account class can be modeled by utilizing several Java 1.5 annotations. There are two class level attributes: `[Author]` and

```
1  @Author(
2      name="Vasian Cepa",
3      name="Sven Kloppenburg")
4  @DAO
5  public class Account {
6
7      @PrimaryKey
8      private string accountID;
9
10     ...
11
12     @Transactional(kind=Required)
13     public void credit(float amount)  { ... }
14
15     @Transactional(kind=Required)
16     public void debit(float amount)  { ... }
17
18     public float getBalance() { ... }
19
20     ...
21 }
```

Figure 3.14: Attribute Annotation Example

`[DAO]`[16], that denote the authors of the class, respectively, that the class needs to be processed in order to enable data persistence. Inside the `Account` class, there is a string field `accountID`, which is decorated with a `[PrimaryKey]` attribute as part of the data persistence functionality. This field will be used to identify the `Account` class records, when they are persisted in the database. Several of the class methods are decorated with a `[Transactional]` attribute, to denote that they must always be called as part of a transaction. Of course, not all code entities have attribute annotations, as illustrated by the `getBalance()` method. The UML examples below use a *tilde* ('∼') sign to separate multiples values in a UML property string value notation as needed, and an *at* ('@') sign to decorate attribute name strings. Possible alternatives are separated with *semicolons* (';').

UML properties, tagged values: This is the first UML representation possibility that comes to mind for attributes. The details and problems of this case were discussed in section §3.3 to motivate the need to explore other notations. In order to represent the explicit attributes, the property notation needs to be extended (a) with a special notation, e.g., '@', used before the attribute names, and (b) to enable the string values to have an inner structure, in order to represent the key–value pairs of the attribute arguments. The extended notation is illustrated in Figure 3.15.

The modification of the UML standard is minimal, and could be easily supported by any

[16] Data Access Object.

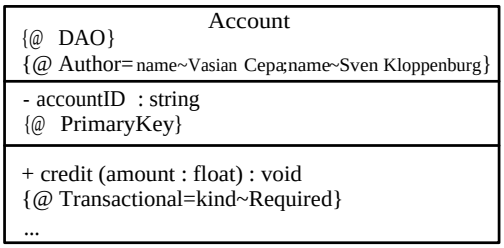


Figure 3.15: Attributes as UML Properties

existing tool that supports UML properties and tagged values. The notation can become verbose, when too many attributes and attribute parameters are used in a class, or other inner elements, e.g., method parameters. The attribute annotations applied to an element cannot be reused. The annotation has to be copied and pasted around. This notation is not very structured, especially the value part of the string, and can be error prone.

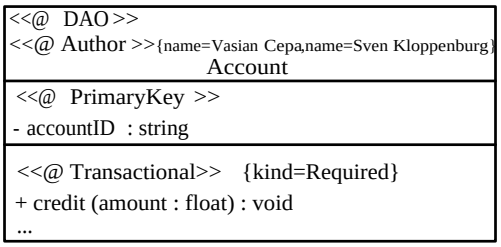


Figure 3.16: Attributes as UML Stereotypes

Stereotypes: An alternative to the UML properties and tagged values for representing attributes is to use the stereotype notation, as in Figure 3.16 [BGJ99]. In this case, the name of the attribute is used as a stereotype to decorate the UML elements. To distinguish a stereotype that serves as an attribute from other stereotypes, a special '@' character can be added before the name. An extension is needed to include the attribute parameter list with the help of a special notation, for example, by using an extended UML property notation as part of the stereotype instance name.

This notation is slightly better than the extended UML properties notation. Using the stereotype notation for the attribute name, rather than the usual property notation, shows explicitly the weight of the attribute in the design. It could also be relatively easily supported by the existing UML CASE tools. The notation is not very structured, especially the parameters part.

Template-like notation: The C++ template argument notation, supported by the UML, can also be used to represent explicit attribute instances (Figure 3.17). This notation places

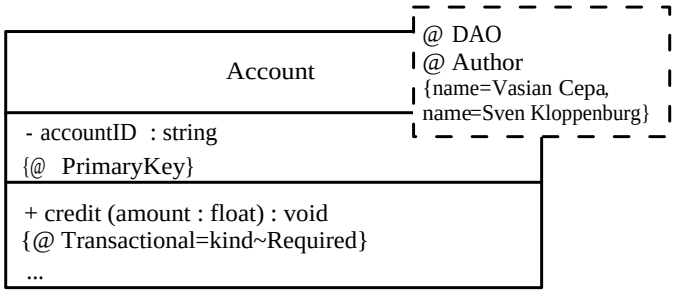


Figure 3.17: Attributes as UML Template Parameters

even more weight to the role of an attribute, and could be useful in cases where attributes are very important in the design. The template notation can also be useful when the space inside the class-name box is scarce and there is a large number of attributes with a lot of parameters that need to be specified.

The template notation will not work well with explicit attributes that annotate the inner elements of a class, e.g., object attributes, methods and method parameters. A variant of the template notation places in the template box the values of attributes for all member elements of a class, inside separate sub-boxes. This variation may require some kind of mapping between the attributes and the existing members. The notation could become verbose for the overloaded methods, where the method name is reused and cannot be used alone as a mapping tag. The template notation (without sub-boxes) could be easily supported by UML CASE tools. A benefit of the template notation is that all class-level attributes are located within a single easily visible place.

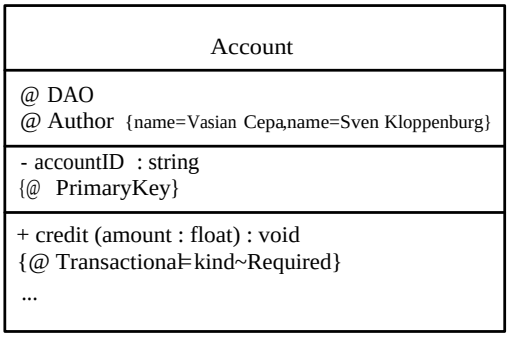


Figure 3.18: Attributes as Extra Class Sub-Box

Class partitions: Explicit attributes could form a separate sub-box inside the class notation, similar to fields and operations, as in Figure 3.18. The sub-box notation could be seen

as a natural way to extend the class definition semantics. Another variation is to have an explicit optional attribute sub-box for each kind of element in a class, such as fields and operations. The attribute sub-box is a stand-alone unit within the class model, and the notation has the same benefits and drawbacks of the template-like notation. The separate sub-box can be used in the UML tools that do not support the template notation.

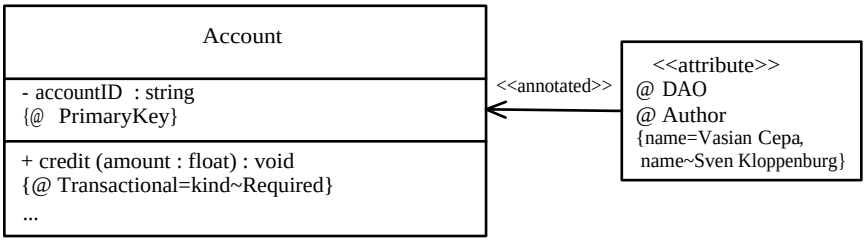


Figure 3.19: Attributes as Separate UML Class

Separate classes: Explicit attributes can be also represented by a separate class-like notation with an ≪attribute≫ stereotype associated with the class, as illustrated in Figure 3.19. There are two variations of this representation: (a) one similar to the template notion, containing all the attribute notations for all elements of a class, (b) a separate class for the attributes of each inner element, associated directly with the annotated element.

In the case of sub-elements, e.g., methods, the class notation may require that the association lines intersect the class boundaries and link directly to the methods or other inner elements, a feature that may not be supported by the standard conformant UML tools. The class notation is more reusable and more structured that the previous possibilities.

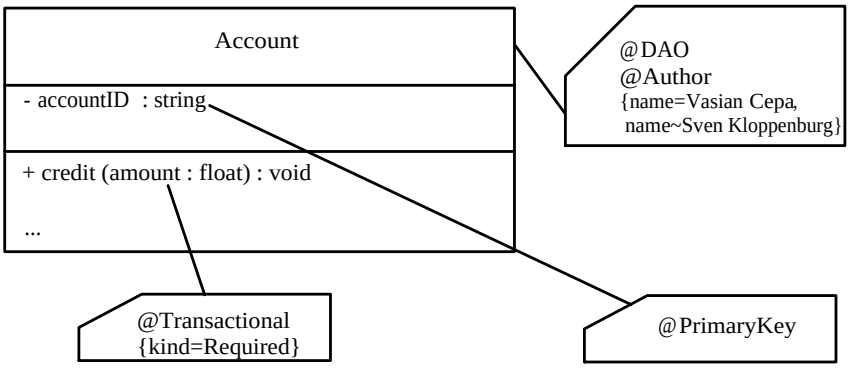


Figure 3.20: Attributes as Comment Boxes

UML comments: Comments allow arbitrary text labels to be associated with every UML element. Comments can be used to model explicit attributes as shown in Figure 3.20. Com-

ments are unstructured, so there may be a need to place some implicit structure in the text string. For example, an '@' character can be used before a name to denote that it is an attribute. The selected implicit notation can be tool and language specific. The comment notation lacks the inner structure, and it may be difficult to distinguish between attribute comments and other types of comments in a diagram. Apart of this, the notation is as reusable and flexible as the separate class notation.

3.3.2 Discussion of the UML Alternatives

Table 3.1 summaries the above discussion, evaluating the alternative UML notations discussed above against the criteria given in section §3.3.1. Plus signs indicate that a criteria is better supported. Minus signs indicate less conformance to a given criteria.

Presentation	Standard	Visibility	Clarity	Usability	Structure
Properties	+ +	- -	- -	-	-
Stereotype	+ +	+	-	-	-
Templates	+	+	-	-	+
Partition	-	+	-	-	+
Separate Class	+	+ +	+	+ +	+ +
Comment	+	+	-	+ +	- -

Table 3.1: Summary of Various Explicit Attribute Presentations in UML

As indicated by the Table 3.1, no representation is better suited than all the other with regard to all criteria. The separate class notation seems to fulfill most of the requirements, however, it may have difficulties to represent the annotation of the inner class elements.

Table 3.1 shows that some notations are better suited that the others is some direction. This is a consequence of the various scenarios that could be covered with explicit attributes in a UML model. Depending on the relative weight of the attributes in the design, their parameters, and the density of the attribute usage in a class, or in the class inner elements, different notations may be useful in different situations. For example, it could be chosen to represent attributes that serve as marking interfaces [NV02] by stereotypes, given that this expresses their importance. When many explicit attributes are used to drive generation and they are repeated over classes, a separate class notation makes more sense. For fine grained notations, e.g., method annotations, or method parameter annotations, an extended variation of UML properties is less verbose and hence better suited. More than one notation could also be combined in the same diagram, as was the case with some of the UML examples in section §3.3.1.

Finally, based on the Table 3.1, it can be concluded that the stereotype and separate class notations are the two most usable notations that should be considered as a first choice to model

the explicit attributes in UML class diagrams. In several cases, some notation based on the UML properties is needed to augment the other representations. Several of the analyzed extensions may not be supported by all UML CASE tools.

3.4 Languages with Generalized and Annotated Abstract Syntax Trees

This section discusses language technology organized around annotated Abstract Syntax Tree (AST)-like representations of program structure, used to support attribute-based DSA in product-lines. This section also discuses the impact of such technology on the processing of attribute annotated code entities.

3.4.1 Attribute Language Support Example: .NET Languages

To illustrate the relation between annotations and AST-like representations of the program structure, this section considers .NET framework [Pro02], as a representative of systems with explicit support for tag interpretation, by means of access to source, or binary representations of programs. .NET follows a hybrid approach with respect to attributes. It distinguishes between *predefined* and *custom* attributes. Predefined attributes are used by various library API-s that come bundled with the .NET platform. For example, `[System.Diagnostics.Conditional-Attribute]` is used by the compiler to include methods conditionally in the compiled version. In contrast to the predefined attributes, custom attributes do not have a meaning to the compiler. The code to interpret custom attributes has to be implemented explicitly by the developer that introduces the attributes to model domain-specific concepts.

In .NET, an attribute is a normal class derived from a predefined `System.Attribute` class. Attributes are part of the type system and can also be marked with other attributes. The interpretation code can be placed inside the attribute class itself. When a larger context needs to be processed, in order to interpret the attributes, the interpretation code could also be placed in a separate module.

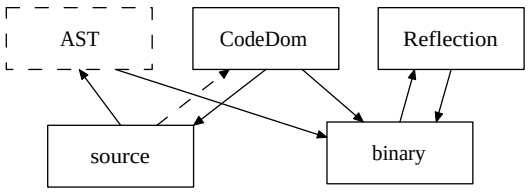


Figure 3.21: .NET AST-Like Program Representations

Figure 3.21 shows a high-level view of the .NET API-s that support access to different AST-like representations of a program. There are by default two main ways how .NET API-s can be used to process attributes:

- *Source code manipulation.* The .NET `System.CodeDom` API [Har03] supports source-level AST-like representation and manipulation of a program. An implementation of this API can be used to build an AST, either manually, or from the source code of a program (by means of `ICodeParser`[17]). Next, the constructed AST-like representation can be transformed and the result AST is saved (back) as source (by means of `ICodeGenerator`), or it is directly compiled to a binary file (by means of `ICodeCompiler`[18]).
- *Run-time manipulation.* The .NET `System.Reflection` API can be used to (a) introspect a binary for its structural elements and the attributes they are decorated with, as well as in the reverse direction, (b) to create executable modules (assemblies). For the latter purpose the `System.Reflection.Emit` API can be used (in combination with the Reflection API) to generate the method internals. While the Reflection API deals with creating the program structure elements, e.g., classes and methods, the `Emit` API deals with Intermediate Language (IL) opcodes used inside methods. The `Emit` API works only in one direction. It can generate new assemblies, but cannot access or modify the IL representation of the existing ones.

.NET API-s provide an infrastructure for creating and accessing AST-like representations of a program beyond the parsing stage of the compiler. Figure 3.22 shows how this infrastructure can be used to interpret attributes in .NET. The first step to utilize the custom attributes in .NET consists in defining new attribute classes (when needed), and using them to decorate the code. The exact actions to be performed next depend on the semantics associated with the used attributes. In a next step, the source-level AST, obtained via `CodeDom`, is used to modify the source code. Alternatively, the code is compiled and the attributes become part of IL binary meta-data. The IL meta-data are accessed later-on at run-time, via the `Reflection` API, and are processed to undertake user-defined actions. Post-compilation manipulation of existing IL binaries is not directly supported by the .NET API-s.

3.4.2 GAAST-Based Language Technology

Figure 3.21 contains also a dashed box named `AST`, which is not discussed so far. This box represents the AST that is internally constructed by the compiler, during the compilation process

[17] The current .NET framework language specific providers do not implement `ICodeParser`. For this reason, the connection from source to `CodeDom` box in Figure 3.21 is drawn as a dashed line. A free third-party implementation for C# is *CS CODEDOM Parser* [Zde02].

[18] `System.CodeDom.Compiler` interfaces (and helper classes) must be implemented by a `CodeDom` compiler provider.

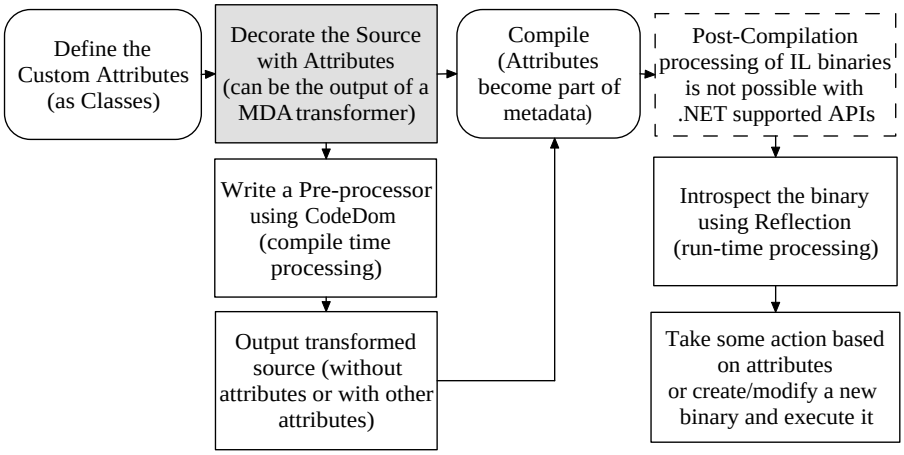


Figure 3.22: Processing Attributes in .NET

of the source code. In .NET, this AST is not available to the end-programmers. The AST box is shown in Figure 3.21 to emphasize the similarity between the different .NET AST-like representations of the program (aimed at supporting program transformations), and the source AST built by the compiler. The .NET CodeDom, or Reflection ASTs, and the compiler AST represent the same information at various levels of detail.

This similarity suggests the idea of having a single common AST-like representation of a program, that would be used by the compiler, as well as, by other attribute-driven transformers in a programming language with annotation support. Such an AST API, that generalizes over different ASTs of the same program and supports annotations, will be called a *Generalized and Annotated AST* (GAAST) API. There are several aspects of a GAAST API that need to be especially discussed:

- *Modeling AST differences.* As illustrated in Figure 3.23, there could be differences in the information found in different representations of the program. Some nodes present at source code are not preserved in the binary representation, or could be presented by different element nodes (gray filled in Figure 3.23). A common example is deploying a prefix or suffix increment / decrement operator. The GAAST API is intended to support attribute-driven transformation for implementing DSA constructs. The source code AST representation is too fine grained attribute-driven transformers. The GAAST API can unify the structural information and leave out the other details. Some nodes can be present in the interface, but a given implementation of GAAST may choose not fill all nodes with valid, or fully parsed information (§3.4.3). This situation is represented by different fillings of (possibly empty) nodes in the unified AST, at the bottom of the Figure 3.23.

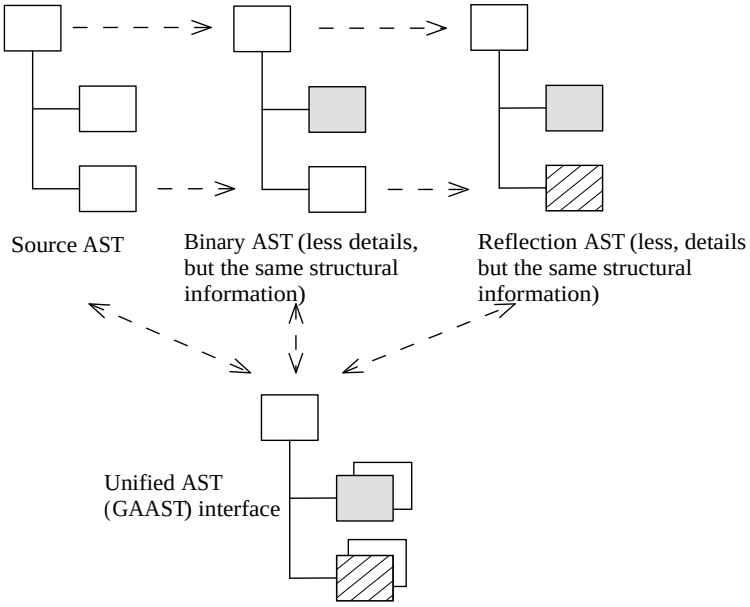
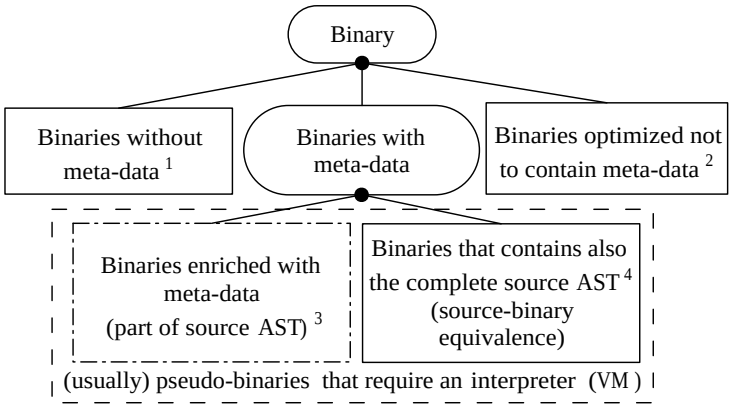


Figure 3.23: Unified AST Representation

- *Matching the language features.* GAAST reflects the features and the dynamicity of the underlying language system. The source code manipulation enables static transformations. Many language systems support only static transformations, because they do not save the AST information as part of their *pseudo-binaries*. Figure 3.24 shows a rough classification of what is understood by binaries in this section. The meta-data, as shown in Figure 3.24, is a way to explicitly save part of the AST obtained from the source code along with the compiled code. The amount of the AST information saved as part of meta-data determines the level of reflection that is possible at run-time in a given language system. The reflective transformations require languages that have enough meta-data to support run-time introspection, represented by the dash-dotted box in Figure 3.24.

  Languages, such as Java or .NET, save almost all structural information found in source code as part of the binary meta-data. These language technologies usually do not preserve a one-to-one map of those parts of the AST that represent control-flow (behavior), e.g. *for* loops. Obtaining such fine-grained control structures requires some reverse engineering. In languages that support meta-data as part of their binaries, the GAAST API could support also static binary manipulation. In reflective languages, GAAST may support either static or dynamic transformations, depending on the language run-time support for reflection.



[1] E.g. DOS EXE files, contain "very few" meta-data
[2] E.g. DOS BIN files, or specially encoded EXE files have almost no meta-data
[3] E.g. Java class files, or .NET assemblies, and to some extent COM TLB files
[4] E.g. any interpreted language, where the source is not compiled, but is reused every time

Figure 3.24: GAAST Relation to Meta-data

- *GAAST as a Language Workbench.* Extending a programming language to support a GAAST-like API is relatively easily, when only specific features of GAAST are needed. For example, any static meta-programming framework can be used as a GAAST-like API for static source code transformations, as long as the framework preserves and enables access to the AST attributes. This makes a GAAST-like API an attractive lightweight language expression to support attribute-based transformations in mobile product-lines (§5).

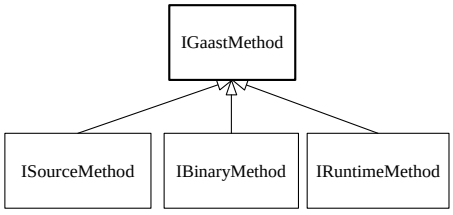


Figure 3.25: GAAST Language Information Organization

A more advanced view is to consider GAAST as part of the language system itself. That is, to make the unified GAAST API a central part of a language framework supported by the language vendor. The other API-s, such as the compiler AST, `Reflection`, or `CodeDom`-like API-s, could be organized around the common GAAST interface, as

illustrated by the method interface example in Figure 3.25.

A language with GAAST support for attribute-driven transformations can be seen as a language workbench [Fow05] to sustain attribute-based DSA. The GAAST API serves as a *contract interface* between the language workbench features, provided by the language vendor, and third-party attribute transformers. It becomes the responsibility of the language vendor to maintain the contract interface as the language evolves. Several languages, e.g., .NET or Java, are already moving into this direction.

In a GAAST-enabled language, the attribute transformers are immune to most changes in the language workbench (e.g., several syntax changes). This is difficult to warranty with third-party implementations that need to be separately maintained, adding accidental complexity to the product-line. In a GAAST-enabled language, the transformer developers would not need to reinvent helper API-s and tools that are covered by the GAAST API. Transformers could reuse third-party modules build on top of the central GAAST and better leverage rapid prototyping. The language platform vendors would also profit from the unified GAAST API, since by unifying several API-s, the total cost of the language platform is decreased and the language system becomes also more attractive to the programmers.

GAAST addresses only the parsing issues related to source or meta-data enriched binaries. Attribute-driven transformation based on GAAST-like API-s will be fully discussed in chapter §4. Specialized abstractions could be built on the top of the GAAST API that would facilitate building transformers. There are two concepts that can be used in combination to implement generic abstractions over a GAAST infrastructure:

- GAAST can be enhanced with declarative query capabilities. An example of query capabilities is `JPath` API for Java developed as part of `EXTRACT` [Cal03] compiler system. JPath defines a set of operations for selecting nodes from a Java AST, borrowing the idea from W3C XPath [XML99] standard for XML [SG01]. Another example of query support directly embedded in the langauge, is the query expression pattern of .NET C# 3.0 language specification [Mic06a], based on the LINQ [Mic06b] project.

- Support for declarative specification of transformations. This is similar to the ideas put forward by OMG MOF Query / Views / Transformations (QVT) proposals [MOF03]. In terms of QVT, JPath is a query and view language. QVT also exposes means to define transformations. Transformers take a view as a parameter and map it to another view. This is similar to graph rewriting techniques [DJ90, Men99, Agr04], but provides a more declarative way to define graph transformations. At the time of this writing a final MOF QVT standard is not yet available.

3.4.3 Implementing GAAST on Top of .NET

Having stressed the importance of supporting GAAST as a central part of the language technology, Figure 3.26 shows how a third-party prototype GAAST API can be implemented on top of the existing .NET API-s.

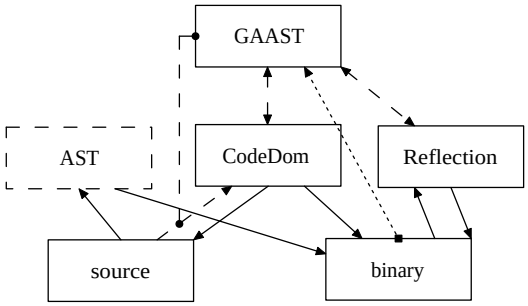


Figure 3.26: Implementing GAAST on Top of .NET

The dotted arrow, from the `binary` box to the `GAAST` box, represents the required custom code to expose the method internals of IL binaries through the GAAST API, which is currently not supported by the .NET Reflection API. The dashed line connecting `source` and `CodeDom` boxes represents the implementation of the `ICodeParser` interface, which is also currently missing in .NET.

Unfortunately, a third-party API cannot directly reuse the AST built by the language compiler(s), and does not have access to the .NET Common Language Runtime (CLR) information, unless a supported interface to expose this functionality is provided. The lack of this interface renders the implementation and maintenance of a third-party framework difficult, especially in face of the evolution of the .NET framework.

As an example, consider Figure 3.27, that shows how the *methods* information is modeled by the .NET Reflection API. The Reflection API enables the introspection of a method to get information about it, e.g., to find the custom attributes that have been used to annotate the method and its return type. When using the Reflection API, the types returned by the methods, e.g., *ReturnType*, are properly resolved, and the system has access to the full loaded assemblies, which makes it possible to return, for example, the attributes inherited from a base class method (the `bool inherit` option of the `GetCustomAttributes` method).

Figure 3.28 shows how the same information is modeled by the .NET CodeDom API. The first difference concerns the returned types. For example, the type returned by the `ReturnType` is not a fully-resolved `Type`, but a `CodeTypeReference`, which could be either an unresolved user defined type, or a resolved system type. Another difference is that the `Custom-`

```
1  namespace System.Reflection {
2
3   public class MethodInfo : MethodBase {
4
5    public Type ReturnType
6    { get{ ... } }
7
8    public object[] GetCustomAttributes(bool inherit)
9    { ... }
10   ...
11  }
12 }
```

Figure 3.27: Reflection API Method Representation

```
1  namescpace System.CodeDom {
2
3   public class CodeMemberMethod : CodeTypeMember {
4
5    public CodeTypeReference ReturnType
6    { get{ ... } set{ ... } }
7
8    public CodeAttributeDeclarationCollection CustomAttributes
9    { get{ ... } set{ ... } }
10
11   public CodeCommentStatementCollection Comments
12   { get{ ... } }
13   ...
14  }
15 }
```

Figure 3.28: CodeDom API Method Representation

`Attributes` method cannot resolve hierarchy information. Information for additional nodes, e.g., code comments, is also present, which makes no sense for binary files. The most profound difference is the capability to modify the information, as can be noted by the `set` version of the supported properties in Figure 3.28. The Reflection API provides read-only access to the method information.

A possible common GAAST-like interface is shown in Figure 3.29. The GAAST interface enables writing the same structural transformations, despite the underlying program representation. When no complete native language support for GAAST exists, as is the case with .NET, a third-party GAAST API, based only on the CodeDom and the Reflection API, could only offer a set of uniform interfaces to access the common information. To initialize the GAAST interface, a factory pattern [GHJV95] can be used, similarly to the way .NET CodeDom supports different .NET languages. The factory can support either source, or run-time representations. A trade-off

```
1  namescpace Gaast {
2
3   public interface Method : Member {
4
5    public Type[] ReturnType
6    { get{ ... } set{ ... } }
7
8    public CustomAttribute[] CustomAttributes
9    { get{ ... } set{ ... } }
10
11   public Comment[] Comments
12   { get{ ... } set{ ... } }
13   ...
14  }
15 }
```

Figure 3.29: GAAST API Method Representation

to model the AST differences could be to ignore representation specific information and throw a missing implementation exception on `set` methods, when only a read-only view is supported.

In practice, unless GAAST languages become available, a third-party GAAST API implementation can be reasonably limited, by implementing only those aspects of the GAAST technology that are needed by a specific transformation system. As the focus is on DSA support with attributes, only the static aspects of GAAST API are explored in attribute-driven transformations. Various aspects the GAAST technology for supporting static and run-time attribute transformations are investigated in the upcoming chapters. A custom GAAST-like API build for Java 2 Micro Edition MIDP [Mob02], which uses JavaDoc [Pol00] tags to emulate annotations, is described in chapter §5. The Tango transformation engine, introduced in chapter §4, supports a GAAST API for static transformations based on a common XML representation of the program structure, enabling either source or binary transformations, if they can be expressed in the common XML format. The Attribute Dependency Checker tool of chapter §4 uses the run-time aspect of GAAST, based on the .NET Reflection API, to automatically enforce system-wide attribute dependencies.

3.5 Comparison to other DSL Approaches

Attributes could be used to emulate DSA constructs in mobile product-lines. There are two aspects how this particular usage of attributes is related to other generic systems that can be utilized to implement DSA:

1. Attributes can be seen as a convenient way to extend the grammar of the language. Attribute-based DSA only cover a subset of possible EDSL approaches, and are limited in the grammar

changes they can introduce. Attribute languages enable implicit grammar extensions. Section §3.5.1 discusses this aspect of attributes languages, as languages with implicitly extensible grammars.

2. When combined with AST manipulation, e.g., via a GAAST-like API, attributes enable meta-programming techniques. Section §3.5.2 discusses the relation of GAAST with other meta-programming systems, intended for easing the DSL implementation. The discussion of the relation between the attribute-driven meta-transformations and other transformation techniques will be postponed until chapter §4, after the attribute transformations for mobile product-lines are explained.

There are several approaches [vDKV00, Kam98, CMA94, BLS98, Vis01b, Hud98, BP01, IR97, BH02, TCKI00] that have been proposed to reduce DSL and EDSL implementation costs. These approaches address (i) grammar extensions and (ii) parsing costs, as well as, the (iii) interpretation costs. As it will be argued in the remainder of this section, with these approaches, the start-up costs remain still higher than those of other variability mechanisms used for product-lines, e.g., OO frameworks (§2.1.3). These approaches also often introduce heavy external dependencies on third-party tools (§2.1.5).

GAAST-enabled languages remove entirely the costs related to (i) grammar extensions and (ii) parsing, for the category of EDSL extensions that can be modeled as attributes (§3.5.1). The advantage of using attributes directly in a GAAST-enabled language is that the programmer does not need to deal directly with the grammar modification [TS97]. Attribute declarations and usage are supported by the system and do not require changes to the compiler. In a GAAST-enabled language, explicit attributes [BCVM02, Hed97] add new semantics to the existing nonterminals of the grammar. The core grammar does not change, only its semantics change selectively. The change of the semantic applies only to the annotated elements. This enables supporting more than one set of attribute-based DSA at the same time, in a given GAAST language. Chapter §4 explains techniques that address the (iii) interpretation issues of the attribute-based DSA.

For an analogy that illustrates the relation between the attribute-based DSA approach and the other open language supported meta-approaches [CMA94, BLS98, BP01, IR97, BH02, TCKI00] with regard to (i) grammar extensions and (ii) parsing, consider meta-modeling with MDA MOF [Fra03, Met02] and UML [OMG03]. To extend the elements used in a MOF model, its meta-model needs to be extended. The meta-model extension requires knowledge about the meta-model and how to modify it. An alternative solution is supported in UML by the means of profiles. With profiles, a UML meta-model can be extended without explicitly modifying the meta-model definition. UML profiles achieve such modifications by restricting the types of possible meta-model extensions to stereotypes, typed values, and constraints. These three generic mechanisms enable changing the semantics of existing meta-model elements selectively. Furthermore, UML profiles enable to define these extensions at the same abstraction level as the

model, making the meta-model manipulation implicit.

Attribute-based DSAs, unlike the other approaches, make the manipulation of the language meta-model implicit. Explicit attributes can be logically mapped, more or less, to UML parameterized stereotypes (§3.3). Similarly to MDA MOF, the attribute-based DSA introduces the possibility to manipulate and extend the language grammar meta-layer, a feature that is missing in most widely used programming languages. Attributes are a central part of a GAAST-enabled language, forming a low-cost language workbench [Fow05] to support DSA.

3.5.1 GAAST Languages and Extensible Grammars

Programing languages need often to be extended with various constructs, which are made part of their syntax. Extensible compilers [ZO01] support language extensions as part of their original design. Extensible compilers require the modification of the parser, the modification of the semantic analyzer, and other back-end passes (e.g., optimization and generation [Muc97]) [ZO01].

AST annotations are traditionally used during contextual analysis and code generation in compilers [WB00, WMBK02]. This book promotes the use of the AST annotations explicitly [BCVM02] in the source code. The GAAST languages, described here, can be seen as an extensible front-end of a compiler. A grammar that support explicit attributes (tags) will be called a *tagged grammar (TG)*, a term that does not conflict with any other existing grammar terms [GJ90]. The tagged grammar term is used to refer to the grammars that support GAAST languages. A compiler that supports a TG-based language has an extensible front-end.

Any language grammar can be extended. There are approaches, e.g., extensible grammars (EG) [CMA94] that structure the changes applied to a grammar, into a series of basic modification operations. EG reduce the costs of introducing and maintaining incremental grammar modifications. An EG is a grammar augmented with *constructor* functions for creating productions, and three (meta) grammar operations:

1. *addition* - introduces a new production in the grammar, by expressing it in the terms of grammar constructors,

2. *deletion* - removes a production from the grammar, by making the grammar more restrictive,

3. *update via replacement* - updates an existing grammar rule, by replacing it with the new definition.

The expressiveness of an EG grammar depends on the rules found in the *core grammar*. Phobos [GH02] is a system for Java based on the ideas of extensible grammars [CMA94], that structures the grammar changes as module (grammar) inheritance. The focus of this section is

in TG expressiveness for supporting EDSL, as compared to more full-fledged EG. There are two ways how a TG can be utilized to support attribute-based DSA:

a. *As an extensible grammar* - TG are restricted in the ways they can modify the core grammar. A TG cannot introduce arbitrary changes. The expressiveness of a TG is limited as compared to an EG. In formal terms, a tag can be seen as a nonterminal symbol $\tau \in T$, with T being the set of the allowed tag strings[19]. A tag that accepts parameters can be expressed as tags without parameters (§3.1.2). A tag enables adding only productions of form $N_{new} \rightarrow \tau N$ to the grammar, where N_{new} is a new nonterminal and N is a nonterminal that exists in the original grammar. An EG supports adding productions of form $N_{new} \rightarrow \alpha_1 N \alpha_2$, where α_1, α_2 are either nonterminals or terminals. This means that a TG can generate less string forms than an EG.

b. *As a transformation system* - In this view, the annotation capabilities of TG are used to mark transformation points (explicit hooks [Aßm03]) over the elements generated by the core grammar. The modifications of the core grammar are external to the grammar itself. The production $N_{new} \rightarrow \tau N$ is interpreted as a transformation function $\tau : N \rightarrow \tau(N)$. Tag parameters can also be modeled as a parameter vector π, so the generic tag transformation function becomes $\tau : N \rightarrow \tau(\pi, N)$ (§4.1.2). The function τ maps the domain N into a co-domain $N_{new} \in TG$[20]. This way, while remaining in TG, any transcendental transformation semantics can be applied via τ. The expressiveness of the semantics, as in the case of EG-s in [CMA94], depends on the core TG (without tags).

It is the arbitrary transformation view of a TG that is of interest in practice. The grammar production context is removed in a TG, from the grammar to the transformation system. When combined with a transformation system, a TG can be used to express the same semantics as an EG with minimal changes. Any computationally-complete general-purpose programming language can be used as a candidate for a core TG. In practice the implementation of transformation τ could be easier, if the language supports at least directly some of the language mechanisms intend to be used by τ. For example, attribute transformations applied to classes are easier to support in an OO language, where the classes are natively supported, rather than in a non-OO language.

GAAST languages can be also used to support other program transformations apart of attribute-based DSA. The GAAST API (and its implementation) could help with technical concerns related to AST processing, so the effort to obtain the AST for the source code is not repeated in custom transformers. The GAAST does not replace other approaches for building EDSL-s. GAAST is rather an intermediate transformation tool that can be used in the intermediate transformation phases of other tools that need to do AST processing.

[19] The T set is infinite, but only a finite number of strings are used from it in a program.
[20] N_{new} may map into several productions in the core TG.

3.5.2 Meta-Programming Approaches

The approaches discussed in this section [TCKI00, BLS98, The02, BP01, IR97, BH02] ease one or more aspects of DSL, or EDSL implementation. They, however, do not make the grammar manipulation as implicit as GAAST languages do (§3.5). Some of these approaches can also be used to implement attribute-driven transformers. The reverse is also true. GAAST-enabled languages remove the need for many of these third-party approaches.

Meta-programming systems are language systems that contain support to process and modify the elements of their own meta-model. There are two categories of meta-programming systems:

- *Static* meta-programming systems, e.g., LISP [Gra95], or OpenJava [TCKI00] are very similar to the GAAST concept. They provide access to the meta-model of a program and enable source-to-source manipulation. Some static meta-programming systems, e.g., OpenJava [TCKI00] or Jak [BLS98], support adding new types of terminals not found in the core grammar, and enable translating these new constructs to the core grammar entities.

- *Reflective* meta-programming systems, e.g., Smalltalk [GR89], Self [The02], or ECMA-Script[21] [ECM02], enable programs to query and, depending on the system, to modify their own meta-information. Forms of reflection can be found in all modern general-purpose programming languages, e.g., .NET and Java. Reflection in these systems is limited to querying information only. Other systems, e.g., Self [The02], can also modify their types and type hierarchies at run-time.

The only way to extend a GAAST language are attributes. This means that generic meta-constructors are not required in a GAAST language. The meta-model is only implicitly extended. Only when the attributes are processed the meta-model manipulation becomes explicit in a GAAST language. A GAAST API reflects the properties of the language. If a language supports reflective meta-programming, a GAAST API for that language could be made available. In languages, such as .NET, the GAAST API enables only static meta-programming[22].

The remainder of this section presents two examples of static meta-systems in more detail. The selected examples were chosen because (a) they are implemented as generic language extensions, introducing external third-party dependencies in the systems that rely on them, and (b) they are not strict meta-systems, but contain also features that support EDSL implementation.

Jakarta Tools Suite (JTS) [BLS98] facilitates the DSL creation by offering a set of related tools that address the complete process of implementing DSL. The JTS tools include:

[21] The JavaScript OO model is quite similar to Self.

[22] Reflection in .NET can be combined with a GAAST API, but full reflective meta-programming capabilities are not supported.

- *Bali* is parser generator used to support parsing custom DSL grammars, similar to JavaCC [Jav02b] or ANTLR [Par02]. Based on a given grammar specification, Bali generates a lexer, a parser, and a hierarchy of Java classes to represent the parsed AST.
- *Jak "is an open, extensible superset of Java"* that extends Java with *"support for meta-programming"*, and *"enables Java programs to write other Java programs"* [BLS98]. Jak provides access the parsed AST and enables manipulating it using Java-like code. Jak can be used alone, or as a back-end for Bali. Jak can parse at run-time uncomplete code snippets (called surface syntax trees) and can validate them with regard to types and symbols in the context of another AST. Jak can also support generation scoping by limiting the identifiers scope to sets of related code fragments (environments). Jak provides also several predefined AST traversal operations similar to Stratego [Vis01b].
- A mix-in [BC90] *way* to compose language extensions and any set of mix-in features, known also as *GenVoca* [BST+94] generators. GenVoca is a scalable model for composition of component-based software that generalizes the concept of mix-ins[23].

JTS tools have been successfully used to implement complete DSL that support product-lines [BJMvH00]. Dealing with grammar evolution is still explicit in JTS approaches. Jak supports source-to-source transformations and is similar to the .NET CodeDom API, but has also generic constructors to create new types of statements or expressions.

Being third-party extensions, Bali and Jak need to be explicitly maintained as the Java language evolves. For example, the evolution of Java from version 1.4 to 1.5 was not followed by the JTS. To overcome these issues, in section §3.4.2 it was required that GAAST be part of the programming language technology, as it is the trend with .NET (CodeDom, Reflection API-s) and to some extent with Java 1.5[24].

In order to benefit from GenVoca compositions, clear decomposition of the domain features must be available. However, many software problems do not have a clear feature-based decomposition structure [MO04], so that they cannot be easily composed as chains of independent features[25], reducing the applicability of GenVoca compositions for organizing the AST transformations.

GAAST-enabled languages address only attribute transformations. This means that GAAST-enabled languages are a special case of a meta-programming system. In a GAAST language, a tool, such as Bali, is not needed, whereas support for the AST manipulation is assumed to be part of the language, removing also the need for Jak-like tools.

Macro systems [BP01, IR97, BH02] can be used to support DSA constructs. Macros are used in languages, such as C, to extend the language with declarative constructs. Macros are processed

[23] In the sense that every possible software composition is treated as feature mix-in.
[24] An unsupported API similar to CodeDom is distributed from Sun with Java 1.5.
[25] A feature-based decomposition structure is typical for abstract data collections and some mathematical libraries.

by a preprocessor, usually by replacing the definition with the macro code. The systems mentioned above support *syntactically rich* macros, where the macro definition can access the AST of the code at the point of the macro application, and modify its own behavior based on the invocation context. This enables implementing more powerful macros, which are not possible in languages, such as C. Syntactically rich macros can be used to implement DSA constructs, and help to address grammar and parsing issues.

While the mentioned macro systems do not have support for attributes, such support could be easily added. The macro systems could be used for static source-to-source attribute-driven transformations, similarly to the .NET CodeDom API. Syntactically rich macros are usually implemented as third-party language extensions, and have the same problems with regard to the language evolution as Jak [BLS98].

Summarizing, GAAST languages have a more limited scope than all other extensible meta-programming systems. Attributes support only a very specific set of EDSL, and not all possible EDSL can be expressed as explicit attributes. What makes attributes attractive is that, they are either part of a GAAST language, or require a minimum one-time effort to be added. Attribute-based DSA can also be processed with any meta-programming tool that exists for a given host language, requiring minimum new investment in a product-line. Extending the meta-model of a language with custom attribute-based DSA does not require any knowledge about the grammar manipulation and parsing. When attribute processing is part of the language (GAAST is supported), there are no external dependencies of a product-line to third-party DSL implementation frameworks. Attributes expose a uniform programming model, free the user from having to learn new syntax, and have minimal education and training costs compared to the other approaches.

Attributes have the lowest start-up costs for supporting DSA in a product-line. While attribute-driven transformations should still be applied, most of the time they require a simple mapping from the attribute-based DSA, to the component libraries implementing the common domain functionality. Attribute-driven transformations add only a very small burden with respect to the implementation cost of component libraries, and can be used as a mechanism of choice for iterative development of a product-line.

3.5.3 AOP and DSA

Aspect-Oriented Programming (AOP) [KLM+97] was mentioned in section §2.1.5 as an example of generic language systems that can be used to support DSL. AOP was technically discussed in section §2.4. This section compares AOP and DSA along two dimensions: (a) AOP as a way to implement DSA, and (b) AOP as a replacement for DSA.

AOP as a way to implement DSA. Section §2.4.2 discussed that AOP engines can be used as generic invasive systems to implement various transformations including the interpretation of

the DSA constructs. There are, however, several liabilities, that prevent AOP engines from being used to implement arbitrary DSA constructs:

- *Hardwired joinpoint / pointcut models.* A distinction can be made between AspectJ [Lad03, KHH+01], and more recent AOP systems. AspectJ is a declarative meta-programming system that hides the meta-model manipulation from the end programmer. AspectJ has a hardwired joinpoint / pointcut model that, as explained in section §2.4.2, covers many generic meta-programing scenarios, but not all. Adding support for different, extensible joinpoint and customizable pointcut models is possible [Asp04, CN04], but could remove some of the benefits of stated in section §2.4.2, e.g., static checking, invested efforts on IDE support, and declarativeness. It also blurs the distinction between AOP systems and other kinds of meta-programming, or open compiler systems.

- *Limited vertical[26] transformations.* A software system is often programmed using more than one language. For example, Java, SQL, various script and declarative languages based, for example, on XML, or visual languages can be used together. DSA can introduce abstractions that crosscut more than one single language technology. DSA enrich the meta-model of a language, and can isolate alien parts of a system to make them look native in any technology [Rie96]. For example, Hibernate [Hib04] provides Java abstractions around object / relational JDBC [JHF01] databases. When using Hibernate, the relation schema of a given database becomes accessible from Java, as normal Java classes. This means that, the AspectJ engine can also be used to manipulate them, which was not possible without Hibernate generated wrappers. Introducing technology wrappers only to support AOP techniques can be costly.

  AOP engines have limited support for vertical transformations that cross-cut more than one meta-model. Aspect engines that support more than one language meta-model, need to work upon some common meta-model of all supported system models. In terms of MDA MOF, such a common joinpoint meta-model will be a M2 level model (§2.1.4). M2 level models are very generic, to be useful for very specific transformations, which makes them not as suitable as language specialized aspect engines. For example, Figure 3.30 show schematically the inner workings of an XML-based AOP approach [EMOS04] for cross-model manipulation. This approach works at the same logical level as MOF M2 level (the meta-model of XML itself is M2 level with regard to data modeled in XML)[27]. The approach supports only the query view of the execution graph (§2.4), because xQuery [XQu05] requires access to the total XML DOM tree. The M2 level compatibility found in [EMOS04] should not be confused with language families, such as the .NET language family [Wea04], that share the same meta-model, but map it to different concrete syntaxes.

[26] Horizontal transformations are used in this section to mean transformations that work within the same meta-model. Vertical transformations work across different meta-models [GSCK04] (§2.4).

[27] The reverse is also true. The MOF can be equivalently mapped to the XML meta-model. An example is XMI [Fra03].

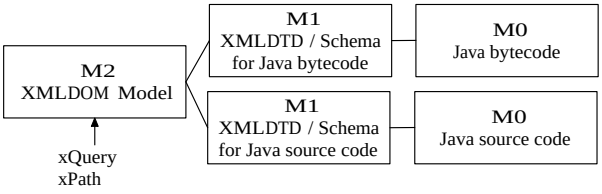


Figure 3.30: The XML MDA Levels

Working at the meta-model level unifies many specific issues that are usually important to be ignored. For the example of Figure 3.30, the XML queries will be more useful if they refer to the information from the M1 level models. Elements of the M1 models, e.g., language specific features, can be explored in a joinpoint model to allow more semantically rich pointcuts. This is not possible when working with the M2 level elements. However, queries specialized for one M1-level model cannot be used for another M1-level model.

AOP as a replacement for DSA. While any generic transformation system can be used to implement DSA, no generic transformation system can replace DSA. DSA promote an explicit programming style and enrich the meta-model of a language. The programmers select explicitly the context where DSA constructs will be used. AspectJ and AOP replace the explicit programming style with selection of points of interest, enabling in theory more automation. DSA preserve the domain architecture in the code. Whereas AOP, when used as a replacement for DSA, hides the domain architecture, which makes it not suitable for emulating the DSA semantics. While AOP techniques cannot replace DSA, neither can DSA replace AOP techniques. They complement each-other and could be used together in a system. AOP engines can be used to apply AOP-style modularization in a system, working over the DSA constructs.

The consequences of AOP-style modularization are outside the scope of this work. Technically speaking, matching AOP pointcuts is a mechanical process (§2.4.1). While pointcuts are declarative, the points of interest need to be specified often as complex composed pointcuts. The shadows of complex composed pointcuts cannot be directly imagined by humans for large systems. This means, that there could be points that are matched when they should not, and vice-versa. Investment in IDE support for AspectJ can help with this regard for a particular AOP system. Other systems [OMB05] explore less fragile pointcut models to deal with evolution of source code artifacts. While this is an area of active research, the mechanical pointcut matching is still far from replacing the human search and could result in unpredictable transformation side-effects. Attributes can be used as a workaround to express complex pointcuts, that are too complicated to be expressed otherwise. AOP techniques will not be explored any more in this book. The focus will be on the DSA support with attributes, and it will be left up to the user to decide the most appropriate programming model, on a case by case basis.

3.6 Proper Usage of Explicit Attributes

Attributes are easy to introduce and can carry any semantics. This can lead to *abuse* of AEP, when attributes are utilized to represent semantics that can be represented better with other language means. This section discusses the proper usage of explicit attributes, giving several examples of inappropriate practices.

3.6.1 When to Annotate?

Every program element can be decorated with explicit attributes. There are natural ways to define the semantics of abstract software models in all programming languages. The OO language abstractions are usually easier to use than the explicit attributes, and are also better statically checked by the language compiler with no additional effort on the part of the developer. The following principle summarizes the first guideline for properly using explicit attributes.

Principle 3.6.1 *Explicit attributes should be used to decorate an entity, when there is no simpler natural representation of the intended semantics, supported by the hosting language.*

The term *natural* is intuitive [RH04], and so is the usage of tags to denote extended semantics. Everything, that can be done with attributes, can also be done without them in a Turing complete [HMU01] language. According to the principle 3.6.1, attributes help to reduce the accidental complexity, and to make the abstractions easier to introduce and implement. Several alternatives how to use explicit annotations are possible and have already been used, however, not all of them follow the principle 3.6.1, as illustrated by the examples below:

- Attributes can be used to decorate any element if it is allowed by the language grammar. In an OO language, decorating a 'Car' object with a 'color' tag can be more naturally implemented with a field attribute of the class 'Car'. In .NET C# [Pro02] decorating a structure with a custom attribute 'class' to denote OO class semantics makes no sense, given that the available 'class' keyword is the natural choice.

- Explicit attributes can be used to augment programming languages with new constructs in a convenient way. Decorating a structure with a custom attribute 'class' makes sense in the ANSI C language to denote a class. Attributes enable customization of a language without changing its grammar. For example, using marking interfaces to translate marked UML class diagrams to source code results in a large number of interface derivations to represent multiple tag values, as was shown in section §3.2.1. It is more natural to deploy an attribute-based approach.

- Non-consistent designs also exist. For example, a serializable object can be distinguished at the run-time[28] if its class implements a required serializable marking interface. It adds nothing semantically to use both an explicit attribute and a marking interface [NV02] to decorate a class. However, implementing the serialization in .NET requires both an interface and an attribute, favoring the attribute for generation, and using the interface for the required serialization methods that need to be implemented[29].

- Explicit attributes can be used to represent the declared model of an application, by defining *explicit hooks* [Aßm03] where code can be inserted. What *declared model* means is a relative concept. A class definition, marked with an additional 'enterprise' attribute, contains more semantics than a class definition without this attribute. However, as shown in Figure 3.31, the implicit program model can be made automatically explicit with attributes, without adding any semantics to the design.

```
1 class WebService1 { public void Method1() {...} }
2
3 // goes to:
4
5 [Class(Name="WebService1")]
6 class WebService1 {
7     [Method(Name="Method1", Modifiers="Public", ReturnType="void")]
8     public void Method1() {...}
9 }
```

Figure 3.31: Converting the Implicit Model to an Explicit Model

  This means that the mere existence of explicit attributes does not show the existence of a meaningful declared model, upon which one can reason about the application's components.

- Attribute decorations are local. It makes few sense to use local attributes for introducing global system-wide functionality. For example, local attribute decorations are used in [Bod04] to check global properties at run-time. Placing the same code in a separate module, would be more suitable and would cut in half the implementation effort. If the checked conditions are local in scope, existing Java 1.5 assertion statements would make more sense.

[28] For example, to be able to serialize an object instance to be sent over the network during remoting [Ram02].

[29] In practice, this design makes sometimes sense, because it reuses as much of the existing language abstractions as possible, rather than relying only on generation techniques based on attributes.

3.6.2 What can be Annotated?

It may seem that in an attribute enabled language everything can be decorated. However, the *identifiable elements* of a program are different, in different representations. Sometimes the structural elements of a program, e.g., namespaces and classes, are invisible when source code is compiled to a binary file. It should be properly clarified what could be decorated, and how long this decoration is about to last. The following principle summarizes the right targets that be annotated:

Principle 3.6.2 *Only entities that exist and can be manipulated in a given context, can be decorated with attributes.*

The generic term *'entity'* was used on purpose. An 'entity' is whatever can be distinguished in some way from the rest of the environment. It can be a class, a method, an object, a thread, a 'for' loop etc. The 'entities' have to exist in the context where the annotation is applied, and when it is interpreted. A 'for' loop exists in the AST of the source code and can be decorated. But the 'for' loop does not exist in the same form in the binary output of a compiler. Therefore the 'for' loop attributes cannot be preserved in that binary representation. Another example, is the possibility to annotate object instantiation in the source code level, because the line and the syntax are known. At run-time, an object can be decorated immediately after it is created, but the instantiation process itself cannot be decorated.

As a consequence of this principle, there are three different contexts, where explicit annotations can be applied:

- *Source-code level* - Attributes can decorate structural elements (e.g., classes and methods) and behavioral elements (e.g., loops and conditionals). Attributes can be processed by a preprocessor tool, or by the front-end phase of a compiler. Attributes can also be preserved for later use in the binary meta-data. However, many source-code level attributes that support code generation (e.g., used to introduce new class fields) should be processed partially or completely before or during compilation (§4). Other source-code level attributes that decorate elements, that will not exist any more as separated entities in a binary (e.g., *for* loops), need to be fully processed before compilation.

  The attribute specifications in both .NET[Pro02] and Java [JSR03] state that attributes do not change the semantics of the code they decorate. This means that a component decorated with attributes should be able to be compiled and used even though the attributes are not processed. In practice there are many exceptions to this rule. Components decorated with attributes usually cannot be used, unless the attributes are properly processed, given that the attributes change the semantics of the component, or the semantics of the context [Low03] it lives in[30]. In other cases, attributes can be used to introduce fields or methods

[30] That is, the attributes are an integral part of the overall semantics of a component.

that are used by the rest of the code. Such scenarios arise often when attributes are used to drive code generation. In these cases, the component cannot compile without processing first the attributes.

- *Binary level* - Not all languages offer this possibility. In real binaries, the structural source code information is not present. The source-level equivalent entities are either not preserved, or are blurred in undeterminable ways by various semantically-equivalent compiler optimizations [Muc97]. Meta-data enriched pseudo-binaries found in languages, such as Java and .NET, save the structural information of the source code AST inside the binary (§3.4.2). In such languages, the annotation of the structural elements is preserved during compilation and can be accessed after the compilation, or at the run-time, by using reflective introspection. For example, Soot [PQVR+00] can statically explore and modify the Java bytecode format.

- *Run-time level* - New entities exist at run-time that are not found in the source code or binaries. For example, in the source code level there are class definitions and object instantiations instructions. The real objects exists only at run-time. The same holds true for threads and virtual methods. The need for attributes at run-time can be easy emulated with additional object attributes[31], and will not be explored any more in this book[32].

These contexts differ in the amount of static or dynamic information that is available. Different language technologies also differ in the amount of the information that is made available in each context. Java annotations [JSR03] make these contexts explicit. A Java annotation can have source, binary, or virtual machine lifetime. For example, the `@Retention(RetentionPolicy.RUNTIME)` meta-attribute denotes that a given attribute should be preserved during the execution by the Java virtual machine.

3.7 Chapter Summary

Attributes are a lightweight language extension used to introduce custom domain-specific abstractions (DSA). Attribute-based DSA support a declarative programming model for product-line development. Attribute families organize the attributes in nested name spaces corresponding to the domain assets. Attributes can be used to drive DSA transformation[33] in the language technologies that support them. An attribute enabled language provides the possibility to decorate and access the decorated AST in various (equivalent) representations of the source code.

[31] The *'Variable State'* pattern [GB04] can be used for specific objects or a lookup (hash) table for all objects indexed by the object.

[32] Using reflection at run-time to manipulate attributes is a case of binary-level attributes, and not of the run-time level attributes.

[33] Source to source, source to binary, binary to binary and / or binary to source.

Attribute enabled languages support more directly model-driven development (MDD) compared to other language technologies used to map UML class diagrams, such as, marking interfaces, or pseudo-syntactic marking. Attribute programming helps to preserve architectural decisions in the source code. Unlike interface based mapping and pseudo-syntactic mapping, attribute mapping closely resembles the original MDA models and provides for better extensibility and a less complex programming model for the targeted OO languages. The mapping process is also simpler. Attribute enabled languages help to close the gap between thinking in terms of models, and source code.

Attributes enable new ways to design the code. UML tags and stereotypes can model only a subset of attribute design possibilities. While mapping UML tags and stereotypes to attributes is straightforward, the reverse process is not directly supported in every case by UML. Several UML notations were investigated, that can be used to model different attribute-based scenarios.

GAAST enabled languages make it easier to extend a general-purpose OO language with domain-specific constructs. GAAST languages offer a single uniform mechanism to introduce custom extension. GAAST languages serve as a convenient alternative to extensible compilers, for implementing attribute-based DSA. GAAST languages do not require changing and maintaining the front-end tools of the compiler. If the GAAST technology is supported by the language vendor, it becomes easy to introduce AEP in an existing project, and to maintain attribute-transformers in the long term. Languages, such as .NET and Java 1.5, already offer a lot of support for GAAST-like transformations.

Attributes should be used carefully to enhance or replace software abstractions that are already available in a language. Adding attributes is easily, however, just using attributes *per se* adds no semantic value to the source code model. The set of the used attributes and their scope validity should be carefully selected, based on the domain concerns. Attributes can be preserved in various representations of the source code or the binary, if the language technology allows it. Depending on the interpretation scope, attributes can be used at (pre) compile time, after compilation, or at run-time.

Chapter 4

Building Modular Attribute-Driven Transformers

All animals are equal, but some animals are more equal than the others.

G. Orwell, Animal Farm, 1946

All transformations are equal, but some transformations are more equal than the others.

(Variation)

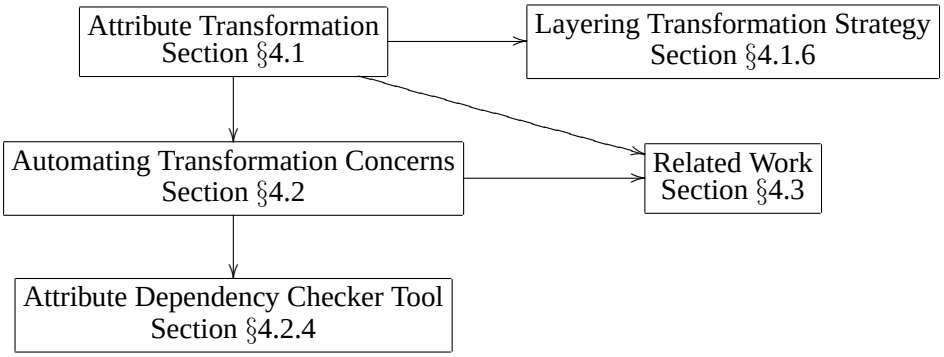
This chapter[1] addresses the implementation of modular attribute-driven transformers by exploring features that are specific to DSA modeling of mobile product-lines with attributes. The developed technology can be used, as well, with other attribute-driven transformations.

Using explicit attributes at the programming language level has recently been attracting a lot of attention with technologies, such as .NET attributes [NV02] and Java 1.5 annotations [Jay04], and also with specialized tools, such as xDoclet [xDo03][2], used with EJB [MH00].

[1] This chapter shares content with references [Cep04, CM04, CM05b, CK06].

[2] xDoclet does not work at the language level.

However, there are currently no systematic ways for transforming attribute-driven product-line specific constructs. This often results in implementations which are not very modular and hence difficult to reuse.



Section §4.1 shows how the AST could be modeled to ease attributes transformation in mobile product-lines. The addressed domain is explored to introduce a horizontal modularization of transformers. The OO language model of MIDP [J2M02b] is used in section §4.1.6 to layer the transformation strategy.

Automating attribute transformation concerns using meta-attributes is addressed in section §4.2. A specialized tool, called ADC, for checking attribute dependencies based on meta-attribute decorations, is presented in section §4.2.4.

Several generative and graph transformation techniques could be used to implement attribute-driven transformers. Related transformation and automation approaches for the techniques developed in sections §4.1 and §4.2 are discussed in section §4.3.

4.1 Attribute-Driven Transformations

The concepts discussed in this section have been implemented as part of an attribute-driven transformation framework, called **Tango**[3]. The Tango framework is designed to quickly add attribute-based domain-specific constructs to existing object-oriented (OO) languages, by reusing the existing language functionality. Tango has evolved as a generalization of the work on the **MobCon**[4] Transformation Engine (MTE), to support mobile containers presented in chapter §5. Examples from Tango will be used as necessary in this section, to explain different aspects of attribute-driven transformations. A distinction will be made between the introduced concepts, which are general and could be implemented in more than one way, and the concrete Tango's specific implementation.

[3]The name *Tango* was coined in the following way: tag → tag-go → taggo → tango.
[4]Mobile Container.

4.1.1 AST Representation

The starting point for the representation of the abstract syntax tree (AST) in attribute-driven transformers will be a GAAST-like API (§3). As the focus of this section is on attribute-driven transformers used to support mobile software product-lines, an abstraction over the AST is utilized to facilitate the transformations for this domain.

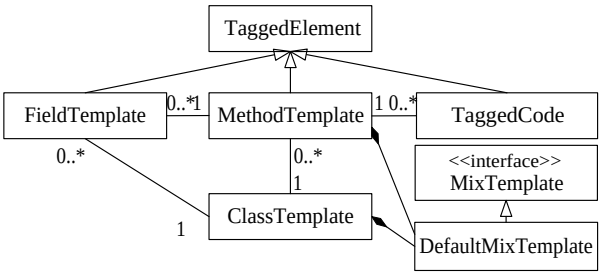


Figure 4.1: CT-AST API

Internally, the GAAST representation is organized around a class-based API called a *Class Template AST (CT-AST)*, shown in Figure 4.1. The CT-AST API wraps the original AST of the source code (§5) and contains operations that facilitate the AST manipulation of classes and methods. For example, method bodies are represented as a list of attribute decorated `TaggedCode` elements divided into three logical blocks: begin, middle, and end (Figure 4.2).

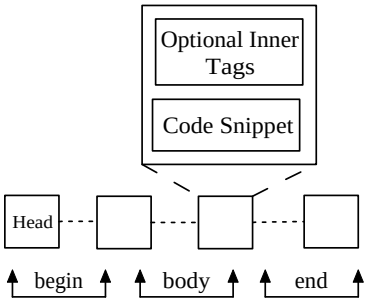


Figure 4.2: CT-AST Method Body Model

This organization helps transformers to select code elements of interest based on attributes and to insert new code elements before or after any other code block. When an existing method is parsed in code, its method body is represented as a single code element in the middle block[5].

[5] All transformers of prototype for J2ME MIDP of chapter §5 work at the method block level.

Each element of the CT-AST API derives from a `TaggegElement` and can be decorated with one or more attributes (§4.1.4). Support for merging different class templates is provided using a `DefaultMixTemplate` class (§4.1.5).

```
1 <!ELEMENT methodbody (startblock?, middleblock?, endblock?) >
2 <!ELEMENT startblock (taggedcode)+ >
3 <!ELEMENT middleblock (taggedcode)+ >
4 <!ELEMENT endblock (taggedcode)+ >
5 <!ELEMENT taggedcode (tags?, code) >
6 <!ATTLIST taggedcode name CDATA #IMPLIED >
```

Figure 4.3: Tango's Internal Model of the Method Body

Internally, Tango works with an XML [SG01] representation of the CT-AST and expects the input to be converted to XML. For example, Figure 4.3 shows how the method body is modeled in Tango as XML DTD. This enables more flexibility for processing code originating from more than one language, that conforms to the CT-AST meta-model. The internal XML representation externalizes the AST parsing outside Tango. The AST parsing can be done with any parsing or meta-programming tool[6].

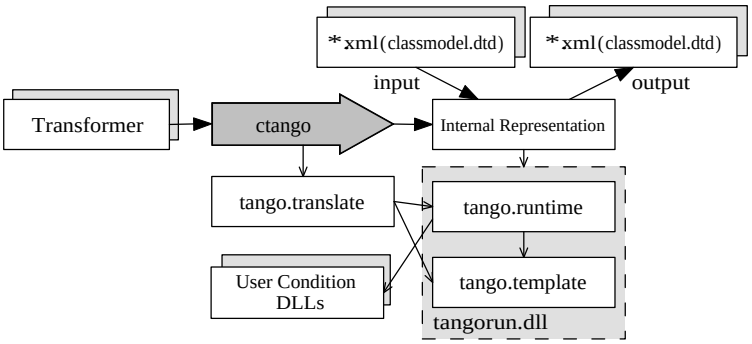


Figure 4.4: Tango Framework

The high-level organization of Tango is shown in Figure 4.4. The usage of the XML representation is implicit in Tango. Tango supports a set of CT-AST classes around XML (part of *tango.template* namespace in Figure 4.4) similar to those in Figure 4.2, that are used by the transformers. The node selection operations of Tango are internally mapped to XML queries using XPATH [XML99]. The Tango processor (ctango) invokes the Tango run-time to process the input code encoded in the Tango's XML format. The transformation functionality is specified as input to the Tango processor. This functionality is first mapped onto the internal Tango

[6]Chapter §5 explains how the real AST for J2ME MIDP is obtained.

representation in XML (*tango.translate* namespace). The Tango processor invokes the Tango run-time (the transformation engine) that maps the input code to the output code based on the transformation logic[7].

4.1.2 Class Transformations

Classes annotated with attributes serve as input units for attribute-driven transformers. Figure 4.5 shows that an attribute annotated class can be transformed in two ways. A new class can be created, or the class can be modified in place[8]. The exact transformation will depend on the semantics that attributes carry and the specific invasive technology used.

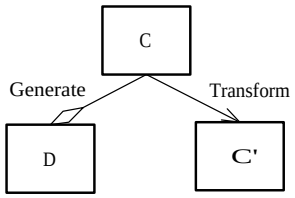


Figure 4.5: Transformation and Generation

- *Creating a new class.* A class C annotated with attributes is processed with a transformer T, giving a new class D, such that $D = T(C)$. The D class is called an adapter [GHJV95]. The original class C does not change. The adapter D contains the additional functionality introduced by T, and forwards the messages as necessary to the original class C. Both, D and C must be present in the system. Another way to implement D is based on the template pattern [GHJV95]. In this view, D is the result of the template T, where the specialization C is applied: $D = T < C >$. The class D is *generated* based on the class C.

  The implementation of T requires that the C's implementation is independent of the intended attribute semantics. This means that C should be successfully compiled ignoring the attributes. This transformation in not possible for most attribute-based DSA. To maximize automation, attributes are used to introduce changes in the code before compilation.

  The D class can either be obtained before (or during) compilation, or at run-time using reflection when supported. For example, .NET saves structural attributes into the compiled binary. At run-time, annotations can be accessed using reflection. In .NET languages, rather than using the component C directly, the implementation of D can use reflection

[7] Custom user condition DLLs are explained in section §4.1.6.

[8] More than a single new class could be created. In-place class transformation and creation of the new classes can also be combined (see equation 4.1).

over the original component C[9]. Alternatively, D can be generated before compilation using the .NET *CodeDom API* [Har03] over the component C's source code. The adapter D could also be changed dynamically resulting in some form of dynamic AOP [KLM+97][10].

- *In-place class modification.* The annotated class implementation is dependent on the semantics of the tags. C could not be compiled directly without applying some transformation (add / remove code) to it. In this case, the attribute-based transformer T gives a new class C', such that $C' = T(C)$. Only the class C' must be present afterward in the system. The class C is *transformed* into the class C'.

  In-place class modification supports invasive changes that need to be applied before compilation. These changes are needed to support attribute-based DSA, and when the original system is not designed with transformation in mind (and makes direct use of C, rather than through a new component D). In-place class transformations are usually applied to source code, because generally the class C can not be compiled directly.

The symbol Γ will be used to stand for both D and C', when it not necessary to distinguish the two cases, that is $\Gamma = T(C)$. If C's implementation supports non-invasive transformations, then Γ could be either D or C' since both are possible, otherwise it is C'. Γ is the result of the transformation process, that is, Γ is the final implementation of C. Γ's implementation is identified by the tuple $< T, C >$. The motivation to split Γ's implementation is that the same transformer T can be used for any class C decorated with attributes that drive the T transformation. This way, an entire set of classes $\{\Gamma\}$ is parameterized based on an implementation set $\{C\}$. T can be seen as an implementation template, parameterized by C. The functionality of T does not need to be repeated in every element of $\{\Gamma\}$.

Additional specialization parameters can be specified as a parameter vector Π to T, that is $\Gamma = T(\Pi, C)$. Parameters offer a convenient way to parameterize T independently of the implementation of $\{C\}$. The representation $\Gamma = T(\Pi, C)$ is very general. It can be implemented in various ways, e.g., using high-order functions [RH04], or generics (templates) [CE00], depending on the hosting language technology. The focus will be on implementing T for attribute-driven transformers, which usually apply in-place class modifications.

Implementation of attribute-based transformers is usually difficult, because more than a single attribute needs to be processed as part of a complex context. Usually an attribute-driven transformer T takes as input a set of classes $\{C\}$ decorated with several attributes $\{\tau(\pi)\}$, where π are the parameters of a single attribute τ. Thus, an attribute transformer T can be defined as a transformation function by the equation 4.1:

[9] It becomes a bottleneck if a lot of reflection code is written every-time, unless an adapter pattern [GHJV95] is used.

[10] In this case, the rest of the program should be aware of the adapter D and use it instead of using the class C directly which may not be always acceptable.

$$\{\Gamma\} = T(\{\tau(\pi)\}, \{C\}) \tag{4.1}$$

The transformer T in equation 4.1 transforms a set of input classes $\{C\}$ decorated with a set of attributes $\{\tau(\pi)\}$ into a set of output classes $\{\Gamma\}$. The attribute set $\{\tau(\pi)\}$ is the *coupling set* between T's implementation and $\{C\}$. The implementation of C needs to be accessed inside the implementation of T, in order to be able to reason about it. Only the case when C is source code will be considered in this chapter. Other transformation based on the same technology can be used for other representations of C, e.g., in the Java bytecode representation, if the possibility to access and manipulate similar entities exists in those other representations.

4.1.3 Mapping Transformation Logic to Attribute Families

Section §3.1.1 explained how attribute families can be used as a variability mechanism for product-lines. Attribute families could also help to logically organize attribute-based DSA transformations. They drive the first modularization of T in equation 4.1.

Each family of attributes can be processed by a single transformation unit specialized for that family. Formally, the entire attribute set $\{\tau(\pi)\}$ is split in a finite series of disjoint subsets ϕ_i, corresponding to each attribute family i in a domain of interest. The transformation function T in equation 4.1 can be then expressed as a composition of transformers Φ_i corresponding to the N attribute families:

$$\{\Gamma\} = \Phi_1 \circ \ldots \circ \Phi_N(\{\tau(\pi)\}, \{C\}) = \Phi_1(\phi_1, \{C\}) \circ \ldots \circ \Phi_N(\phi_N, \{C\}) \tag{4.2}$$

All individual family transformers have access to all classes passed as input to T and select the ones of interests based on their specific attributes ϕ_i. The decomposition of each transformer Φ_i could be done similarly, if needed, based on attribute sub-families. The Φ_i transformers of individual attribute families can be organized as plug-ins of a generic attribute-driven transformation framework (§5).

4.1.4 Controlling Composition Semantics with Inner Tags

A side-effect of the equation 4.2 is that it removes the global transformation state that was preserved in T (equation 4.1). There is no global way to maintain information about the overall transformation process. Results of previous computations should be explicitly passed to the next transformer in the chain. This can be done by means of additional parameters passed to each Φ_i, or by encoding the information to pass in the transformed classes that result from the input $\{C\}$. Each transformer $\Phi_i(\phi_i, \{C_i\})$ needs to re-parse the information of the previous transformer

$\{C_i\} = \Phi_{i-1}(\phi_{i-1}, \{C_{i-1}\})$. Reprocessing the AST can be avoided, if the information that needs to be passed between the transformers is also encoded as attributes. Now, each transformer has the form $\Phi_i(\phi_i, I_i, \{C_i\})$, where I_i are the additional attributes expected to be found in $\{C_i\}$ before the ϕ_i-driven transformation can be applied. The I_i attributes do not represent directly domain-level concepts as the ϕ_i do. When present, the I_i attributes only help with the transformation concerns. The output C_i' of each transformer Φ_i is decorated with a set of attributes O_i made up of a combination of the unprocessed explicit attributes, and the I_{i+1} phase attributes added during the Φ_i transformation:

$$(O_i, \{C_i'\}) = \Phi_i(\phi_i, I_i, \{C_i\}) \tag{4.3}$$

In Tango, the I_i attributes are known as *inner tags* (or attributes). Tango treats the inner tags as a natural extension of the explicit attributes, that enable a declarative transformer composition. Inner tags are used only in the inner operations of attribute-driven transformers and offer a convenient means for specifying the *coupling sets* between the composed transformers, by removing the need to reinterpret the code. Inner tags are similar to ASF+SDF [vdBHKO02] *placeholders* for saving intermediate results. In Tango, the inner tags offer a uniform model to controll custom semantics, that is integrated uniformly with the remainder of attribute-driven transformer operations. Transformers deal with the inner tags in the same way they process the explicit attributes. All Tango's basic edit operations can specify inner tags to decorate the entities that they modify.

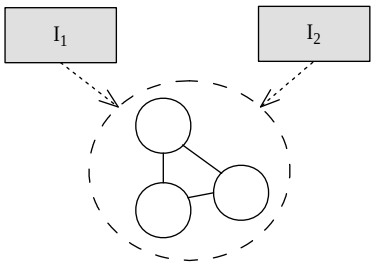


Figure 4.6: AST Node Grouping and Labeling with Inner Tags

While inner tags place transformer interaction semantics, the alternative is to reprocess the AST in each transformer, to check whether it fulfills a given condition. Using inner tags does not grow transformer coupling (it remains the same). Inner tags only make the coupling declarative, avoiding AST reprocessing. Logically, the inner tags are used to create arbitrary graph node sets [Men99], which may not correspond directly to the generalized AST graph nesting structure as shown in Figure 4.6. A group of AST nodes is decorated with same inner tags to form a single logical node unit. Inner tags also help to associate more than one label with a node (group) as in Figure 4.6, in order to select the node in more than one logical set.

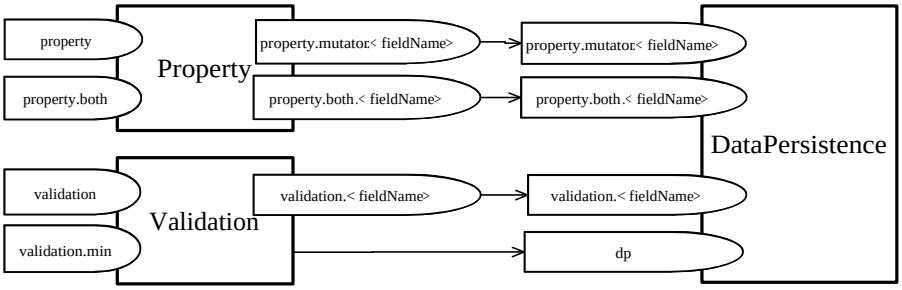


Figure 4.7: Transformer Composition Based on Tags

Consider again the attribute-driven transformation example of Figure 3.4, presented in section §3.1.1. Figure 4.7 shows a possible combination of the three attribute-driven transformers, namely, `Property`, `Validation` and `DataPersistence` for processing the code of the example in Figure 3.4. The exact combination order may change in a particular implementation. In this example, the `Property` transformer expects the input unit(s) to be decorated with attributes `property` and `property.both`. The `Property` transformer decorates its output, which is made of the added property methods, e.g., `setScore()`, and their corresponding fields, e.g., `score`, with a `property.mutator.`<`fieldName`> inner tag (among the other tags). The <`fieldName`> in Figure 4.7 stands for the actual field name, e.g., `score`. The actual value is expanded during the transformation process.

The `DataPersistence` transformer combines the output of the two previous transformers. `DataPersistence` requires that the AST, it works upon, is decorated among the others with the (inner) tags `property.both.`<`fieldName`> and `validation.`<`fieldName`> to be able to read, modify, and validate the field data inside the persistence code that it adds. `DataPersistence` treats the inner tags, e.g., `property.mutator.`<`fieldName`>, in the same way it treats explicit attributes, e.g., `dp`, and cannot distinguish between the two.

4.1.5 The Transformation Workflow

For the transformation of the equation 4.2 to be applicable in practice, a partial order over Φ_i should be found. The partial order of Φ_i depends (a) on the domain assets that Φ_i models, expressed by the attribute set ϕ_i and (b) on the details of the Φ_i implementation, expressed by the attribute set I_i. The implementation order is accidental, and the I_i will depend on the order placed upon ϕ_i. Because the transformers Φ_i model domain concerns, they can be named according to the concerns they address ($\Phi_i \rightarrow name$). The names of the Φ_i transformers can be used by the developers to define a partial order over Φ_i. The process can be partially automated by specifying for each addressed domain concern, what other concerns need to be processed

before and after it, as $\Phi_i\{before\}\{after\}$ lists. The elements of before and after lists are made of the Φ_i transformer names, or of special quantifier symbols, e.g., *all* or *any*. These local dependency relations are processed automatically to build the total Φ_i dependency graph, as shown in the example of Figure 4.8. Encoding the dependency relations locally in each transformer, rather than globally, is preferable, because the dependency relations depend on the particular implementation of the Φ_i. The local encoding of dependencies frees the developers from having to build manually the total dependency graph.

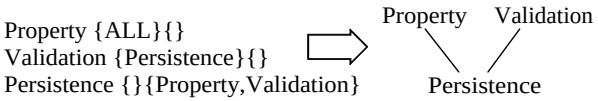


Figure 4.8: Resolving Dependencies

The dependency graph defines the transformation workflow. The developers can modify the automatically generated dependency graph manually to resolve any conflicts, on a case by case basis. A minimal workflow language is supported in Tango for this purpose. The Tango's workflow syntax is made up of (a) class variables ('$') that can point to the processed classes, (b) constructors that create empty or initialized class variables, and operations to save them back to source files, (c) an operation to assign (deep copy) the class variables ('='), and (d) two composition operators: the sequential composition (',') (which is equivalent to the function '()' operator) and the try operator ('|') that is similar to an 'if ... else' construct. A transformer in Tango, either succeeds or fails.

For example, the workflow order of the transformers of Figure 4.7 can be specified as `$CT = Property, Validation, Persistence;` which is equivalent to `$CT = Persistence(Validation(Property($CT)));`. Actually both notations are supported. For the MIDP domain concerns addressed in this book, e.g., data `Persistence`, it makes no sense to apply the transformations continuously to a class. For this reason, Tango, currently, does not define loops, or other more sophisticated workflow operations.

Once the dependency graph, based on the domain concerns, is in place, the inner tags are used inside the transformer implementations to coordinate the control flow inside the individual transformers. Sometimes *transformation adapters* need to be created to process the input, so the code has the form and the inner tags expected by an existing transformer. The equation 4.2 leads to a sequential evaluation of transformers. However, based on the domain dependency graph, some of the Φ_i transformers could be orthogonal to the others and the parallelism ('||') could be explored:

$$\{\Gamma\} = (\bigcirc, ||)\Phi_i(\phi_i, I_i, \{C_i\}), i \in 1, N \tag{4.4}$$

For example, in Figure 4.7 the input AST for the `DataPersistence` comes from two

other transformers that could work in parallel `Property` and `Validation`. The `DataPersistence` transformer does not need to reevaluate the AST. It only checks for the required attributes. In this case `DataPersistence` mixes also the output of `Property` and `Validation` transformers. In this case, `Property` and `Validation` could also be seen as adapters of the original AST, to make it suitable for further processing by the transformer `DataPersistence`. All the composition semantics are represented as attribute names (labels) in Figure 4.7.

Because of the parallelism, different versions of the same class could be present during the transformation, as in the case with the classes output by the `Property` and `Validation` transformers. Because of the orthogonality of parallel transformers, these classes need to be merged. A default merge operation that returns the union of the members of two classes, and the union of corresponding method code blocks (§4.1.1) is provided in MobCon (§5.3.1). More complicated class merging need to be coded manually inside a transformer. Tango generalizes the merging concept by allowing any combination of classes to be merged at once inside a transformer. This enables a uniform composition of the transformers, unlike other systems, e.g., JMangler [KCA04], that make a distinction between individual transformers and mergers. While Tango evaluates all transformers sequentially, threading and synchronization could be implemented automatically if needed, based on the data flow [RH04] graph on the Φ_i transformer boundaries.

4.1.6 Layering the Transformation Strategy

Up to now, only the knowledge of the domain assets is used to enable a *horizontal* modularization of the attribute-driven transformers (§4.1.3) and to define the transformation workflow (§4.1.5). It is also possible to explore the features of the supported language meta-model to enable a *vertical* modularization of the transformers. Tango uses a common OO meta-model based on classes, fields, methods, and method blocks. This hardwired OO meta-model is enough to support the transformations that mobile containers introduce in an OO mobile product-line (§5). The specific features of the language meta-model have been used in the past to create customized solutions for specific sets of languages. For example, denotational semantics [NN99, Fin96] explore the features of procedural languages, in order to provide a specialized model for expressing and verifying language semantics, which applies better to procedural languages than the generic operational, or axiomatic semantics [NN99].

The structural information of the hardwired common OO meta-model can be used to structure the transformation *strategy* [Vis01b] in layers, corresponding to the nesting of the structural elements in the OO meta-model. Layering enables reasoning at different levels of abstraction about the transformation strategy, and enhances the reuse of the low-layer modules in more than one transformer. The attribute-transformers in Tango are organized in several hierarchical layers, shown in Figure 4.9. The *transformers workflow layer* was already discussed in section §4.1.5,

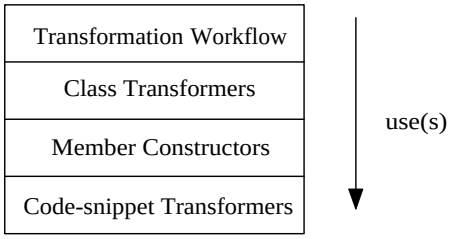


Figure 4.9: Tango Framework Layers

the *class transformers layer* (T_c) defines operations applied to classes, the *member constructors layer* (T_m) defines operations applied to fields and methods, whereas the *code snippets layer* (T_s) helps to manipulate the code templates that are used inside the method blocks. The transformer Φ_i of equation 4.3 can be written as:

$$\Phi_i(\phi_i, I_i, \{C\}) = T_{c_i}(\{C\}, T_{m_i}(\{C_{members}\}, T_{s_i}(\{members_{code\_blocks}\}))) \tag{4.5}$$

The information about ϕ_i and I_i is distributed as needed between the transformers of each layer. The classes from $\{\mathrm{C}\}$ to be processed in T_{ci} are again selected based on the ϕ_i and the I_i attributes, as in Φ_i. Layering is possible in Tango because the OO meta-model is hardwired, so the nesting of the structural elements is completely known. An open meta-model cannot be layered clearly with declarative constructs specialized for its elements.

Tango enforces layering using special syntax. Every attribute-driven transformer implemented in Tango has to be modularized to conform to the layering of Figure 4.9. Each layer uses the elements of the successive lower layer, but cannot create elements of the upper layer. For example, class templates are only created in the workflow layer. The class transformer layer can only modify the class templates, but cannot create new ones. To find out which class templates take part into a transformation, only the workflow layer needs to be examined. Similarly, member constructors are used in the class transformers layer only. Finally, code snippets are used only by the member constructors.

To understand how vertical layering is implemented beyond the workflow level, consider how the *Property* attribute family transformer for the *GameScore* example of Figure 3.4 can be implemented. Figure 4.10 shows the complete implementation of the *Property* transformer (class transformers level) in Tango. Each **class level transformer** operates in one or more class template variables, passed to it as arguments by the workflow layer, e.g., the ˜ct at line 1 of the code shown in Figure 4.10. The class level transformers cannot create new class templates. If new class templates need to be created, this has to be done at the workflow level.

A class level transformer in Tango has two main parts: an optional *precondition* part (lines

```
1  transformer Property(~ct) {
2  precondition {
3      if(not check(~ct, tags(["property"])))
4          noapply;
5  }
6  action {
7      $fields = select(~ct, FIELDS, tags(["property.*"]);
8      if(check($fields, empty())) error;
9      iterator($field in $fields) {
10         if(check($field, tags(["property.accessor"])
11             or tags(["property.both"])))
12             add(~ct, METHOD, GetMethod($field),
13                 [tag(<"property.accessor", $field.name>)]);
14         else if(check($field, tags(["property.mutator"])
15             or tags(["property.both"])))
16             add(~ct, METHOD, SetMethod($field),
17                 [tag(<"property.mutator.", $field.name>)]);
18     }
19 }
20 return ~ct; // optional the first argument is returned
21 }
```

Figure 4.10: The Property Class Transformer Implementation

2 - 5) and an *action* part (lines 6 - 19) (Figure 4.10). The optional precondition section is used to apply some quick checks on the input, in order to decide whether the transformation can be applied or not. Only non-edit operations are allowed in the precondition section. Tango preconditions are *optimistic*. The action part can apply more specific checks, and the transformer may still fail as a consequence of a failed condition in the action part, even thought the preconditions succeeded. The precondition section is only aimed at being able to quickly determine the attribute family, or the main inner tag families, supported by the class transformer. Preconditions are not performance optimizations rather, they give a clue about the attribute family semantics addressed by the transformer. Preconditions remove the need to consult the more complex action section, when only a high-level view inside the functionality of a class transformer is needed. The *noapply* operator (line 4) tells the Tango workflow that this transformer cannot be applied. If a transformer fails to apply, the control is returned to the workflow. Theoretically, a precondition and a post-condition can be defined for any graph rewriting operation [Men99], but in practice, it is cumbersome to enumerate them for each operation (post-conditions can be written as preconditions [Men99]), so only important checks are applied in the preconditions part in Tango.

The AST editing operations are allowed only in the *action* part. The action part (lines 6 - 19) will modify the input class template, to add getter and setter methods for all fields decorated with an attribute belonging the *property* family. The fields are selected in line 7 and stored as a list in the variable `$fields`. The generic *select* operation is used to filter a set of nodes of

the same type that fulfill a given condition. Only predefined types of the supported meta-model, e.g., `FIELDS, METHODS`, can be used. The third argument of *select* is a *condition*. Several predefined conditions are supported:

- 'tag' - filters AST nodes based on attributes,
- 'name' - filters AST nodes based on names,
- 'empty' - checks whether a node list is empty,
- 'count' - checks the number of list items.

For conditions that expect string arguments, regular expressions are supported. Users can define additional custom conditions by implementing a required interface `IUserCondition` and packing the condition logic as a separate DLL placed in a predefined sub-folder. Every time the Tango's run-time finds an unknown condition name, the run-time searches whether a custom condition DLL with the same name can be found. If found, the DLL is loaded and the `IUserCondition` interface is used to invoke a generic `Filter` method using reflection. Any argument list for the condition is passed to the `Filter` method. The `Filter` method returns a filtered list. If no such DLL is found, or an exception occurs when the DLL is invoked, an error is reported.

The *check* operation in line 8 is similar to *select*. It applies all the conditions passed to it as the first argument and returns a *Boolean* value indicating whether conditions have succeeded or not. The individual conditions can be combined using Boolean operators: `and, or, not`. The *check* and *select* operations pass the first argument implicitly to all conditions. This makes it easier to understand what a *check* or *select* statement does, given that all conditions work on the first argument of *check* or *select*[11], and is similar to the concept of *conventional interfaces* in [ASS96][12].

The iterations over the meta-model AST are divided between the *check*, *select*, and *iterator*. The *check* and *select* operators do implicit iterations over *finite* lists of elements (via the conditions). The *iterator* operation in line 9 is used to apply an operation to all the elements of a list. The *iterator* and *select* could be a single operation, however, it makes sense to separate these operations in cases when lists, other than those returned by *select*, are processed.

The *add* operation (lines 12 and 16) is an example of an edit operation. It adds a meta-element to the class template given as its first argument. The `GetMethod` and `SetMethod`

[11] At the cost of a slightly slower implementation, since each condition has to re-iterate through the input list. Set operations are used to mix the results of different conditions: `and` corresponding to *intersection*, `or` corresponding to *union*, and `not` corresponding to set *difference*. Given that `not` operation is easier to implement during iteration, the `Filter` method of the `IUserCondition` interface, accepts a special boolean flag indicating the `not` condition.

[12] `http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-15.html#%25_sec_2.2.3`

are member constructors. All supported edit operations in the class transformers layer, i.e., *add*, *delete* and *modify*, work upon class templates and use member constructors of the next lower layer. In the example of Figure 4.10, the *add* operation also appends an inner tag to the newly added element: `[tag(<"property.accessor", $field.name>)]`. The tag is created by the explicit attribute constructor *tag*, combining the field tag type and the field name, by using the string concatenation operation `<...>`. This inner tag can be used later in a *select*, or *check* statement of another transformer in the same way as an explicit attribute.

The *add* and *select* operations use a common pattern. The type of processed meta-elements is given explicitly. This improves the readability of the code (in the case of *add*, the type could have been deduced from the member constructor type). An alternative would be to have distinct operation names for each meta-element type, e.g., *addMethod*. This would then require to maintain the Tango parser if its supported language meta-model is customized. The modified class template is returned in line 20.

The **member constructors** layer defines the actual meta-model member implementations that are used in the class transformers layer. Class templates cannot be passed as arguments to the member constructors. Currently, Tango defines member constructors for fields, methods and tags. As explained in section §4.1.1, method bodies are represented as a list of attribute decorated code snippets divided into three logical blocks: begin, middle, and end (Figure 4.2). This representation enables to quickly add or remove blocks of code to a method, without dealing directly with the AST node composition. All the changes that the attribute-based DSA for mobile containers introduce, modify the method internals at the block level only.

```
1 method ToRecord($fieldArray) {
2   methodName(makeMethodName(["toRecord"]));
3   methodReturn(ByteArray());
4   addBody(StoreFields($fieldArray));
5 }
```

Figure 4.11: Method Constructor Example

Figure 4.11 shows a method member constructor `ToRecord`, that is used in the persistence transformer. The constructor expects an array of fields that will be serialized in a byte array, and returns the implementation of a method `toRecord` that takes care of the serialization. The method name for the newly created method is set in line 2, and its return type is set in line 4. The example specifies the method return type by invoking a code-snippet constructor *ByteArray* (line 3). A code block is added in the middle block of the method, by using the `addBody` statement in line 4. The `addBody` invokes another code snippet constructor namely, *StoreFields*, to obtain the code that implements the field serialization. The method body is produced by the code snippet constructor *StoreFields* (Figure 4.12). The *addBody* operation adds this code snippet at the end of the middle block list. No direct code is manipulated in this layer.

Instead the code-snippet constructors, e.g., *ByteArray* and *StoreFields*, are called.

4.1.7 Code-Snippet Templates

Section §2.3.2 explained that invasive techniques lead to writing more code because of the meta-level indirection of the programming model. The situation can be improved by using code templates [Voe03a, Aßm03]. Attribute-based DSA modify the components they decorate. Most of the changes happen inside the method boundaries, or by adding new methods. Code templates can help (a) to reduce coding efforts inside the member constructors layer and (b) to isolate the rest of an attribute-driven transformer from the language specific details.

In Tango, the transformation abstractions are isolated from the remainder of the transformer implementation via the concept of a *code snippet*[13], which is similar to Stratego's [Vis01b] *concrete syntax*. A code snippet is a source code node that Tango treats as a string. It is the only part of code which depends on the concrete syntax of programming language, where the code being transformed is written. The rest of a transformer's implementation in Tango works with the hardwired OO model. Code snippets enable the representation of parameterized clusters of the source code graph in the remainder of the transformer implementation, without having to deal with details of the AST nodes.

Tango requires that a particular implementation of the code-snippet templates has a way to replace parameters inside the code snippet. An example of a template code snippet, that makes use of the Apache Velocity [Vel03] script language, is shown in Figure 4.12. The code generates a possible implementation for the body of the *toRecord* method shown in Figure 3.5, and that was used in the example of Figure 4.11.

```
1  code StoreFields ($fieldArray) language (Velocity) {
2      ByteArrayOutputStream baos = new ByteArrayOutputStream();
3      DataOutputStream outputStream = new DataOutputStream(baos);
4      #foreach($field in $fieldArray)
5      outputStream.$dp_write($field.type)
6          (this.$Tango.makeMethodName(["get", $field.name]()));
7      #end
8      return baos.toByteArray();
9      #macro(dp_write, $type)
10         if($type == string) UTF
11         if($type == int) Int
12     #end
13 }
```

Figure 4.12: Code Snippet Example

The code of Figure 4.12 is a Velocity string template specialized using the values in the

[13] This term is borrowed from .NET CodeDom API [Har03].

`fieldArray` argument at line 1. A `ByteArrayOutputStream` is created and used to initialize a `DataOutputStream`, so that the fields can be output formatted according to their type. The specialization is done in lines 4 - 7, which generate code to store each field of the class, in line 5. A special macro named `dp_write`, defined in lines 9 - 12, is deployed to generate the right name of the `DataOutputStream` function, depending on the variable type. Only *Integers* and *Strings* are shown in this example. Strings data are saved using a portable UTF[14] encoding.

4.1.8 Termination

The attribute-based transformation approach presented in this section can be considered as an instance of a *graph rewriting system* (GRS) [DJ90, Nag96, ET96, BS99, SWZ95, Sch94b, MA99, Aßm00, Plu01]. Attributes form a finite vocabulary (*signature*) used to transform a finite number of variables, represented as classes of an OO language. The approach can be seen as a form of graph labeling [Plu95, Men99], where labels form a *hypergraph* [Plu95] made of attribute family trees. The explicit, and a part of the inner attributes, can be simplified to source code as a ground form. Some of the inner attributes, such as those used to support traceability (§4.1.9), are *irreducible* and are preserved in the final normalized form (source code).

In terms of [DJ90], the approach is not convergent, as the order of transformation matters. That is, not all possible combinations of sequences of rules lead to the same normal form. The ordering or rules, however, ensures *termination* [Gra96]. Termination is ensured for a relation if its transitive closure forms a well-founded ordering (definition 12 in [DJ90]). The approach presented here has such a well-founded termination ordering. In the top level, the order of the transformation is ensured by the dependencies between the domain assets, forming a stratified [CGT89, Aßm00] structure for applying transformers. The workflow graph is finite and contains no endless substitution chains. Explicit loops are not allowed in the workflow syntax, whereas recursive, or circular transformer dependencies cannot appear, because the vertical transformation order is made up of a well-defined finite number of three hierarchical [Ohl02] expanding levels: classes, methods, and method blocks. Rules of each level cannot refer to the parent, or the sibling rules, which prevents the circular dependencies[15].

In [Aßm94, Aßm96] termination criteria are defined for special forms of GRSs, based on labeled graphs ($\Sigma - Graphs$). A series of derivations is finite, if it contains a finite number of finite derivations steps. Termination is warranted for the GRSs that contain only graph edge accumulation, called edge-addition rewrite systems (EARS). Furthermore, termination criteria are also specified for exhaustive GRSs (XGRS), made up of additive GRSs (AGRS) and subtractive GRSs (SGRS). The attribute-driven transformation presented here could contain subtractive operations, however, in practice changes are always additive (AGRS) and complement the original

[14] Unicode Transformation Formats. `http://www.unicode.org/`

[15] Infinite loops in the lowest code-snippet layer indicate programming errors.

code of the annotated components.

The termination of hypergraph transformations in GRSs is discussed in [Plu95], where a rule is defined as a combination of a left-side and a right-side graph morphisms. A forward closure is defined as a minimal set of successive derivation steps. Termination is warranted when the GRS consumes a rule in each step, and does not admit to an infinite forward closure. The attribute-driven transformations of this section, consume at least one attribute in each step, and because of the finite ordering, the derivation sequence is finite.

4.1.9 Transformation Traceability

Traceability is important for generative frameworks because in Tango the code snippets replacements are not fully statically checked at the time of the string replacement. Rather, this action is postponed until the compilation time, where any possible errors will appear. These errors need to be connected back with the input code locations. Decoration with inner tags, in order to enable traceability logging, helps to deal with error tracing[16].

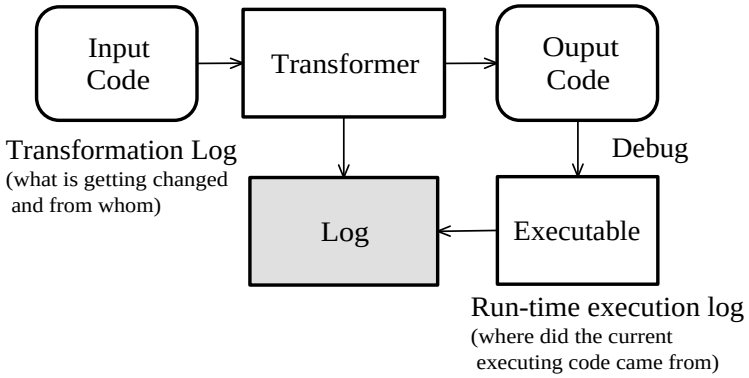


Figure 4.13: Transformation and Execution Log

Traceability is integrated in the Tango framework (Figure 4.13), and is implemented with the help of the inner attributes. Tango distinguishes between (a) transformation time log and (b) run-time execution log. The transformation log gives feedback about the transformation process. The execution log helps to trace the origin of the executed code. Traceability can be turned on and off for pieces of source code of interest by using a special explicit attribute `log`, directly in source code. When present, this attribute instructs the framework to automatically add special log calls to all methods that are transformed by every Tango transformer. The log calls are

[16] Yacc [LMB92] solves this issue with ANSI C pragma directives that overwrite the line numbers in the outputted source file. This technique does not work with Java and .NET compilers. Furthermore, Tango transformers can carry out complex transformations and line number based traceability may not work in all cases.

decorated with special attributes, that contain the transformer names which have edited a given method. When output code is executed, the log trace statements are printed on the console, enabling the display of the exact transformation history for each executing method (an example is given in Figure 5.18). This centralized tracing capability helps to debug the transformation-related side-effects.

As discussed in chapter §3, GAAST languages support the preservation of the MDA PIM architecture into the PSM, when the PSM is source code, shifting the model transformation to the language level. The structured attribute-driven transformations of this chapter further preserve the traceability of the PIM, after the PSM transformation. Inner attributes (§4.1.4) enable the association of traceability semantics with the transformed elements. The Tango framework preserves the inner attributes in the outputted code. This approach enables tracing back the origin of the code and depending on the design of the transformers[17] and the redundancy of the used inner attributes, it could also used to support full round-trip engineering. The preserved inner attributes do not influence the performance of the end system. The representation of the inner attributes could also be combined with specific language support for attributes, as in the case of Java 1.5 (§3.6.2), to directly support traceability at run-time.

4.2 Automating Attribute Transformation Concerns

The transformation process for interpreting the attribute-based DSA was modularized in section §4.1 based on the domain assets. Several transformation related issues, e.g., checking for the right usage of attributes, or validating the AST to conform to the decorated attributes, must still be implemented separately in each transformer. There are many repetitive transformation concerns that crosscut more than one attribute-driven transformation. The repetite transformation concerns can be factored out from the individual Φ_i transformers, and implemented as generic attribute-driven operations $\Delta_j(\{\tau(\pi)\},\{C\})$. The equation 4.4 becomes 4.6, with the Φ_i-s simplified by the Δ_j-s.

$$\{\Gamma\} = (\bigcirc, ||)[\Delta_j(\{\tau(\pi)\},\{C\}), \Phi_i(\phi_i, I_i, \{C_i\})] \tag{4.6}$$

The transformation cross-cutting concerns Δ_j are candidates to be handled automatically by an attribute-driven transformation framework. For example, logging (§4.1.9) is a cross-cutting concern handled automatically by the Tango framework as part of the traceability. This section demonstrates a generic technique that can help to factor out declaratively the cross-cutting functionality Δ_j, outside of the individual Φ_i transformers. To illustrate the concept, two examples of cross-cutting attribute transformation concerns are considered next.

[17] For example, MobCon (§5) transformers are additive. They never remove the original user code from the transformed components, only append to it. This ensures complete traceability when combined with the preservation of the inner attributes.

I) Attribute dependencies. In section §2.1.5, an example of how a web service can be modeled with DSA was given. Figure 2.4 is repeated here for convenience as Figure 4.14. Section §3.1 illustrated how the same web service DSA can be modeled with attributes in Figure 3.2, repeated here as Figure 4.15.

```
1 webservice TravelAgent {
2   ...
3   webmethod GetHotels(){...}
4   ...
5 }
```

Figure 4.14: A Domain-specific Extension to Implement Web Services (Figure 2.4)

```
1 [WebService]
2 class TravelAgent {
3   ...
4   [WebMethod]
5   public void GetHotels(){...}
6   ...
7 }
```

Figure 4.15: A Web Service Class with two Inter-depended Attributes (Figure 3.2)

When a DSL (or EDSL) is used, an explicit grammar rule, such as, `webservice := webmethod+`, explicitly defines the context relation between `webservice` and `webmethod` in Figure 4.14. A web method will appear only inside a web service and vice-versa, a web service will contain web methods. This grammar rule is automatically enforced by the parser.

When attribute-based DSA are used, there is no default way how such a generic dependency constrain can be declared and enforced. It remains the responsibility of the corresponding attribute transformer to validate the dependency constrain. Depending on whether the class or its methods are considered, there are two constrains that need to be enforced: (a) public methods of a class decorated with `[WebService]` should be decorated with the `[WebMethod]` attribute[18], (b) any method decorated with a `[WebMethod]` attribute should be declared within a class decorated with the `[WebService]` attribute. That is, the two attributes are inter-dependent. The dependency relation needs not to be symmetric, e.g., a `[Validate]` attribute may require another `[Validate.MaxValue]` attribute, whereas the `[Validate.MaxValue]` attribute may also be used alone.

Checking attribute dependencies is needed in any attribute family transformer, in order to make sure, for example, that suitable sub-attributes have been used. The code required to enforce dependencies between attributes, stating, for example, that a certain attribute is present in the program hierarchy before another attribute can be used, is a cross-cutting concern (CCC) that is repeated in every transformer.

II) Virtual instances and `this` keyword usage. This example is motivated by the im-

[18] For simplicity, it is assumed that all public methods need to be decorated. The actual implementation explained in section §4.2.4 removes this restriction by enabling user-defined filters for specifying which code elements, in this case methods, will be checked and which will be not.

plementation restrictions of the EJB [MH00] programming model. The EJB specification states, among other restrictions, that components whose instances are managed as *virtual instances* [VSW02] should not pass *this* as a parameter or as method return value. The underlying rationale is that it makes no sense to return a direct pointer to an object that will be reused with a different internal state, at a later point of time, by the container. While the EJB implementations up to version 2.1 do not rely on attributes, Java 1.5 annotations will be used in EJB 3.0 [EJB04] instead of the marking interfaces (§3.2.1).

```
1 [VirtualInstance]
2 class C {
3   ...
4   [InitInstance]
5   public C initialize(Id id){...}
6   ...
7 }
```

Figure 4.16: A Class that Requires Virtual Instance Support

The imaginary example shown in Figure 4.16 supposes that the lifetime of an instance of a class `C` is going to be managed by the container, only when the class declaration is decorated with the attribute `[VirtualInstance]`. For a class `C` tagged with the attribute `[VirtualInstance]`, the restriction about *this* must hold. The method `initialize()` of class `C` is invoked by the container when a virtual instance needs to be initialized. This method is identified by annotating it with an `[InitInstance]` attribute to distinguish it for later processing.

The transformer of the `[VirtualInstance]` attribute needs to check the *no-this* restriction for the methods that the class contains. The same check could be repeated in the member constructor transformer for the `initialize()` method, which processes the `[InitInstance]` attribute. Transformers for other domain assets, e.g., database connection pooling transformers, may also need to check the *no-this* restriction. The same validation code needs to be repeated in different transformers for the AST blocks inside a method.

4.2.1 Expressing Cross-Cutting Concerns with Meta-Attributes

In a GAAST language (§3.4), all entities can be decorated with attributes. *Meta-attributes* are attributes used to decorate other attributes. Meta-attributes can be used to associate arbitrary semantics with attributes that can be checked for, or enforced automatically. Meta-attributes do not decorate attribute instantiation[19]. Rather meta-attributes decorate attribute definitions. When an attribute $\tau(\pi)$ is defined, its definition is decorated with meta-attributes $\{\mu(\mu_\pi)\}$, where μ_π are parameters used to specialize the meta-attribute instantiation. Meta-attributes provide a

[19] Attribute parameters (§3.1.2) are used to express the variability of the attribute instantiation.

native mechanism to declaratively model the transformation cross-cutting concerns in GAAST languages. Meta-attributes help to factor the code that handles the cross-cutting concerns out of individual transformers, in generic concern processing tools, which can be made part of an attribute-driven transformation framework. The generic Δ_j transformers, working over meta-attributes $\{\mu(\mu_\pi)\}$, can be expressed as in equation 4.7. The $\{\mu(\mu_\pi)\}$ define semantic constrains (or operations) that must hold over $\{\tau(\pi)\}$, when $\{\tau(\pi)\}$ are applied to $\{C\}$.

$$\Delta_j \rightarrow \Delta_j([\{\mu(\mu_\pi)\}, \{\tau(\pi)\}], \{C\}) \tag{4.7}$$

Using custom meta-attributes in .NET [Pro02] is similar to using the predefined `[System.AttributeUsageAttribute]`. This special attribute is used in .NET to decorate the definition of a custom attribute, providing information about the lexical scope in which the attribute at hand can be used. Based on the usage attributes, every time a custom attribute is encountered in a program, the compiler can check whether the attribute is being used in the right lexical context, and report an error when this is not the case. The idea underlying `[System.AttributeUsageAttribute]` can be adopted to introduce new custom checks by using custom meta-attributes.

```
1  [AttributeUsage(AttributeTargets.Class)]
2  class NoThis : System.Attribute { ... }
3
4  [NoThis]
5  [AttributeUsage(AttributeTargets.Class)]
6  class VirtualInstance : System.Attribute { ... }
7
8  [NoThis]
9  [AttributeUsage(AttributeTargets.Method)]
10 class InitInstance : System.Attribute { ... }
```

Figure 4.17: Modeling `[NoThis]` Constraint as a Meta-Attribute

Consider for example, the *no-this* restriction (§4.2) modeled as a `[NoThis]` meta-attribute. The `[NoThis]` meta-attribute is used to decorate the definition of all attributes that need to check, or enforce, the *no-this* constrain, as shown in the .NET C# example of Figure 4.17. A cross-cutting attribute checker tool, that enforces the *no-this* restriction, needs to check each attribute of a code entity, whether the attribute definition[20] contains a `[NoThis]` (meta-) attribute. If the condition is fulfilled, then the methods decorated with the corresponding attribute need to be checked. The cross-cutting concern checker tools operate outside the Φ_i transformers, reducing the amount of code that needs to be implemented and debugged in each individual Φ_i transformer.

[20] Whether the attributes of the attribute itself.

The *attribute dependency* example (§4.2) is more complex, as it requires a bigger context to be handled, when the attribute dependencies are resolved. Ideally, it is preferable to declare the attribute dependencies similarly to grammar rules. Meta-attributes enable expressing attribute dependencies declaratively in GAAST languages. The rest of this section explores in detail how attribute dependencies can be modeled as meta-attributes and how they can be enforced automatically. The presented implementation is based on .NET [Pro02]. The details of the attribute processing in .NET were discussed in section §3.4.1.

Using meta-attributes frees the developers that write attribute-driven transformers from coding repeated concerns, by centralizing the way cross-cutting concerns are processed. The developers only declare constrains, e.g., dependencies, without taking explicitly care of how the corresponding concerns are resolved and enforced. There are of course many ways (§4.3) to declare and enforce architectural dependencies [Min98]. Using meta-attributes to decorate other attributes is a natural way for GAAST-like languages, because meta-attributes could be directly processed by the attribute-driven transformers presented in this chapter.

4.2.2 The Attribute Dependency Model

The attribute dependency model presented in this section distinguishes between (a) *required* dependencies, stating that a given attribute requires another one in order to be used, and (b) *disallowed* dependencies, stating that a given attribute cannot be used, if another attribute is present. Furthermore, children nodes in a program's structural hierarchy can declare dependency constrains on parents of any level, and vice-versa. An attribute of a certain program element instance may require that certain attributes are present in the set of the attributes of the structural children of its program element. For example, an attribute of a $Class$ may require that a certain attribute be present in the annotation of the class' $Methods$. The attribute dependency model generalizes these notions to any depth of the structural tree.

A restriction in this model is that the attributes of a program element instance cannot place any constraint on the attributes of sibling instances. For example, the attributes that a $Field$ instance is decorated with, cannot imply anything about the attributes of $Method$ instances, or attributes of other $Field$ instances. In the prototype for the J2ME MIDP (§5) sibling dependences are not used. However, the attributes of a program element instance can place constrains on the other attributes of the same instance. For example, a method attribute a_{m1} of a method m may require that another attribute a_{m2} to be present for m. A more generic dependency model could be created (to include siblings constrains), but the more detailed semantics add nothing new to the meta-attribute concepts discussed.

The semantics of the $disallowed$ relation on the structural AST elements and instances can be specified similarly and will not be repeated.

4.2.3 The `[DependencyAttribute]` Class

.NET custom attributes are classes derived from the class `System.Attribute` (§3.4.1). Custom attributes may have arguments specified either as constructor parameters (unnamed arguments), or as properties of the attribute class, which generate *getter* and *setter* methods in C# (named arguments). Attribute classes may also contain methods and state (instance fields), as every other .NET class does. Using properties to specify attribute arguments is more flexible than using constructors, because .NET does not support complex types to be passed as parameters to the constructors[21].

To handle the attribute dependencies, a new meta-attribute named `DependencyAttribute`[22] is defined as a custom attribute class. The `DependencyAttribute` contains one `Required*` and one `Disallowed*` property for every program element type, for which the attribute dependency checking is supported (`Assembly`[23], `Class`, `Method`), as shown in Figure 4.18. Given that the number of the node types in a program's structural tree is limited, it makes sense to enumerate such operations. This makes the code easier to understand compared to having a single dependency property for all meta-element types.

```
1  [AttributeUsage(AttributeTargets.Class)]
2  public class DependencyAttribute : System.Attribute {
3    ...
4    public DependencyAttribute() {...}
5    public Type[] RequiredAssemblyAttributes {...}
6    public Type[] DisallowedAssemblyAttributes {...}
7    public Type[] RequiredClassAttributes {...}
8    public Type[] DisallowedClassAttributes      {...}
9    public Type[] RequiredMethodAttributes {...}
10   public Type[] DisallowedMethodAttributes {...}
11 }
```

Figure 4.18: The Dependency Attribute

Adding support for object `Fields` to this model is trivial (§4.2.4). Readers familiar with .NET may note that `namespaces`[24] were skipped from the list of program elements above. The reason is that a `namespace` is a logical, rather than a physical concept, so even though theoretically a `namespace` can be decorated with attributes, practically there is not a single physical place where to store the attributes, given that a `namespace` may extend in many

[21] Only basic constant types and `System.Type` can be used. `System.Object` is also listed in the documentation, because it is the parent of simple types and of `System.Type`. This does not mean that arbitrary objects can be passed as constructor parameters.

[22] When used in code the suffix `Attribute` may be omitted from the name of an attribute class.

[23] A .NET `Assembly` maps roughly to a Java JAR file. The `Assembly` attributes map roughly to custom JAR manifest entries.

[24] A .NET `namespace` maps roughly to a Java `package`.

modules and assemblies[25].

The current .NET implementation seems to restrict the complexity of the validation logic that could be implemented inside the property methods of a custom attribute. The .NET documentation is not clear about whether that code is ever activated. Furthermore, if the code inside a property is more than a simple assignment, that property may be not included in the attribute class without any warning from the C# compiler. For this reason, the code of the `DependencyAttribute` properties must be kept as simple as possible, and checks that could normally be done by the property method code, e.g., that the attributes passed as a parameter to a property have the right `[System.AttributeUsageAttribute]` target[26], should be postponed until the actual dependency checking is performed.

The `DependencyAttribute` only stores the required / disallowed attribute arrays (lists), and contains only code for printing these arrays as strings, needed for error and log reporting. It does not contain any code to interpret the dependencies and its implementation has no other module dependencies. As a consequence, the `DependencyAttribute` is independent of any particular dependency checker implementation and can be distributed and used alone to decorate attribute libraries.

```
1 [Dependency(RequiredMethodAttributes(new Type[]{typeof(WebMethod)})]
2 [AttributeUsage(AttributeTargets.Class)]
3 class WebService : System.Attribute { ... }
4
5 [Dependency(RequiredClassAttributes(new Type[]{typeof(WebService)})]
6 [AttributeUsage(AttributeTargets.Method)]
7 class WebMethod : System.Attribute { ... }
```

Figure 4.19: Using the Dependency Attribute

Figure 4.19 shows how the `DependencyAttribute` is used in code to express the dependency semantics of the attributes `[WebService]` and `[WebMethod]`, for the example of Figure 4.15. The C# '`typeof`' operator is used to obtain an instance of the class type of each attribute. Class types fall into the category of types supported by .NET as attribute property arguments. The dependency lists are saved as C# arrays inside the two instances of the `DependencyAttribute` and will be preserved from .NET as part of the IL binary meta-data.

[25] This is the reason why .NET does not list `namespace` as an entry in the `AttributeTargets` enumeration.

[26] E.g., that `AttributeTargets.Method` is present in the declaration of an attribute included in the list `RequiredMethodAttributes`.

4.2.4 The Attribute Dependency Checker (ADC) Tool

The Attribute Dependency Checker (ADC) tool is implemented as a *post-processor* using the .NET `Reflection API`, exploring the reflective capabilities of GAAST languages (§3.4). After the code is compiled and linked, the ADC tool can be run over the IL binaries, to detect the possible attribute dependency errors. Alternatively, ADC could be implemented as a *pre-processor* tool to be run before the source is compiled, using the `CodeDom API`[27].

Figure 4.20 shows the architecture of the ADC tool. Almost all the functionality of the dependency checker is found in the abstract class `AttributeDependencyChecker`. This class uses several other helper classes and interfaces (a) to filter the processed elements (`IDependencyFilter`), (b) to log information about the progress of the checking process (`ICheckLogger`), and (c) to report errors (`ErrorReport`). The `IContextMap` class encapsulates the meta-model structure in a single place using a special internal coding. The ADC tool can be invoked from the command-line, or programmatically. For illustration, Figure 4.21 shows how the ADC library can be used programmatically to check the attribute dependency constrains for all elements of a given .NET assembly (line 1).

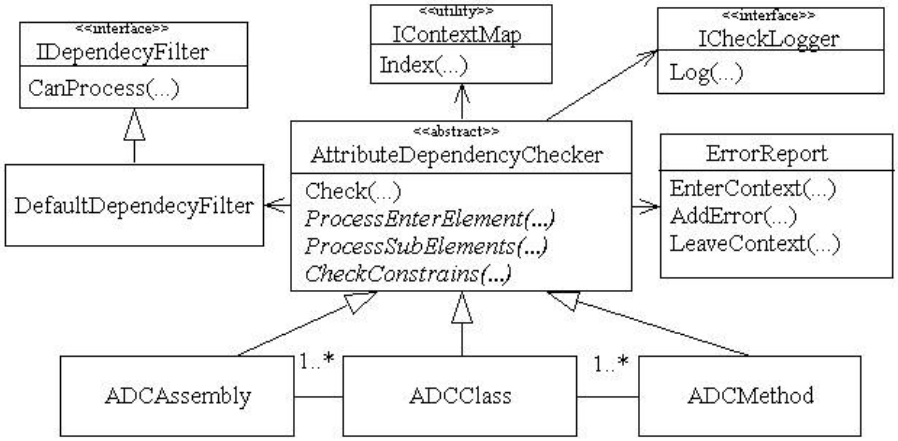


Figure 4.20: The Run-time Attribute Dependency Checker Structure

In order to implement the semantics of the dependency attribute, initially the attribute dependency sets need to be built for every structural element, by processing the element and all its structural children. After the dependency sets are constructed, the dependency constrains of the element can be checked. That is, a post-order transversal of the structural tree is required. A boolean flag in `AttributeDependencyChecker` controls whether the inherited attributes of the structural elements are processed or not. The actions performed during a call to the

[27] A third-party implementation of `ICodeParser` for C# can be found in [Zde02].

```
1 Assembly a = ...; // obtain an assembly
2 ADCAssembly c = new ADCAssembly();
3 c.Filter = ...;    // set filter
4 c.Logger = ...;    // set logger
5 c.Check(a);        // check the assembly
6 if(c.errors.HasWarnings())
7 { // process: c.errors.GetWarnings() ... }
8 if(c.errors.HasErrors())
9 { // process: c.errors.GetErrors() ... }
```

Figure 4.21: Using the Run-time Attribute Dependency Checker in Code

`Check(t)` method (line 5 in Figure 4.21), where `t` is the current program element whose attribute dependencies are being checked for, are illustrated in Figure 4.22.

First, the filter object is used to check whether the element at hand should be processed (step 2 in Figure 4.22). Filters can be used to put arbitrary constraints on the elements that will be processed, e.g., by using pattern matching on names. The `DefaultDependencyFilter` processes all the elements. The ADC command-line tool uses a customized filter called `ClassDependencyFilter` derived from `DefaultDependencyFilter` that can restrict checking to a subset of classes, whose names are given in the command-line. More sophisticated filters can be written and used programmatically (line 3 in Figure 4.21). Filters can also be used to implement profiling by keeping track of various counters; e.g., `ClassDependencyFilter` counts the number of classes and the methods processed.

Next, the call to `ProcessEnterElement()` (step 3) sets the proper `ErrorReport` context (explained later) in step 4, which is used when processing the sub-elements of the element at hand. The `ProcessSubElements()` method in step 6 calls the `Check()` method of all sub-elements. As shown in Figure 4.20, the specific attribute checkers for different meta-elements, e.g., `ADCAssembly` and `ADCClass`, are derived from `AttributeDependencyChecker` by implementing the abstract methods: `Check(object t)`, `ProcessEnterElement(object t)`, and `ProcessSubElements(ref ArrayList ctx, object t)`. For illustration, Figure 4.23 shows the implementation of the `ProcessSubElements` method in `ADCClass`. Only the traversal functionality for finding the sub-elements is part of this method.

The context to be used during the processing of a node and its descendants (the parameter `ctx` in the signature of `ProcessSubElements`) is managed as an `ArrayList`, similarly to the method call stack frames in a compiler [ASU88]. A frame in the context array contains the attributes and the dependency attributes of a particular element. As the structural tree is traversed by calling `ProcessSubElements`, the dependencies context is filled by passing the `ctx` object to every processed sub-element. When being processed, an element can modify the dependency information of the context frames of its parents. After all sub-elements of a given

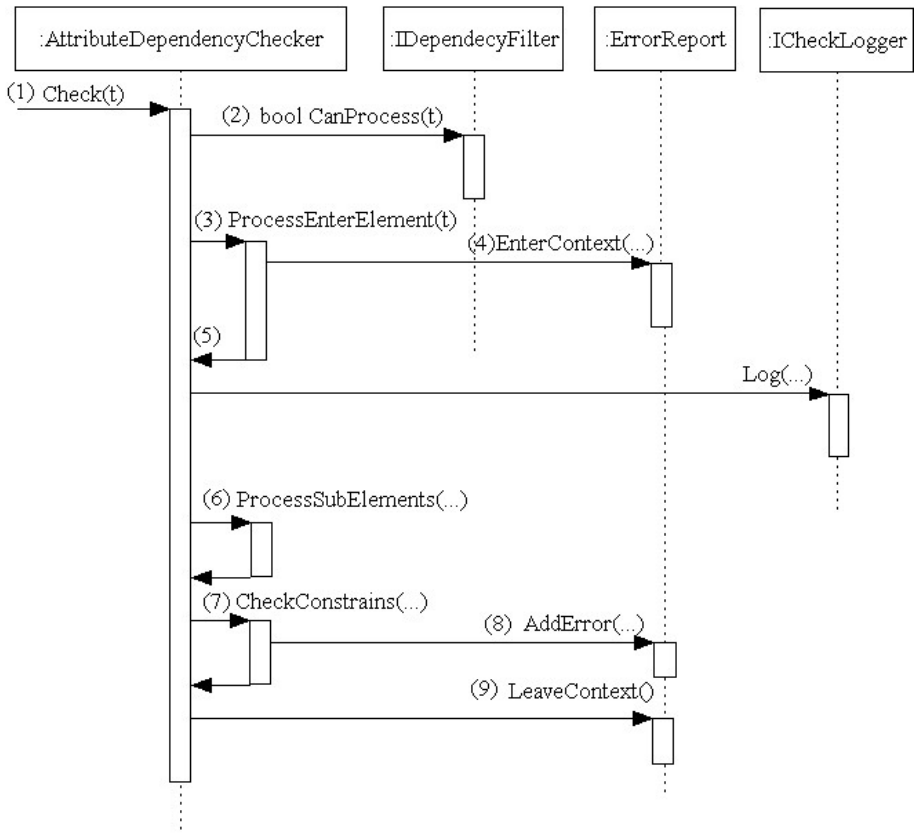


Figure 4.22: UML Sequential Diagram of `Check()` Method Call

element are processed, all the required context information is in the context stack frames. This information can be used to check the dependencies of the given element. The dependency meta-attributes of the current attribute are compared with the total dependency attributes for the current frame, using set operations in the `CheckConstrains()` method (step 7 in Figure 4.22).

The `:ErrorReport` object maintains its own separate context stack (set up in step 4), so that when an error is reported in step 8, the error message can be embedded within the proper structural context. The `ErrorReport` context is used to report messages in a useful way, as illustrated by the error message:

```
| Required CLASS attribute missing:
| adctests.CA01Attribute @ adctests->adctests.nunit.TDependencyUtils
```

This error message specifies that the required class attribute `adctests.CA01Attribute` is missing in the class `adctests.nunit.TDependencyUtils`, which is part of the

```
1  protected override void ProcessSubElements(ref ArrayList ctx, object t) {
2    MethodInfo[] m = ((Type)t).GetMethods(
3      BindingFlags.Instance |
4      BindingFlags.Public |
5      BindingFlags.DeclaredOnly |
6      BindingFlags.NonPublic);
7
8    foreach(MethodInfo mi in m) {
9      ADCMethod adc = new ADCMethod();
10     CopyStateTo(adc);
11     adc.InitialContext = ctx;
12     adc.Check(mi);
13   }
14 }
```

Figure 4.23: `ADCClass` Implementation of `ProcessSubElements` Method

`adctests` assembly. By default `ErrorReport` accumulates the errors, but this behavior can be changed via a switch, so that it will stop the checker execution when an error happens, by throwing a `ADCException`. The `ErrorReport` contains also functionality to accumulate, or immediately report, the improper usages in code of the parameters passed to the `DependencyAttribute` itself. An example is passing an attribute declared with a class lexical scope, as an argument to a `RequiredMethodAttribute` property.

Finally, the `ICheckLogger` interface (step 5) enables the programmer to associate a custom output stream logger with the dependency checker (line 4 in Figure 4.21). If the logger stream is not `null`, a hierarchy of the processed elements with details about their attributes and attribute dependencies is printed in the logger stream. A filter could also be used for custom logging. All objects shown in Figure 4.22, with exception to the `:AttributeDependencyChecker`, are singletons and are passed to the processing of the sub-elements as part of the context.

The implementation of the class `AttributeDependencyChecker` is generic with regard to the implementation of both the `DependencyAttribute` and the meta-model elements. This means that the implementation of the `AttributeDependencyChecker` class can be reused with new attributes as well as with other meta-model elements. The `AttributeDependencyChecker` achieves this generality by using a combination of the following three techniques:

- First, all the hierarchy information of the supported meta-model is factored out into two static methods of the `IContextMap` utility class. `AttributeDependencyChecker` uses the `IContextMap` to implement a strategy pattern [GHJV95]. By changing the `IContextMap` class, users can change the supported meta-model. Theoretically, the information in the `IContextMap` is enough to check the dependencies, i.e., no spe-

cific checker classes for different elements of the meta-model, e.g., `ADCClass` would be needed. However, the .NET Reflection API design is not consistent for traversing the meta-elements hierarchy. Other API-s, e.g., XML DOM [McL01], have a single base interface[28], from which all elements are derived. .NET Reflection API does not expose a single generic interface for all meta-element types, because the number of meta-elements is limited. This requires that when adding new meta-elements to the ADC library, special classes need to be derived for the new elements. These new classes will contain only code to traverse the sub-elements, as described above (Figure 4.23).

- Second, given that the structure information of the meta-model is present in the `DependencyAttribute` property names, the reflection inside the `Check(...)` method is used over the `DependencyAttribute` properties, to map them to the internal `IContextMap` data. The use of the reflection ensures that if the required / disallowed properties of the `DependencyAttribute` class are added or removed, the implementation of the `AttributeDependencyChecker` does not need to be changed. Another generic alternative would be to generate the checker code based on the `DependencyAttribute` implementation, but this would require to re-generate and re-compile the `AttributeDependencyChecker` class for every different version of the `DependencyAttribute` implementation.

- Third, the template method pattern [GHJV95] is used to call abstract methods that need to be implemented in the derived classes, such as the `ProcessSubElements` method in Figure 4.23, required to traverse the sub-elements. The entire checking functionality is part of the abstract class `AttributeDependencyChecker`.

The resulting ADC library can be easily extended to support new meta-elements. If a new type of checker for attributes of another meta-element need to be added, then a class from `AttributeDependencyChecker` must be derived. The derived class implements the abstract methods discussed above. In addition, the `IContextMap` class needs to be modified to accommodate the hierarchical structural relation of the new element, with regard to the existing elements.

4.3 Related Work

Attributes are a form of graph labeling [DJ90, Plu95, Men99]. Attributes make it possible to associate arbitrary semantics with graph nodes of interest, and can be used to drive graph transformations. Considering attributes in a very generic level is not very useful in practice. For example, being able to normalize a series of add / remove / modify operations applied to a graph

[28] The `Node` interface in XML DOM.

[Men99] is an interesting theoretical operation. In the domain of mobile applications considered as a use case in this book, the scenarios that add the same block of code or method, or remove it several times, do not happen. Graph labeling is more interesting when it is specialized for a domain. For example, in [Big00] tags are used to reduce transformation search space [vWV03] by serving as transformation hints, in the same way as tags in MDA [Fra03] models. The class of graph labels addressed in this book is made of attributes used directly at the programming language level. Attributes are complex graph labels that are specialized using parameters, contain behavior, and could be decorated themselves with other attributes.

Several general-purpose languages, such as .NET [Pro02] and Java 1.5 [Jay04], already offer support for attributes. As discussed in chapter §3, these languages offer also GAAST-like means to process the code decorated with attributes. Other non-mainstream approaches also exists [BCVM02, Att02]. These approaches fall in the category of API-based generators [Voe03a, CE00]. The GAAST-level of generality was the starting point for the modularization techniques presented in this chapter. The parsing issues were not addressed on purpose in Tango, to provide flexibility to use any GAAST API, or meta-programming tool discussed in section §3.5.2, e.g., [TCKI00, BLS98, BP01, IR97, BH02, QDo03] to map the AST to the common Tango OO model.

Tango modularization techniques can be seen as domain specialization of graph rewriting systems [DJ90, Nag96, Men99]. The changes introduced by an attribute-driven transformer to an annotated class could be seen as a series of primitive operations. Tango supports a hardwired language meta-model with a finite number of elements. This makes it possible to enumerate the edit operations. The approach is similar to implementing graph-based schema evolutions [BK87], but specialized for a special set of OO attribute-driven transformers.

Stratego [Vis01b, vWV03] is a library of strategy operators and traversal primitives to reduce the program transformation costs. Stratego is implemented as a set of reusable transformations primitives that can be used with any language system, given that a suitable parser to map the specific language constructs, to Stratego constructs can be created. The Stratego modules are translated to C code. Stratego organizes transformation operations as series of strategies that work over grammar rules, to support, for example, term rewriting [DJ90]. *"A strategy is an operation that transforms a term into another term or fails"* [Vis01b]. A strategy is made up of:

- *Traversal primitives* that support reusable term traversal techniques. Examples: *all* - all nodes in an expression; *repeat* - an generic iteration strategy to repeat an operation; *bottomup* - does a bottom-up traversal of an expression.
- *Strategy operators* that allow the combination of strategies. Examples: *sequential composition* - combines a sequence of traversal primitives; *choice* - selects between two or more traversal primitives.

The approach presented in Tango is different in two ways from Stratego: (a) the strategies are

organized in layers and (b) the strategies are specialized for the entities of each layer.

There are several general-purpose transformation frameworks, e.g., DMS [BPM02], TXL [Cor04] and ASF+SDF [vdBHKO02], which can be used to address domain-specific transformations. These general-purpose frameworks have an open meta-model meaning that any specific language can be mapped to them. These tools could also be used to process explicit attributes. However, generic frameworks do not provide declarative specialized rules for every specific domain of interest. Hardwiring the domain, as in Tango, enables abstracting the transformation strategy operations at different levels.

Tango's vertical attribute modularization bears resemblance to layered context-sensitive graph grammars [RS97] and multi-stage programming [TS97]. Layered grammars [RS97] have been used to recognize left-hand side patterns in graphs, and specify an evaluation order for graph rewriting rules. Tango is specialized only for attribute-driven transformers. Multi-stage programming uses special annotations to structure the computation of expressions as multi-stage transformations. Multi-stage programming is driven by the need for optimizations and partial evaluation [JGS93] and uses a predefined set of annotations. Tango can be used to implement arbitrary invasive transformers [Aßm03] at source code level, in the top of a GAAST-like language. Tango also supports an arbitrary number of user defined attributes.

Some transformation approaches generalize the ways a transformer works with an AST-like representation of the program: (a) *Filtering related approaches*, e.g., JPath [Cal03], are motivated by XPATH [XML99] node query language for XML [SG01]. The idea is to be able to define XPATH-like queries over the AST to declaratively select sets of nodes of interest. For example, the query "`//class[@name="Class1"]/method[@name="Method1"]`" selects and returns all methods named *Method1* found in class named *Class1*. (b) *Mapping related approaches* build upon filtering related approaches and that are motivated by XSLT [TS01] and more recently by MOF Query View and Transformation [DSC03] proposals. These approaches define declaratively how to map a set of selected nodes to another set of nodes, enabling this way the generic transformations of one graph to another. These approaches are very general. For example, XML and XPATH are used internally in Tango to implement the specific Tango features on top of them.

The feature-based categorization of MDA transformation approaches presented in [CH03] distinguishes between *model-to-code* and *model-to-model* approaches. Tango is a *code-to-code* transformation framework. Given that the transformation of a marked model to marked code is trivial (§3.2.3), the approach can be seen also as *model-to-code*. With regard to the implementation, Tango can be seen as a *template*-like transformer with source-oriented deterministic iterative rules. The categorization in [CH03] is too wide, and many tools could fall into the same category. Tango is unique for its well-structured layers, extensibility of primitive operations, and heavy reliance on inner tags.

Code snippets in Tango are an example of the combination of template and frame-based

systems [CE00, Voe03a]. Similar approaches include syntactic unit trees [MF03] and various template and macro-based approaches [IR97, BP01]. Tango code snippets differ from these approaches, because code snippets are restricted to represent only non-structural elements, e.g., a block of code inside a method. Tango code snippets can also be labeled with inner tags and reprocessed later as atomic elements of the AST.

Declaring and checking attribute dependencies is one example of explicitly enforcing architectural principles [Min98]. Attributes offer a unified way to express evolution invariants in languages that support explicit annotations, given that any structural entity can be decorated independently of the syntax. This makes attributes attractive for expressing law-governed system evolution rules. Architectural principles that must hold between program entities can be expressed as attribute dependencies between architectural attributes used to decorate program entities. System-wide invariants can be expressed in .NET as `Assembly` attributes and system-wide rules can be expressed declarative by using meta-attribute annotations over the architectural attributes.

The meta-attributes approach enables explicit factorization of cross-cuttings concerns in attribute-driven transformers. Meta-attributes make the cross-cuttings concerns definitions declarative and fit well in the paradigm of a GAAST language. Meta-attributes are not well-suited for checking arbitrary program restrictions. Cross-cuttings concerns based on meta-attributes can be factorized using specific tools, e.g., ADC (§4.2.4) or any generic meta-programming tool. There are also generic meta-programming tools [PMD03, Bor03, EMOS04] designed to ease enforcing of program rules that cannot be checked directly by the programming language compiler. These tools could be used to implement checking based on attributes directly, or based on meta-attribute factorizations if they can access them. Cross-cuttings concerns could also be factorized with AOP [KLM$^+$97, Lad03, EMOS04, KM05] techniques (§2.4). Aspect-oriented programming techniques can be also used to enforce architectural decisions. An example how AspectJ [KHH$^+$01] can be used to enforce system-wide constrains is given in [SY01][29].

Attribute extension grammars can be used to enforce properties about library components that can not be enforced otherwise with object-oriented systems [Hed97]. The work has been superseded by language technologies, such as .NET [Pro02], that directly support attributes and offer GAAST-like access to the AST information along with the decorated attributes. The meta-attributes approach enforces rules at a more abstract level, using attributes to define declarative rules that must hold between attributes.

It is also possible to define a central notion of dependency and model any kind of dependency by a Dependency Code Calculus (DCC), based on a computational lambda calculus [ABHR99]. Such a formal abstraction can be interesting for proving properties of dependent system elements, but it must be specialized to a specific domain to be of real usage, yielding

[29] There are meta-programs that cannot be expressed as AspectJ programs.

different special purpose calculuses [ABHR99]. However, some of the dependency problems mentioned in [ABHR99], such as slicing calculus, do not map directly into source code program dependencies and cannot be expressed as source code attributes.

4.4 Chapter Summary

The transformations of attribute decorated entities become problematic in a system when the number of attributes and their required interpretation context grows. The attribute-driven transformer implementation is the ultimate source of documentation about the semantics of an attribute or a set of such. It is important that the transformation process is implemented in a well structured and modular way to ease understanding and enhance reuse.

The domain knowledge for attribute-based DSA used to support mobile product-lines is explored to modularize attribute-driven transformers. The overall attribute-based DSA transformation is structured in parts corresponding to the domain assets, which are modeled as attribute families.

To enable declarative composition of the transformer units, inner attributes are used to express the inter-transformation semantics. Inner attributes serve as a uniform composition mechanism for attribute-driven transformers, that treat the inner attributes equivalently to explicit attributes. Inner attributes make the transformer coupling explicit.

The knowledge of semantics of the domain assets is used to define the transformation workflow. The process is partially automated by having the dependencies expressed as *before* or *after* lists. The generated dependency graph can be overwritten manually by using specialized workflow operations to resolve potential conflicts.

The hardwired OO language meta-model addressed by Tango is explored to achieve a vertical modularization of transformer units. The transformation strategy is structured according to the nesting of the structural elements in the language meta-model. This structure is enforced with language constructs. Specific strategy operations optimized for each layer are created to make the transformer implementations more declarative.

The common OO meta-model is isolated inside the transformation framework, from the details of the specific language, by using code snippets. Code snippets are parameterized templates that contain language specific syntax, and are used only inside the member constructors, e.g., the method transformers.

Repetitive cross-cutting attribute-transformation concerns are factored out of individual transformers, and made part of generic tools of the transformation framework. Meta-attributes offer a convenient way to implement a mix-ins structure, used to declaratively express the cross-cutting concerns of attribute-driven transformers.

Attribute dependencies are a cross-cutting attribute-transformation concern that is made declarative by using meta-attributes. A special tool was demonstrated, which extends .NET to declare, and automatically check, attribute dependencies using Reflection over the IL binaries.

Chapter 5

MobCon: A Generative Middleware Framework for J2ME

There is abundant evidence to show that high buildings make people crazy.

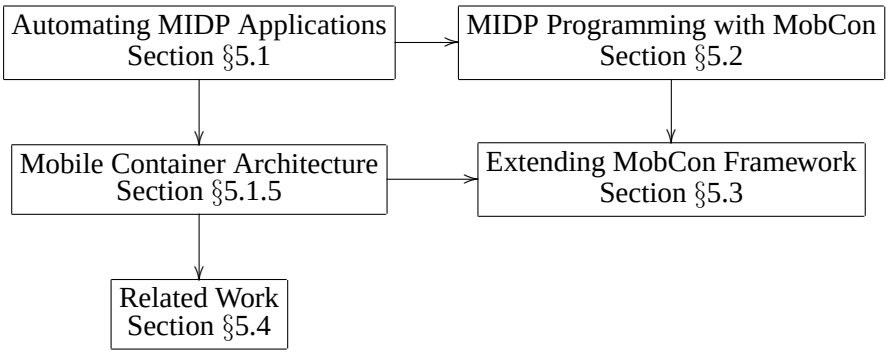
C. Alexander et al, Four-Story Limit pattern, A Pattern Language: Towns - Buildings - Construction, Oxford University Press, 1977

MobCon[1] is a framework[2] [Mob04] for automating cross-cutting concerns of Java 2 Micro Edition (J2ME) - Mobile Information Device Profile (MIDP) [Jav05] applications. MobCon uses the technology for attribute-based DSA applied to organize product-lines, introduced in this book. MobCon is implemented as a generative framework that organizes the product-line domain assets as container-managed services specialized for the MIDP development.

The domain of J2ME MIDP applications is presented in section §5.1. To support a low-cost programming model, a GAAST-like representation is implemented for MIDP. The domain variability is expressed by means of attribute families.

Section §5.1.5 represents the details of mobile containers and how they can be used to

[1] The name *MobCon* stands for *Mobile Container*.
[2] This chapter shares content with reference [CM05b].



structure the MIDP product-line domain assets. A comparison of mobile containers with server-side enterprise containers is presented.

MIDP programming with MobCon is explained in section §5.2. The declarative programming model introduced by MobCon preserves the model of the domain assets in code and results in shorter development time.

Section §5.3 gives details about the MobCon internals. Knowledge of the MobCon internals is needed to extend MobCon. The plug-in meta-data and customization of the transformation workflow are explained. Related container approaches are discussed in section §5.4.

5.1 Automating Cross-Cutting Concerns of J2ME MIDP Applications

The development of mobile software has been moving from OEM[3] specific operating systems and API-s, e.g., Palm OS [Pal04] and Windows CE / Pocket PC [Bol03, Poc04], to nearly standardized frameworks, e.g., Java 2 Micro Edition (J2ME) [Jav05] and .NET Compact Framework (.NET CF) [YD04]. The latter aim at offering the same API to application developers independent of the underlying operating system found in the mobile device. J2ME and .NET CF offer an abstraction of the device hardware and software and enable for the mobile software to be portable from one device to another. The success behind frameworks, such as J2ME [Jav05] and .NET CF [YD04], results from the current trend to abstract from concrete hardware execution models by means of *virtual machine* models against which *pseudo-code* is generated. The pseudo-code is then run (compiled, interpreted, compiled just-in-time) in a specific virtual machine specialized for a given mobile device.

The virtual machine execution model introduces an abstraction between the real hardware and particular OEM software on the one side, and the application programs on the other side. The

[3] Original Equipment Manufacturer.

virtual machine abstraction is especially handy for mobile software, where the underlying device hardware and software models change rapidly. It becomes the responsibility of the manufacturer to maintain the specific virtual machine for the manufactured devices. The software applications are mostly isolated from such changes, unless the newly provided features need to be used.

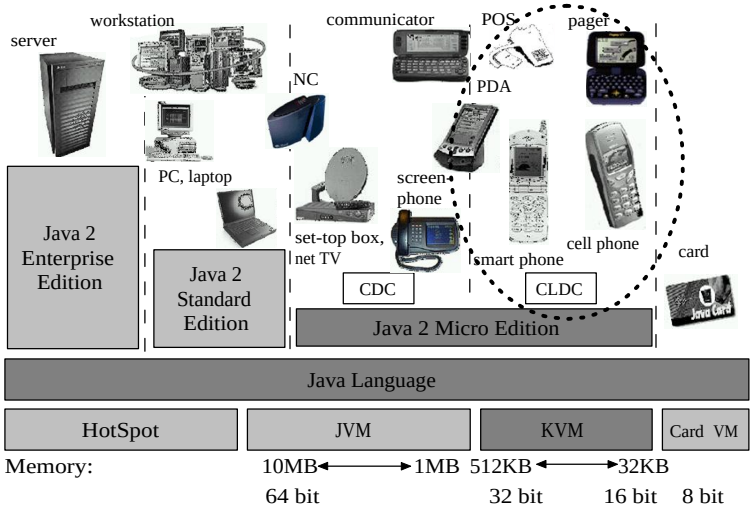


Figure 5.1: Java Technologies (Source: http://java.sun.com)

Figure 5.1 shows the range of the Sun's Java technologies classified according to the targeted devices and hardware technology. Java 2 Micro Edition (J2ME) is the set of Java technologies available for mobile devices. The J2ME technology itself is separated into several families that target different sets of mobile and embedded devices. The most powerful mobile and embedded devices are grouped in *Connected Device Configuration* (CDC) family. The CDC set of devices includes embedded software into TV-Sets, satellite receivers, home video / DVD player devices, powerful PDAs, etc. These devices have enough CPU / RAM resources, power supply sources, and often, constant network connectivity. Standards such as *Open Services Gateway Initiative* (OSGi) [OSG02], implemented as *Java Embedded Server* by Sun, target CDC devices that could run some form of the usual Java runtime[4]. Scaled down versions of the Java runtime, such as, *Personal Java* profile, could also be used on some of the CDC devices.

Of special interest for this chapter are the devices belonging to *Connected Limited Device Configuration* (CLDC) family. The CLDC family is well supported by Sun and several mobile device hardware vendors [J2M04, J2M02a]. Devices in the CLDC family include, for example, small PDA-s and cellular phones. While PDA-like mobile devices are becoming more and more powerful, the CLDC family will continue to remain in focus as a representative of portable

[4]JDK 1.2.

mobile devices. CLDC compatible devices expose many properties usually associated with mobile devices, e.g., limited processing resources compared to other contemporary devices, limited battery life, and sporadic network connections. The CLDC specification is made up of *Mobile Information Device Profile* (MIDP) [J2M02b], whose technology stack is shown in Figure 5.2. MIDP relies on the CLDC virtual machine standard, whose back-end is specialized for the OEM operating system in each device model. The MIDP applications can use the MIDP API-s only and be portable in any CLDC device. Alternatively, MIDP applications can also rely on OEM specific code and be partially portable.

5.1.1 The Domain: Automating J2ME MIDP Applications

The domain addressed in MobCon is that of J2ME MIDP 2.0 [J2M02b] applications. J2ME is intended to make programming uniform and simple for mobile devices, following the same goals as Java technology for desktop and enterprise computing [Hel02]. The MIDP programming model [Eff02] is based on a very stripped down version of Java. MIDP has a simple language run-time, a simple threading model, and no reflection. The collections and utility libraries are limited to basic types (`Hashtable`, `Stack`, `Vector`). The input / output libraries contain basic stream support. MIDP also contains a set of specialized packages that target CLDC devices. They deal with GUI creation, supporting the idea of a `Display`, various other input / output gadgets, and game support. It is also possible to play various types of media, e.g., sound or video.

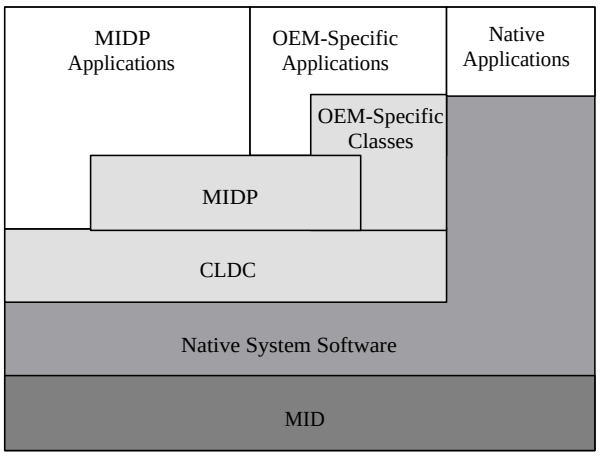


Figure 5.2: MIDP Technology Stack (Source: http://java.sun.com)

The network communication is supported via a special package for CLDC devices. The basic supported protocols (for MIDP 2.0) include HTTP, HTTPS, UDP, and raw sockets. These

protocols are not necessarily supported over TCP/IP. The idea is that any specific device network (packet based) protocol can be used to implement the above protocols and the implementers are free to choose the underlying details. Security is supported via the ability to sign applications and authenticate connections with X.509 certificates. Data persistence is supported via a simple record based model for the local device memory.

A MIDP application is called a *MIDlet*, similarly to a Java applet. A MIDLet should be derived from a specific predefined MIDP class, and should implement a set of required callback methods, e.g., `startApp()`, that are called by the virtual machine during various phases in the application's life cycle. For an extended and practical discussion about the MIDP programming model the reader is referred to [Moo05].

The MIDP technology has proved to be very successful for mobile phones and other small devices, and is supported by many hardware device vendors [Law02, J2M04, J2M02a, Del02]. There are also good tools and IDE support, and a free reference implementation by Sun. MIDP will be used in this chapter as an underlying technology to demonstrate the concepts represented in this book.

Despite its simple programming model, MIDP lacks support for modularizing the implementation of *technical concerns*, e.g., data persistence, screen management, session management and security management. These technical concerns [Par72] are secondary to what a MIDP application does, but still they must be taken care explicitly by the developers to get the application running. When using MIDP, the implementation of secondary concerns cuts across several mobile applications, or even several places within a single application, resulting in duplicated code. The lack of modularization leads to scattered and tangled code: Not only is the code for technical concerns scattered around several places, it is also tangled with the core functionality. For example, every time the data needs to be saved locally into a mobile device, the *RecordStore* structure that MIDP exposes, needs to be used. The record structure has to be customized manually to match the structure of the data. Similar code is repeated in many places with only very slight differences. The same applies to other concerns, e.g., creating GUI screens and managing the session context.

The consequence is manifold. First, it makes the development of mobile applications a tedious and error prone task increasing its costs. Furthermore, it is even harder to maintain and further develop such applications, especially in the face of rapidly changing middleware technology for the domain. Given the extreme speed of the domain, it becomes crucial to support off-the-shelf application components that can be reused in a variety of devices. This in turn calls for technology for modularizing technical concerns separately from the application functionality as was explained in chapter §2.

5.1.2 A GAAST-like Representation for MIDP

In chapter §3 GAAST-like languages were presented as a prerequisite for introducing low-cost attribute-based DSA to declaratively model the product-line abstractions. The Java dialect supported by MIDP does not support an attribute annotation facility as Java 1.5 does. MIDP also does not support a standard way to access the AST of the parsed MIDP Java code. As the focus in MobCon is in static code transformations, a GAAST-like API for static processing of the MIDP Java code is needed. Adding GAAST-like support to a language is a cheap one-time effort operation. MobCon development was faced with the same problem, to quickly introduce a static GAAST-like representation for the MIDP code.

The Java dialect of MIDP is similar to Java 1.4. MobCon emulates attributes with JavaDoc [Pol00] comments. A special form of JavaDoc comments using an @-sign before the name has been used in several Java 1.4 tools e.g., xDoclet [xDo03] and Attrib4J [Att02]. When using JavaDoc comments to emulate attributes, no grammar changes need to be introduced in a parser for Java 1.4, apart of the capability to process and preserve the JavaDoc attribute comments. Any tool that parses Java 1.4 could be modified to offer static support for a GAAST-like representation of the source code.

Several third-party generic parsers for Java 1.4 exists, e.g., ANTLR [Par02], or JavaCC [Jav02b]. Some specialized Java parsers, e.g., xDoclet [xDo03], qDox [QDo03] have support for processing attributes emulated with JavaDoc comments. Java 1.4 also supports a JavaDoc API that can be used to process JavaDoc comments and could also be used to drive attribute transformations. Unlike full-blown Java code parsers, these tools focus only on parsing JavaDoc comments and have various limitations in the way they present the Java code. The JavaDoc API, for example, ignores most of the AST information inside methods and cannot be directly used to introduce changes in its model of the Java AST. Other tools, e.g., xDoclet [xDo03] and qDox [QDo03], have similar restrictions.

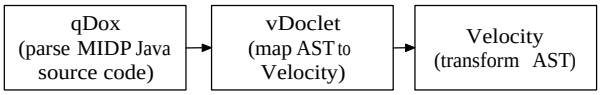


Figure 5.3: MobCon MIDP AST Parsing Tools

The qDox [QDo03] tool was chosen in MobCon as an underlying tool for creating a GAAST-like representation for MIDP code (Figure 5.3). There are several, properties that make qDox interesting. As mentioned above, qDox knows how to process JavaDoc style attributes. qDox is also a free open source tool with a very fast parser implementation, which ignores many parts of Java 1.4 syntax. In MobCon, the qDox tool was modified to fully parse MIDP code, that is, to return the method internals as part of the AST and to have support for Java arrays which was not implemented in the original qDox tool. Another very interesting aspect of qDox

that motivated its selection in MobCon was the integration of qDox with a tool called vDoclet [vDo03].

The vDoclet tool maps the AST parsed by qDox to a hierarchy of Java objects that can be accessed inside Apache Velocity [Vel03] script engine. Apache Velocity is an open source script engine motivated by WebMacro [Web03], mainly intended for supporting web applications. Using web script engines as meta-programming tools have been explored in many tools, one of the first being Gen<X> [Gen02]. The core of the Velocity is a generic template-like engine that can be used to transform any kind of parameterized source templates. An interesting property of Velocity is that it contains only very basic programming constructs, but allows any Java object to be mapped onto Velocity and accessed directly from the script code. The vDoclet [vDo03] tool uses this property of Velocity to make the Java AST accessible as native objects in Velocity. The vDoclet tool was also modified in MobCon to reflect the changes that were introduced in qDox. This way, a complete GAAST-like engine was made available in MobCon for transforming the J2ME MIDP applications.

5.1.3 Modeling MIDP Attribute Families

Attribute families were introduced in section §3.1.1 as a way to model the variability of the product-line domain assets with attribute-based DSA. MobCon models the generic non-functional MIDP concerns as attribute families. The attributes of the same family are combined with a name space dot-like notation.

```
@src
|- height
|- width
+- form
|  |- label
|  |- firstDisplay
|  ...
|  |- choiceGroup
+- textfield
|  |- label
|  ...
|  |- maxSize
...
```

Figure 5.4: The MIDP Screen Management Attribute Family

For example, the attribute family `@scr`, part of which is shown in Figure 5.4, denotes the screen management concern (§5.2.2). Inside a family, attributes are distinguished by the family sub-prefix separated by dots. For example, `@scr.label` denotes the technicalities involved with managing a label gadget within the screen management concern. Many MIDP attributes take arguments that are represented as strings separated by space after the attribute name, e.g.,

`@scr.label "First Application"` specifies the text attribute of a label.

End-programmers of the MobCon use the predefined set of MIDP attributes that come with the framework to build mobile applications. If an attribute is used inside a component (class), MobCon requires that the attribute family name be present to decorate the component itself. The decorated source code is then processed and the required MIDP technical concerns are injected.

5.1.4 The MobCon Transformation Engine

The *MobCon Transformation Engine* (MTE), illustrated in Figure 5.5, is designed to be a general-purpose transformation framework organized around the modified Java 1.4 parsing tools introduced in section §5.1.2.

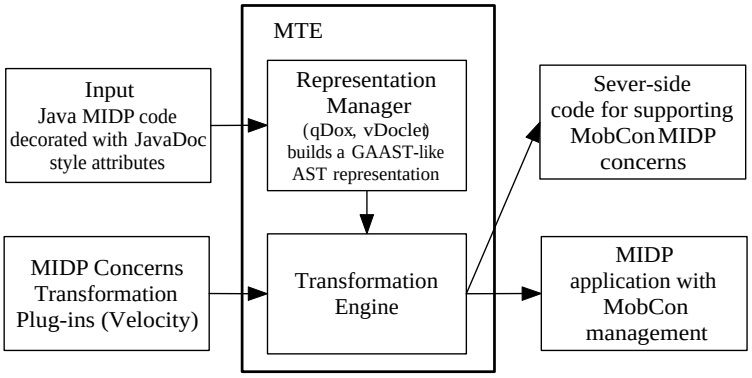


Figure 5.5: MobCon Framework

The individual MIDP concern transformers are implemented as plug-ins. A plug-in may contain one or more Velocity scripts, Java code, or other resources that describe the plug-in (§5.3). The MTE contains functionality to properly find plug-in dependencies and manage the transformation workflow (§5.3.1) by invoking the Java parsing and transformation tools as necessary. Experience with MTE plug-ins was used to develop the Tango attribute-driven transformation framework. Tango and different aspects of building modular attribute-driven transformers processing including those in MTE were discussed in chapter §4.

MobCon relies on the template method pattern [GHJV95] to support well defined extension points (§2.1.3). An abstract class, `AbstractMobApp`, that inherits from MIDP `Midlet` contains the bulk of the generated code. Users do not modify this class directly, since it is modified by the generator each time the attributes in the original source change. Instead, programmers add code to a derived class, `MobApp`, which is called by the abstract class. The

template method pattern is useful for generative frameworks [Voe03a]. It allows programmers to extend the generated code in a modular way without editing it directly.

5.1.5 The Mobile Container Architecture

Section §2.2.3 motivated the usage of the software containers as an architectural abstraction to organize the domain assets in software product-lines. Section §2.2.4 explained the need to specialize the container concept to address the MIDP domain concerns. One or more mobile client applications run on a mobile device and could connect with server-side services as illustrated in the case (a) of Figure 5.6. The server-side part contains functionality that is common to server all clients and functionality specialized for mobile clients, shown by the gray box part in the case (a) of Figure 5.6. This specialized client functionality can be factored out from the server part into a separate server-side component known as *adaptive proxy* [FGCB98], as in the case (b) of Figure 5.6. This separate part is called a *proxy* because it impersonates the server to the mobile clients and vice-versa the mobile clients to the rest of the server (environment) services. It is called an *adaptive proxy* because it does not simply forwards the messages and data from the mobile clients to the server and vice-versa, but also actively modifies the data to adapt them to the specifics of mobile clients. An example of such adaptation is the modification of the resolution of images returned by the server services, to match the resolution required by a mobile device.

A mobile container addresses the automation of technical concerns in a mobile application. However, because the functionality of the adaptive proxy part is tightly connected to the functionality of the mobile clients, a mobile container automates both the mobile clients and the adaptive proxy, as in the case (c) of Figure 5.6. The mobile client can rely on the container to provide common services specialized for its requirements. In a similar way, in the server-side, the adaptive proxy can make use of the server-part of the mobile container services. The mobile container supported services in both client and server parts are generated and injected based on the requirements of the mobile applications. While a mobile client can still use the server-side services directly, usually it will rely on the mobile container to manage all the communication.

The conceptual architecture of a *mobile container* is shown in Figure 5.7. A mobile container is made of a *client-side* (**MCCS**[5]) and a *server-side* (**MCSS**[6]) part. In the MIDP programming model, the applications cannot share code, including libraries or carry inter-process communication (IPC). This requires to have a MCSS part for every mobile application. It is specialized by generation in MobCon to contain only the functionality needed by the particular application. The MCSS part is instead shared by many mobile clients. It contains generic code in MobCon to identify the clients and fulfill the client-specific requests. The mobile container as a whole manages the communication between the mobile application and the server, and trans-

[5] *Mobile Container Client-Side* part.
[6] *Mobile Container Sever-Side* part.

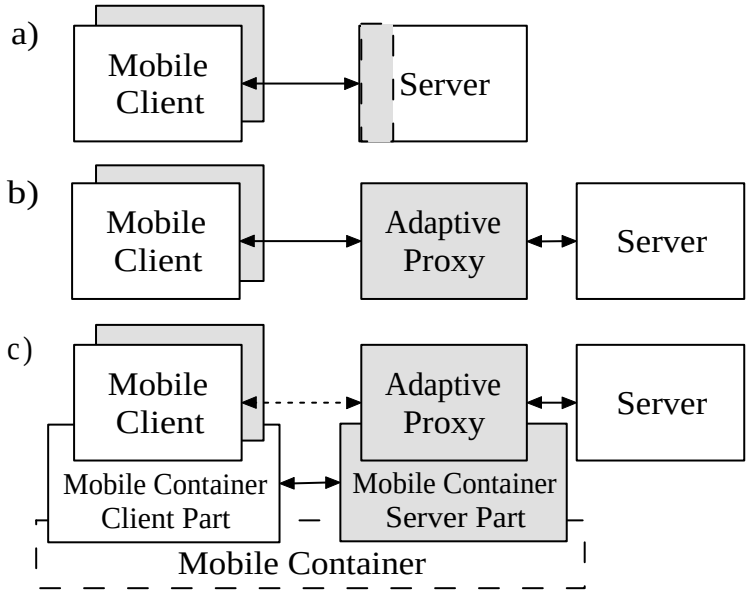


Figure 5.6: The Mobile Container

parently handles the requirements for services by a mobile application. The application specific functionality in both mobile client and the proxy, that is not provided as generic services by the container, is the only code programmed manually by the developers.

In MobCon, the MIDP transformation plug-ins contain functionality to generate MobCon MCCS code to support the transformed MIDP application, and MCSS code used to support the MobCon concerns in server. The MCSS code must be integrated manually in a server-side application. In MobCon the container is open and extensible. This means that depending on the specific MIDP product-line, developers could add new services to the mobile container needed by server application apart of those already supported by MobCon. This way the mobile container serves as an architectural abstraction (§2.2.3) to organize the MIDP domain concerns in a product-line. This architecture can support at the same time, several families of different domains.

There are more differences between mobile containers and other software (enterprise) containers apart of the explicit separation into two separate physical parts. A mobile container extends from the client device to the server-side. A normal enterprise container works on the server-side and contains some proxy broker code to run on the client. The mobile container does the reverse. The broker-like code is at the server-side. The MCCS part takes care of automation of cross-cutting concerns in a MIDP application running on a mobile device. In an enterprise container, e.g., EJB [EJB03], the client-side contains only the communication bus and proxy

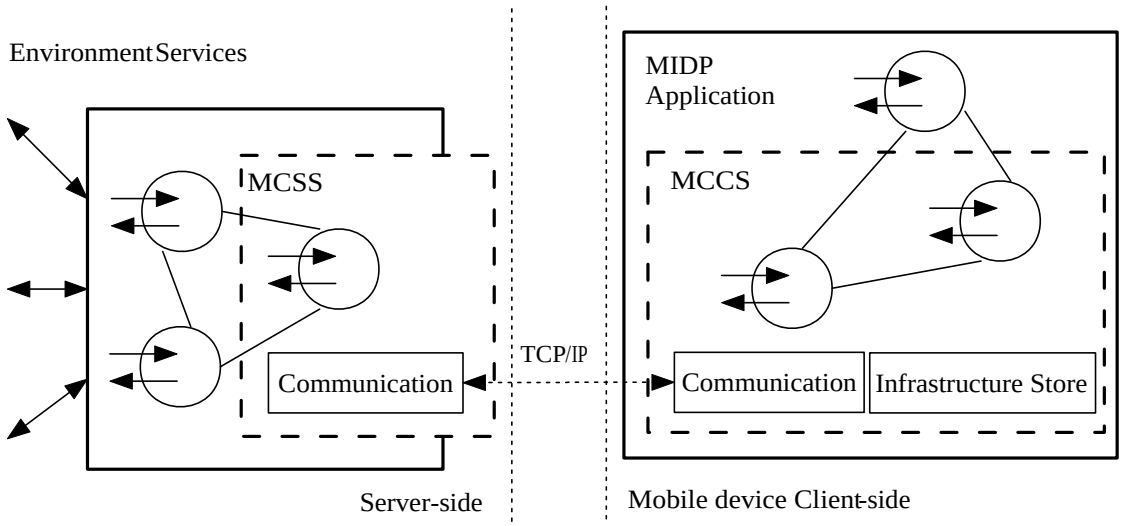


Figure 5.7: Mobile Container Components

code [VSW02]. The MCCS contains the complete functionality to manage the concerns of MIDP applications. A MIDP application relies on the mobile container to provide the services that map to the required MIDP domain assets. The container manages different aspects of the MIDP application, e.g., the persistence of the mobile application components, and the network communication between the mobile device and the server-side services. Other, more specific domain services, such as image adaptation (§5.2.4), are also supported in the same manner as the more generic services by the mobile container (§2.2.3).

The MCSS part deals with the server-side cross-cutting concerns (CCCs) [Par72, KLM+97], which are specific for mobile device applications. A mobile device client is different from a desktop client because of the limitations in the processing power, or hardware configuration, e.g., the size of the screen, or the available memory. A mobile device client is neither a thin[7] client, because it should be able to work with sporadic network connections, nor it is thick clients since there are usually processing restrictions on the mobile devices. The MCSS part addresses mobile device client issues transparently. The interface of the MCSS part to the rest of the server-side services is that of a normal desktop client. The rest of the server-side software in Figure 5.7 communicates only with the a mobile device client only through the MCSS. The MCSS part contains, thus, more functionality that its symmetrical part in a client of an enterprise container and handles the technical and specific concerns of the adaptive proxy [FGCB98].

[7] A thin client, here, is an application that contains only the user interface logic and delegates most of the application-functionality processing to the server. A think client does all the data processing itself.

5.2 MIDP Programming with MobCon

MobCon is presented to the end-users as a set of attribute-based DSA that support the automation of several technical (and specific) concerns in a product-line for J2ME MIDP applications. The attribute-based DSA modify the semantics of the existing components of a MIDP application. MobCon currently provides MIDP 2.0 support for *data persistence*, *screen management*, *session and context management*, *image adaptation*, *encryption*, and *network communication*. End-users may extend this set of services or modify the existing services (§5.3). MobCon generates code for the technical MIDP concerns on the client-side (mobile device) and code to be placed on the server-side. The name *MobCon* will be used synonymously with the term *MIDP mobile container* in this section.

A design goal of MobCon is that the attribute decorated code can be used with the manually written parts of the application in a seamless way. The developers can (a) rely on MobCon for the entire application management, or (b) use parts of MobCon for concerns of interests and ignore the others. Developers can use the MobCon generated components or replace them with their own component implementations. MobCon supports an incremental adoption of the MIDP container model inside a specific application. This section presents the MobCon programming model for MIDP applications. First, several MIDP concerns automated by MobCon are presented. Then a complete MIDP application that combines most of the addressed MIDP concerns for a medical X-Ray diagnostics application is explained.

5.2.1 Data Persistence

Any non-trivial MIDP application needs to store data persistently. The non-volatile memory in a MIDP device is organized as a set of record stores managed via the `javax.microedition.rms` package [Mob02]. Each record store is identified by a name unique for a Midlet[8]. It contains byte records identified by an integer, the `recordID`.

A MIDP application might need to save data that are too complex to be indexed only by integers. For illustration, consider the *GameScore* record store example in the documentation of the `javax.microedition.rms` package [Mob02]. A *GameScore* object contains a `playerName`, a `score`, and an optional `comment`. Both the `playerName` and the `score` fields are used as primary keys. There is no direct way to enable accessing data in a record store for *GameScore* via arbitrary primary keys. The `javax.microedition.rms` package provides support to map arbitrary keys into `recordID`-s by means of enumerating records in a store according to different criteria. The criteria must be coded in a case by case basis for each record store type by implementing two required interfaces: `RecordComparator` and `RecordFilter`. The `RecordComparator` defines a partial order over the records in the

[8] MIDP applications are known as *midlets*, similarly to Java *applets*.

store, while `RecordFilter` helps to selects certain records.

When using MIDP, a developer must always take care of the store organization details. The store details are intermingled with the application functionality, in the case of the example with the game functionality. This adds significant accidental complexity to the code. This is illustrated by the implementation of a store for the *GameScore* data which comes with the `javax.microedition.rms` package [Mob02] documentation. The code consists in about 216 lines of code of which, as will be shown below (Figure 5.9), only about 12%, belongs to application functionality.

The increased complexity of the programming model is only one aspect of the problem with a direct approach for implementing the data persistence concern. Other, tightly related issues are maintenance and evolution, which are made more difficult by the lack of the modularity when directly implementing the technical concerns, e.g., data persistence. Automating the data persistence issues and hiding them from the programmer is highly desirable. Implementing a MIDP store for custom data is a routine operation in a MIDP application and the details are only slightly different from one case to the other.

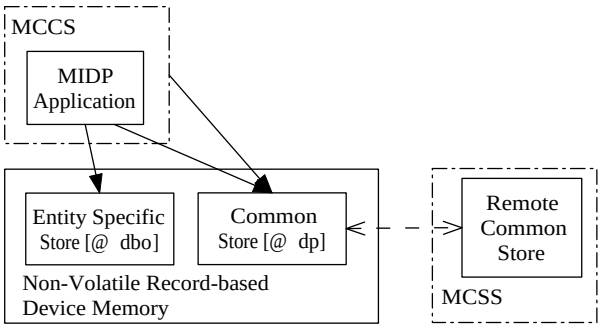


Figure 5.8: MobCon MIDP Data Persistence Architecture

MobCon addresses data persistence as a technical concern to be factored out by the container and automates store management. The complete data persistence architecture is shown in Figure 5.8 and is supported by two attribute families `@dbo` and `@dp`. The `@dbo` family supports specialized stores for components entities. The `@dp` family supports data persistence in a shared common store. To illustrate the `@dbo` entity data persistence, Figure 5.9 shows the complete code for implementing the *GameScore* example in MobCon using the `@dbo` attribute family, as well as some sample code of how the `GameScore` class can be used.

```
1  /**
2   * @dbo
3   */
4  public class GameScore {
5      /**
6       * @dbo.pk
7       * @dbo.min 1
8       * @dbo.max 256
9       * @dbo.sort asc
10      */
11     private String playerName;
12
13     /**
14      * @dbo.pk
15      * @dbo.min 0
16      * @dbo.max 10000
17      * @dbo.sort des
18      */
19     private int score;
20
21     /**
22      * @dbo.min 0
23      * @dbo.max 512
24      */
25     private String comment;
26 }
27 /**
28  * @dbo.use GameScore
29  */
30 class GameApp {
31    ...
32    public void modifyScore(string userName)  {
33      GameScore g = GameScore.retrieve(userName);
34      ...
35      g.setScore(...);
36      ...
37      GameScore.store(g);
38      ...
39    }
40 }
```

Figure 5.9: GameScore Example in MobCon

The `@dbo.pk` attributes in Figure 5.9 are used to denote the primary key fields. The `@dbo.sort` is used to specify how the records should be sorted according to the corresponding field. This information and the field type are used to automatically generate the `RecordComparator` implementation. The information provided by the `@dbo.min` and `@dbo.max` attributes is used to generate simple validation rules for primitive types[9]. In addition, an implementation of the `RecordFilter` interface is generated to enable record matching based on

[9] For integers, the `@dbo.min` and `@dbo.max` attributes specify the allowed lower and upper bounds, whereas for strings they specify the same bounds for the length.

the values of primary key fields. Filtering support also includes queries over records by means of the individual primary key fields.

The MobCon implementation of the class *GameScore* in Figure 5.9 consists of only 27 lines of code, that is, 8 times less than the pure MIDP implementation[10]. Furthermore, the semantics of the data persistence are declaratively stated in the MobCon implementation, rather than being buried in complicated imperative code for implementing the comparators. The generative MobCon container frees the programmer from having to deal with data persistence details. The programmer uses declarative attribute-based syntax to express the required functionality and can concentrate on the logic that is peculiar to the application. The MobCon framework handles all the details of the data persistence implementation. The attribute-based variability mechanism can also be easily mapped to a graphical wizard, or a visual modeling tool.

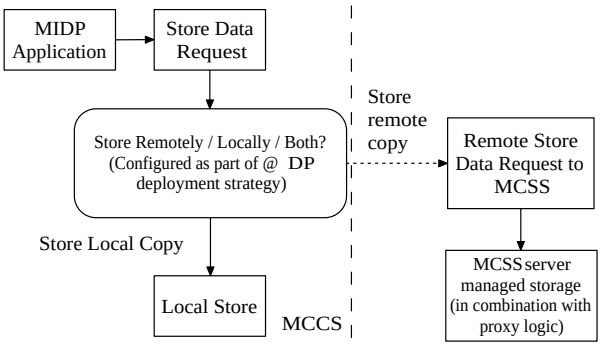


Figure 5.10: The `@dp` Data Store Request Management

Some mobile applications (or components) might not need a database-like abstraction to persist their data as supported by the `@dbo` family. A simpler persistence model, in which data is stored as unstructured strings in a hash table structure, i.e., a persistence model more similar to Java serialization, might be good enough. To support this type of data persistence, MobCon provides the `@dp` attribute family. Fields annotated with the `@dp` attribute family are persisted in a shared record store. This common store is also used by MobCon for its internal bookkeeping data, e.g., for session and network management. The code needed for performing the data storage is generated by the MobCon framework. The data of the common store could also be additionally stored remotely on the server, if there is a network connection (§5.2.6). When the `@dp` is deployed in a MIDP application, there is a possibility to select between saving all data (a) locally, (b) locally and remotely, or (c) remotely only. These cases correspond to different application scenarios. To explain how MobCon automates remote data storage case consider the logical flowchart of Figure 5.10, which shows how a store data request by a MIDP application is handled by the MCCS part.

[10] For details about the generated code see [Mob04].

The deployment case (a) saves all common data locally. In this case the MIDP application stores the data in the mobile device and does not need any remote network connection. The case (b) automatically creates a back-up of the data in server (apart of saving the data locally) when there is a network connection present. When a request for data is made, the local copy is used when no network connection is present. This ensures that the MIDP application can still function reasonably when no network connection is available. The deployment case (c) corresponds to the scenario when the data needs to be saved only remotely, but not locally. This is needed when there is a scarcity of local non-volatile memory in the device, or when the data will be shared and updated by other clients or by the server, and each client needs to operate on the latest copy.

These three scenarios are handled automatically as part of the `@dp` deployment. Other scenarios must be currently handled manually. The `@dp` generated code will take care of establishing the connection to a remote database, if possible, and will perform all the needed actions to persist the common store data for each client on the server-side. Because of the diversity of possibilities for storing data in the server-side (files, different relational databases, XML databases, etc.), MobCon only generates code to manage the store (represented by a `TreeMap`) and serialize data to byte arrays. The developers must add code manually in the proxy to store these data to an appropriate storage organization on the server. In the specific deployment case is known and shared by many applications, then the developers can customize the `@dp` server-side functionality found in `DataPersistenceTrans.java` file inside the `@dp` plug-in so that it can be reused by every individual application.

5.2.2 Screen Management

The organization of GUI[11] screens in a mobile application often follows well-defined patterns [Hui02]. For example, in the *Wizard Dialog* pattern, the user of a mobile application is lead through a series of screens (derived from the class `Displayable` in MIDP). In this case, it is possible to manage the screen stack automatically, for example, by removing loops, as illustrated in Figure 5.11. The screens S2 to S4 in Figure 5.11 are removed from the stack when S2 is re-shown from screen S4, so pressing the *back* button in screen S2 will show up the screen S1, not S4.

$$S1 \rightarrow \underbrace{S2 \rightarrow S3 \rightarrow S4}_{\textit{remove}} \rightarrow S2$$

Figure 5.11: Reducing Loops from the Screen Stack

The MobCon screen management concern takes over the issues related to screen organization and screen stack management. There are attributes in the `@scr` family to declaratively

[11] Graphical User Interface.

specify the order of screens, the characteristics of displayed forms, text fields, lists, alerts, and image screens. Common command actions are also taken care by the container implementation of the screen management concern[12].


```
1  /**
2   *@scr
3   */
4  public class MobApp {
5    /**
6     * @scr.label "Test"
7     * @scr.firstDisplay
8     * @scr.exitButton
9     * @scr.textField textField
10    */
11   private Form form;
12   /**
13    * @scr.label "First Application"
14    * @scr.string "Hello World"
15    */
16   private TextField textField;
17 }
```

Figure 5.12: MobCon *Hello World* Example Running on a MIDP Emulator

For an illustration, Figure 5.12 shows how a *'Hello World'* MIDP application is programmed using the screen management attributes. The `@scr` attribute attached to the class `MobApp` declares that this class requires the screen management concern. Only the interface gadget elements are declared as part of the code in the fields `form` and `textField` of the class `MobApp`. Each field element is then decorated with attributes, that declare how it relates to the rest of screen elements. The `@scr.firstDisplay` attribute in line 7 denotes the form that will be displayed first. The form will contain also an 'Exit' button (line 8). The properties of the elements, e.g., their labels, are given as specific attribute arguments.

Most of the repetitive code to be written when a MIDP *midlet* is created, e.g., the required `start` and `destroy` methods, as well as details about implementing the GUI, e.g., the code for creating, customizing and composing the GUI elements, is not explicitly present in MobCon applications. This code is automatically generated by the container. This makes the code involving GUI easier to write and allows the programmer to focus on the core functionality. The interested reader can find the complete MobCon generated code for the simple *HelloWorld* example in appendix §A. Should the MIDP API-s for managing screen change, the end MobCon based application code does not need to be changed.

[12] More features could be added to this concern in the future, including automatic screen caching and other additional interface gadget combinations.

5.2.3 Session and Context Management

Session management is needed in MIDP applications when they communicate with a remote server. The session data support stateful network communication at the application level. For thin clients, it is usually server's responsibility to manage the session data. MIDP applications may want to make use of the possibility to have access to the server session to store their own session data. Related to session management is execution context management, that is, identifying the right mobile device application related to the session data.

```
1  /**
2   ...
3   * @ses
4   * @ses.rememberLastDisplay
5   * @ses.id getMobileID()
6   * @log
7   * @log.logCommand
8   * @log.logMethod
9   * @enc
10  */
11 public class MobRay {
12  ...
```

Figure 5.13: Using Session, Log, and Encrypt MobCon Attribute Families

In MobCon, the context is implemented as a dictionary using a `Hashtable`. The application can save any data to the context and retrieve them. The session information is saved partially locally, and partially on the server. The session identifier, `mobileID`, is a unique 128 bit number. This unique ID is assigned to a MIDP application the first time it connects to the server. The `mobileID` is saved locally by the container for an application in mobile device. The `mobileID` is reused when the application is reconnected to the server (§5.2.6). The `mobileID` serves also as a kind of application *cookie*, which always identifies a particular device instance of a MIDP application. For the session concern, server-side code is generated that handles MIDP sessions and manages the identification based on the `mobileID` identifiers.

The `@ses` attribute should be used to decorate the main application class, as shown in Figure 5.13. Predefined support is provided to remember the last displayed form and the unique mobile device session identifier. The session identifier can be changed using the `@ses.id` attribute. In the example, the parameter of the `@ses.id` attribute is a code snippet `getMobileID()` that return the automatically generated `mobileID`. The `getMobileID()` is generated by MobCon when the `@ses` attribute family is used. In a static generative container, such as MobCon, it is possible to use code snippets as attribute arguments. This feature is explored by MobCon to enable the developers to conveniently specify small units of specialization code.

5.2.4 Image Adaptation

Mobile devices have limited capabilities so often data, e.g., images, need to be adapted before they are sent to an application that runs on such a device. Images accessed by a MIDP application via a remote server are adapted on the server to match the resolution and colors required by the mobile device. MobCon automatically handles image adaptation using the adaptive proxy pattern [FGCB98]. The images are stored in their original quality on the server. The MobCon generates code for the MIDP clients as well as for a proxy image adapter to be placed on the server-side. Currently MobCon handles only the MIDP PNG[13] format images and transforms them on the server-side using JIMI [JIM04].

```
1  /**
2   * @img.local
3   * @img.name "/xrayLocal.png"
4   */
5   Image ray;
6
7  /**
8   * @img.name getDbe().getImageName()
9   * @img.width 100
10  * @img.height 100
11  * @img.maxcolors 32
12  * @img.maxsize 25000
13  */
14  Image I_ray;
```

Figure 5.14: Using Image Attribute Familiy for Static Image Resources

The image resource can be either local (stored on the mobile device), or remote and stored in a server. The MobCon `@img` family can be used to manage the loaded images. The attributes supported by this family allow either decoration of predefined Image objects as shown in Figure 5.14, or enable loading images dynamically, by using an automatically generated (overloaded) method:

```
1  Image retrieveImage(String imageName, int width, int heigth,
2        int numColors, int maxSize, boolean dither){...}
```

The name of the image will be used to check first for local images, and then for remote images. Figure 5.15 shows how a request for an image is handled (a UML sequence diagram). First the MCCS part checks to see whether the image is local. If this is the case, MCCS returns the image to the MIDP application. If the image resource is not found locally, a remote request

[13] Portable Network Graphics.

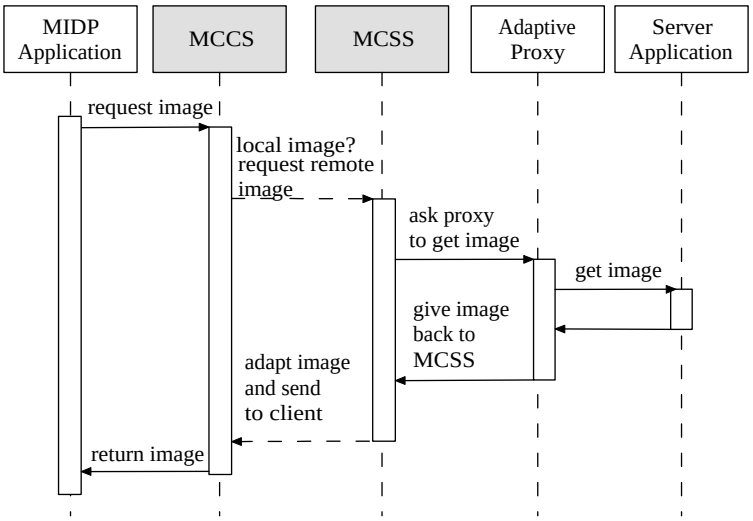


Figure 5.15: The Image Request Management

for the image is sent to the MCSS part. MCSS relies on the manual code written in proxy to get the image from the server storage medium of application. The image is then modified in MCSS to match the MCSS request parameters and is returned to the MCCS, which in turn returns the image to the MIDP application. The MobCon implementation of this concern currently does not cache the returned images, as this consumes a lot of memory in the device. MobCon leaves it up to the application to cache the image, e.g., using the managed `@dp` store as required.

5.2.5 Data Encryption

Currently, the encryption support in MobCon works tightly coupled with data persistence. When data is transfered over the network as part of data persistence, encryption functionality is added to it when required. This enables safe data exchange even when the `https` protocol is not supported by the MIDP implementation. In the future, encryption could be exposed as a separate concern to be used with network messages. Encryption can be activated application wide using the `@enc` attribute as shown in Figure 5.13.

When the encryption is activated the data persistence generated methods will be modified to encrypt / decrypt the serialized byte arrays based on an internal key, generated based on the 128 bit unique application `mobileID` (§5.2.3). The implementation of encryption concern is based on the third-party `BouncyCastle` [Bou04] lightweight cryptography API for MIDP. A symmetric AES stream cipher in the CFB mode is used to encrypt the network session data.

5.2.6 Network Communication

Network communication is currently not an explicit separate concern in MobCon. Other concerns, e.g., data persistence and image adaptation, rely on networking support from MobCon, because their implementation is split between the client application and the server-side (Figure 5.16). The exact detail of networking support in MobCon are not of direct interest for the scope of this book, so they will be only briefly explained.

MobCon supports a centralized way to send and receive network messages automatically. This keeps the number of network connections minimum in order to save resources. MobCon can forward any container management message to the appropriate component using a single TCP/IP socket connection. Network messages are by default asynchronous, running on their threads. The network messages are fully identified by:

a. A unique MobCon container instance ID (`mobileID`) in the device corresponding to a MIDP application. This number is used to identify mobile clients and the MCCS requests in the MCSS part, which deals with more than one client at a time (§5.2.3).

b. A domain asset plug-in ID that is unique for a given domain. That is, each MIDP concern plug-in has a unique ID. This number is used by the MCCS part to identify the concern's plug-in implementation that issues a message. Different concerns define different message subsets, to request different operations from the MCSS. The message subsets of a concern are further identified by the following two numbers.

c. A plug-in instance ID, that is unique for an application instance of the concern, and allows identifying (if needed) the exact component instance that uses the concern.

d. A plug-in dependent operation code, that can be used to carry out commands between the MCCS and the MCSS parts of the container. This code is implementation specific, and is used to specify the meaning of a network message, e.g., whether it is a data store or a data retrieve request.

5.2.7 Traceability

MobCon supports traceability in the same way as discussed in section §4.1.9. Traceability is implemented by a special attribute family `@log`, that can be used as shown in Figure 5.13. The `@log` attribute family supports the execution traceability. The execution log lists all MobCon transformers that have changed a given method at run-time as shown in Figure 5.18. The execution log functionality is implemented as just another MIDP plug-in in MobCon.

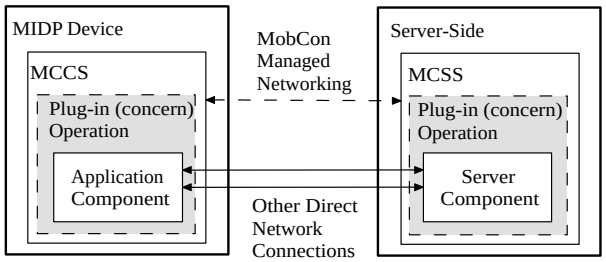


Figure 5.16: MobCon Managed MIDP Network Communication

5.2.8 Case-Study: The *MobRay* Application

To demonstrate the usefulness of the MobCon framework for the J2ME MIDP applications, a complete use-case application called *MobRay* is implemented. MobRay is based on a ubiquitous X-Ray medical diagnostics scenario [Rad03]. In this scenario, a remotely located hospital uploads X-Ray pictures of patients on its web server and registered doctors use PDA-like devices that run MIDP to review images remotely. Doctors send back to the hospital server their diagnostic comments for the reviewed X-Ray images. The MobRay application was chosen because its implementation demonstrates almost all concerns currently supported by MobCon.

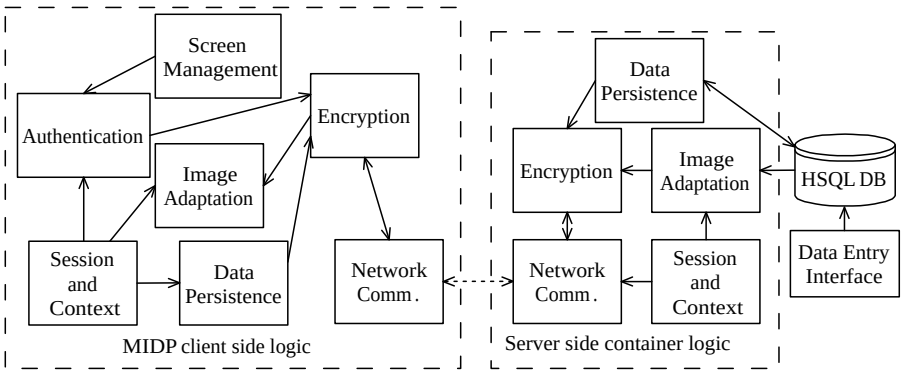
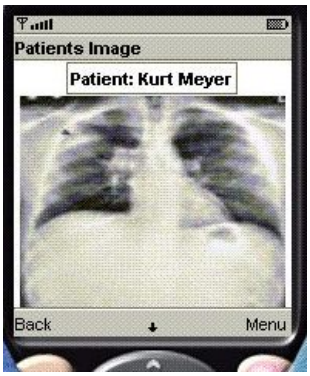


Figure 5.17: MobRay Application Organization

The high-level architecture of the MobRay application and its MIDP concerns are shown in Figure 5.17. The server-side part of this demo application is implemented using a HSQL database [Hyp04] with a simple command-line interface to allow uploading and editing of patient records. The mobile client-side of the application is developed using MobCon. The client-side contains functionality to connect to the server, to authenticate the doctor and to retrieve the x-ray

list to be diagnosed. The x-ray patients lists could then be used to review and send comments on selected x-ray images. The MobCon code for the server-side is placed on a folder called `server` inside the output folder generated for the MIDP application. The entire client side of the application consists of about 300 lines of MobCon source code (including comments), which then results in more than 1500 lines of code (1:5 ratio) in the generated application. Only the application functionality is explicitly coded. All the named concerns in the boxes of Figure 5.17 are handled by the code generated by the framework.




```
mobcon.message.MessageHandler: Getting message from server
METHOD:  commandAction [@ses, @log]
COMMAND:  'Select' executed in screen 'Choose a Patient'
METHOD:  choosePatientCG_Action [@log]
METHOD:  retrieveEntry [@log]
mobcon.message.MessageHandler: Getting message from server
METHOD:  setDbe [@dp, @log]
METHOD:  callF_patient [@scr, @log]
METHOD:  callSI_patientName [@scr, @log]
METHOD:  getDbe [@dp, @log]
METHOD:  callII_ray [@scr, @log]
METHOD:  getDbe [@dp, @log]
METHOD:  retrieveImage [@img, @log]
mobcon.message.MessageHandler: Getting message from server
METHOD:  callI_ray [@img, @log]
```

Figure 5.18: MobRay Running on a MIDP Emulator and Part of its Execution Log

Figure 5.18 shows a screen shot of the MobRay application along with an extract of its execution log generated by applying the MobCon traceability attribute `@log` (§5.2.7). For example, the method `callF_patient` is modified by the screen concern transformer (as it is tagged by `@scr`). And of course, all methods effected by logging are modified by the traceability concern transformer `@log` (§5.2.7).

5.3 Extending the MobCon Framework

The plug-in architecture of MTE (§5.1.4) enables several types of extensions: (a) the existing plug-ins can be extended with more services for the MIDP assets they manage, (b) the existing plug-ins can be fully replaced with new MIDP plug-ins, or new plug-ins for new MIDP assets could be added, (c) an entirely new set of plug-ins to support a container for a new domain based on Java 1.4 can be introduced. In all the cases, the domain assets are modeled as attribute-based DSA, which removes the need to modify the grammar of the original language. In order to apply successfully the attribute-based DSA and to be able to extend MobCon the transformation details need to be understood. The transformation workflow should be chosen based on the semantics of the domain assets. Chapter §4 discussed all these concepts in a generic level based on the Tango framework. This section only explains details specific to the current implementation

of the MobCon framework, noting differences between MTE and Tango as necessary. More information can be found in the MobCon documentation [Mob04].

5.3.1 Workflow and Plug-in Metadata

A MobCon plug-in is packed as a Java JAR file whose structure is shown in Figure 5.19. The plug-in JAR file contains, among other files, a manifest file with custom entries that describes the plug-in meta-data. An example of a manifest file for the session management (`@ses`) plug-in is shown in Figure 5.20.

```
<session>
|- MANIFEST.MF
+- <jar>
   |- session.tag
   |- session.vm
   |- ...
```

Figure 5.19: The `@session` Transformer Organization

```
1  Manifest-Version: 1.0
2
3  name: Transformer
4  Transformer-Name: Session
5  TID: 04
6  IID: 01
7  File-Name: session.vm
8  Tag-File: session.tag
9  Server-File: SessionTrans
10
11 name: Dependencies
12 Use-Before: 02.01
13 Use-After:
```

Figure 5.20: The *MANIFEST.MF* file for the *Session* Plug-in

The manifest file contains several sections identified by the different `name:` labels in lines 3 and 11 of Figure 5.20. There is a user friendly transformer name in line 4. The transformation ID (TID) in line 5 identifies the domain concern that the plug-in transformer supports, in this case the session management. The next number, the plug-in instance ID (IID) is the number given to the actual implementation of this plug-in (§5.2.6). MobCon enables using more than one possible implementation for a given TID in an application, or more often, in different applications. The main Velocity script file of the plug-in transformer is specified in line 8. When the transformer generates also serve-side code (Java SDK 1.4 is used for server-side code), then the line 9 specifies the names for the generated server-side files. The dependencies section, in

lines 10 to 13, defines the default plug-in dependencies explained in section §4.1.5. While user-friendly names can be used, MobCob requires unique identifiers in the form TID.IID to be used in the dependency lists.

```
1  <flow>
2      <group>
3          <transformer>
4              <name>Screen</name>
5              <tid>02</tid>
6              <iid>01</iid>
7              <merger>
8                  <class>MixTemplateExample</class>
9              </merger>
10         </transformer>
11         <transformer>
12             <name>Mix Example</name>
13             <tid>07</tid>
14             <iid>01</iid>
15         </transformer>
16     </group>
17 ...
```

Figure 5.21: MobCon Dependency File

As explained in section §4.1.5, the global transformation workflow is calculated automatically in MobCon based on the local plug-in meta-information found in the *MANIFEST.MF* file. This frees the developers from having to create the total dependency graph manually. In the MIDP case, the existing plug-in dependencies are specified in their corresponding manifest files. If a new plug-in is added or an existing plug-in is modified or replaced, the plug-in manifest file must be edited to reflect the new order. MobCon supports also an application specific order, where each application can override the workflow order by modifying the `lib\plugins\depend.xml` dependency file generated by MobCon inside the application directory. For example, the dependency file of Figure 5.21 has been modified to support a custom combination of the output of the *Screen* transformer (line 4) and a custom *Mix Example* transformer (line 12), by using a custom mixer `MixTemplateExample` transformer (line 8). A mixer transformer must implement the `MixTemplate` interface of Figure 5.22. Chapter §4 discussed how this concept is generalized in Tango to mix an arbitrary number of classes at the same time.

Apart of the dependency meta-data a plug-in file contains also a tag dictionary file (`.tag`). The tag dictionary is a Java property file that maps the attribute names used in source code to those used by the plug-in internally. This allows developers to customize the attribute names used in code. For example, instead of `@dp` they could use `@dataPersistence` in code. The plug-in implementation treats the attribute names as string resources: `$class.getTag($tagDic.getTag("dp"))`. The trace log will use whatever name the user choses for the attributes.

```
1  public interface MixTemplate {
2    public ClassTemplate
3       mixClasses(ClassTemplate ct1, ClassTemplate ct2);
4    public MethodTemplate
5       mixMethods(MethodTemplate mt1, MethodTemplate mt2);
6  }
```

Figure 5.22: MobCon MixTemplate Interface

5.3.2 Transformation Details

MobCon plug-ins use the Apache Velocity [Vel03] script engine to manipulate the code. The default mode for the Velocity engine is to output text using the Java `System.out` object. The MTE modifies the Velocity initialization not to output code directly. Instead, MobCon makes accessible for modification to the Velocity scripts, a set of GAAST-like class template API (CT-API) as explained in section §4.1.1. Working with the CT-API facilitates many aspects of the transformer implementation, that would need to be expressed as text Velocity templates otherwise. The vertical attribute-driven transformer modularization explained in section §4.1.6 is not supported in MobCon. For an example of how a transformation can be implemented in MobCon, consider the attribute-decorated code of Figure 5.23.

```
1  /**
2   *@scr
3   *@dp
4   */
5   public class Test{
6   . . .
7  /**
8   * @dp.access
9   * @scr.store
10  */
11  private String id;
```

Figure 5.23: Example MIDP Input Code

Figure 5.24 shows a part of a transformer implemented in Velocity that processes the `dp.access` tag in Figure 5.23, referred by the name *"accessible"* in the tag dictionary (line 3). A Velocity macro `dp_meth_get` (lines 9, 16) is used to generate the code in line 5. Inside the macro, a class template object that represents a method is created and mapped as a Velocity object (called a *bean*) by using the vDoclet tool (line 10 in Figure 5.24). The method template object is then modified by the macro to customize it to model an accessor method.

The generated code for the accessor method looks as in Figure 5.25, where a method named

```
1  #foreach ($field in $class.fields)
2    #set($tag = false)
3    #set($tag = $field.getTag($tagDic.getTag("accessible")))
4    #if ($tag)
5     #dp_meth_get($field.type $field.name)
6    #end
7  . . .
8
9  #macro(dp_meth_get $type $name)
10  #set($MT = $vdoclet.makeBean("mobcon.ct.MethodTemplate"))
11  $MT.setAccess("public")
12  $MT.setType("$type")
13  $MT.setName("get$StringUtils.capitalizeFirstLetter($name)")
14  $MT.addEnd("return $name;")
15  $CT.addMethod($MT, $tagDic.getPrefix())
16 #end
```

Figure 5.24: Example Velocity Script for Processing Figure 5.23

`getId()` is generated. Chapter §4 discussed how attribute-based transformers can be further modularized to scale the transformation.

```
1  . . .
2    public String getId(){
3      return id;
4    }
5  . . .
```

Figure 5.25: Example Output Code for Example of Figure 5.23

5.4 Related Work

Section §2.2.2 discussed the J2EE containers and explained that EJB [EJB04] is moving toward a simpler programing model based on attributes. The motivation to use attributes in EJB is not to support a low-cost DSA mechanism. Rather, a predefined specific set of attribute-based DSA is used as a clear and uniform way to support the EJB programming model for enterprise containers. The current version of the latest EJB specification is not fully-completed and leaves the discussion open for feedback in some places, on what is the best way to support a concern, or how to express it better with attributes. MobCon is an open attribute-driven framework for supporting arbitrary attribute-based containers. MobCon is specialized with a set of plug-ins for MIDP applications. The set of attributes that can be used to support a product-line is left open and MobCon enables experimentation. The transformation workflow is customizable and can be

modified on a per application basis. The source code parsing engine of MobCon supports a Java 1.4 compatible syntax, similar to the one found in MIDP and has not support for Java 1.5 style annotations[14].

The SmallComponents [Voe03b] approach is based on generative techniques and visual modeling to introduce a logical container abstraction. SmallComponents address embedded systems, where the size of the middleware must be optimized with respect to specific application needs. To achieve this goal, an adaptable container abstraction is used as a replacement for the middleware itself. SmallComponents is based on graphical domain-specific languages (UML), rather than on attributes. The SmallComponents approach addresses embedded and real-time systems, where generative approaches are superior to other code abstraction techniques. In MobCon, the container abstraction is not seen as a replacement for the mobile middleware. It rather augments the MIDP middleware with convenient software abstractions to make the MIDP applications easier to develop and maintain. Attributes support a modeling-like view, directly at the source level (§2.1.5).

There are many source-to-source transformation and meta-programming approaches and frameworks available [CE00]. The MobCon transformer framework is specialized for building attribute-driven source code transformers, which may need to be maintained often, to reflect changes in the underlying middleware. Unlike other Velocity-based [Vel03] transformers [Vel04, And03], MobCon transformers do not use Velocity directly to output code, but rather work on the specialized GAAST-like API (§3). This makes it easier to support code generation in the case of cascaded transformers. The effects of attribute-driven transformation and their relation with other transformation approaches were discussed in chapter §4.

Adaptive proxies [FGBA96, FGCB98] enable transparent access of server-side services from a mobile client. They stand between the server-side application and a mobile client application, and dynamically adapt the server data requested by the mobile client in order to fit to the capabilities of the mobile client device. The adaptive proxy can reside in the same machine as other server-side services, or in a separate machine. MobCon automates the technical concerns of the adaptive proxy related to the mobile application. The server-side part of a mobile container transforms the data to prepare them for the mobile device. MobCon generates the server-side part based on the requirements for the container concerns support in the client-side part. These services are used then by the adaptive proxy functionality.

Aspect-oriented programming (AOP) [KLM+97] techniques discussed in section §2.4 can also be used to support attribute-driven transformations [KM05], and to achieve independence from specific middleware [BCH03]. AOP tools, e.g., AspectJ [Lad03, KHH+01] can be used in two ways to support transformations enabled by MobCon. (a) AOP-style factorizations based on the description of the component joinpoints as pointcuts (implicit hooks [Aßm03]) can be used.

[14] The AST representation manager could be replaced, if needed, in MonCon to enable Java 1.5 support. The new AST needs to be mapped similarly to vDoclet [vDo03] tool. These issues are outside the focus of this book. The current MIDP Java dialect syntax is very similar to the Java 1.4.

This is the only available technique in AspectJ before the support for Java 1.5 annotations. This indirect style of programming requires redefining the pointcuts for every component that needs container support, in every specific application, in order to match the advice code. (b) AOP tools can be used to work upon attributes (explicit hooks [Aßm03]) as a generic meta-programming engine (§3.5.3). AOP techniques require also some form of run-time support (added statically by AspectJ) to support the pointcut context management. This infrastructure would be appended to the infrastructure code required by the mobile container. The duplicated infrastructure would result in more overhead than with other meta-programing techniques that maintain the node selection context explicitly during the transformation.

AOP techniques could be also supported completely at run-time with additional run-time support. In [YCS+02] dynamic AOP techniques are used to implement context adaptable sockets. In [PAG03] *spontaneous containers*, based on dynamic AOP, are used along with Jini [New00] to adapt mobile applications to several environment services. These approaches support for dynamic services. Dynamic AOP techniques are too heavy for MIDP applications and require a more powerful class of devices. For example, MIDP currently does not support programmable code downloading and has no reflection capabilities[15], which makes such techniques not applicable. For this reason, the MobCon approach presented here deals only with static transformations of the J2ME MIDP 2.0 applications.

5.5 Chapter Summary

MobCon is a framework for automating technical concerns of MIDP 2.0 applications built upon the concepts introduced in this book. J2ME MIDP applications exhibit repeated cross-cutting functionality that can be factored out from one or more applications. This common functionality can be parameterized and made part of product-line to support MIDP applications.

It is preferable to express the MIDP domain assets declaratively in source code. The programmers concentrate on the specific application functionality and express the cross-cutting concerns declaratively. Attribute-based DSA provide a low-cost mechanism for supporting a customizable declarative programming model. A GAAST-like representation of the MIDP Java source is realized based on several open-source tools. These tools allow manipulation of JavaDoc-like decorated source code.

MobCon's programming model is based on the decoration of program entities with attributes. The variability of the technical MIDP concerns is modeled as attribute families. Each family is specialized for a given concern, and contains nested levels of sub-families to express different aspects of the variability. Finally, attribute arguments are used to parameterize the attribute families. The MobCon Transformation Engine (MTE) is organized around the attribute families and employs the different transformation units as plug-ins. MTE contains functionality

[15] Therefore, Jini [New00] cannot be used.

to drive the transformation workflow based on the plug-in dependencies. The template method pattern is used to separate the generated and manually entered code.

Mobile containers are a special kind of client container specialized to automate the concerns of product-lines for mobile applications. A mobile container automates not only the software running in the client mobile device, but also the related part of software into the server-side. Mobile containers offer an architectural abstraction to organize the domain assets of MIDP product-lines.

Several attribute sets are predefined and automate concerns related to J2ME MIDP, e.g., the persistence of data in the mobile device or across the network. The implemented concerns were selected to be representative of common MIDP tasks, which show up repetitively in many MIDP applications. A case study based on an application for medical X-Ray diagnostics was shown. The addressed MIDP concerns are injected automatically into the X-Ray application, resulting in less code, focusing only to the specific functionality.

The concerns currently addressed by MobCon are by no means a complete set covering every possible MIDP application. The MobCon attribute-driven transformation engine is extensible and allows new plug-ins to be defined and integrated into a container. Plug-ins represent one or more transformers specialized for a specific concern. New plug-ins could add more MIDP services, or provide support for a totally different domain. The generative framework itself is written in Java and is independent of the plug-ins functionality. The transformation control flow is determined based on the declared plug-in workflow preferences, and can be overwritten manually for specific cases.

Chapter 6

Summary and Outlook

I would have made the book shorter, but I did not have any more time.

E. H. Connell, Elements of Abstract and Linear Algebra, 1999

6.1 Summary

This book has been concerned with the problem of finding better and faster ways to automate and reuse software in mobile device applications[1]. The focus has been in (a) developing easy to implement techniques for organizing the common domain functionality in mobile product-lines that (b) enable declarative reuse of the domain functionality in mobile applications. The problem has been addressed (Figure 6.1): (1) by introducing attribute-based DSA supported by GAAST-enabled languages, (2) by having a structured way to interpret attributes with modularized attribute-driven transformers and finally, and (3) by using a software container abstraction to organize the domain assets of mobile application product-lines.

After an introduction to the main topics in chapter §1, **chapter §2** motivated product-lines support for the development of mobile applications and presented mechanisms for implement-

[1] Mobile applications, for short.

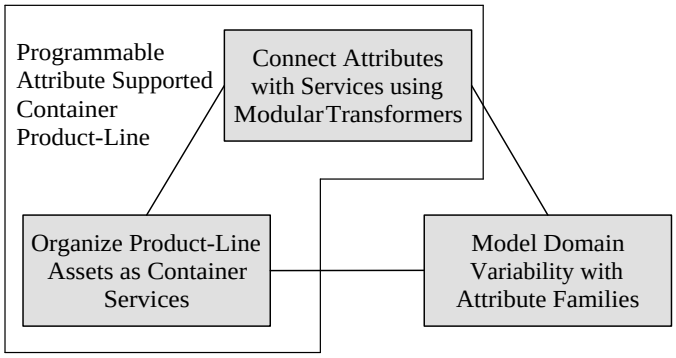


Figure 6.1: Attribute-Supported Container-Based Product-Lines

ing mobile product-lines. Chapter §2 proposed to organize mobile product-lines with mobile containers, and to support such containers with DSA at the source code level.

- Automated product-lines are needed to support mobile applications and reuse the common functionality. The mobile product-lines factor out the common functionality of a family of applications and inject the common functionality transparently into a specific application. Automation helps to deal with the cross-cutting nature of the common functionality that is present in more than one application.
- There are several variability mechanisms for supporting product-lines. Declarative domain-specific abstractions (DSA) supported at the programming language level help to make the domain architecture explicit and to preserve the domain model in the source code. DSA blur the distinction between a modeling phase and the coding phase, and offer support for the traceability of domain concerns in specific applications.
- A software container serves as an architectural abstraction to organize the domain assets. Containers are known from the domain of server-side enterprise applications. When combined with attribute-based DSA (§3), the container offers an architectural abstraction to connect the implementation of the domain assets with the declarative attribute constructs in code.
- There are many ways to implement containers. Invasive generative containers are better suited for mobile product-lines compared to non-invasive techniques. Invasive containers supported by declarative domain-specific abstractions (DSA) offer more automation and could introduce easy domain-specific optimizations.
- Aspect-oriented programming (AOP) engines could be used as generic invasive frameworks to implement attribute-driven transformations. Specialized transformation engines

are, however, preferable for specific domains. The specifics of attribute-based DSA are utilized to create specialized attribute-driven transformers. AOP and DSA can coexists and complement each-other (§3.5.3). DSA support an explicit programming model and enable vertical automation. AOP-style modularization can be applied over the existing DSA language constructs.

Chapter §3 focused on the aspects of attribute-enabled languages and the effects of *attribute enabled programming* (AEP) in the software development process. GAAST was introduced as a common API of an extensible language workbench, intended to support attribute-based DSA. Several aspects of using attribute-enabled languages in the software development process were investigated.

- Attribute-based DSA decrease the cost of introducing custom DSA in a language, making attributes preferable for supporting iterative product-lines. Attributes expose a uniform programming model and do not require grammar modifications. Attribute-based DSA selectively customizes the semantics of existing components or language constructs.
- Domain variability is modeled as attribute families. Each attribute family represents a domain asset of interest that needs to be automated. Inside each family, a name space organization is applied to nest attribute sub-families that define further specialization of the modeled asset. Attribute parameters model the variability ranges of individual attributes.
- AEP is attractive for mapping MDA UML class models to source code. Unlike other approaches, e.g., marking interfaces and pseudo-syntactic marking, attribute programming fully preserves the architecture of the model in code. Attribute programming serves as an extension to the overall MDA transformation process. Attributes supported by GAAST-enabled languages move the MDA transformation concerns at the language level and offer a single transformation system to the programmers.
- UML stereotypes and tagged values model only a subset of the AEP development scenarios. The full range of attribute-based design possibilities in UML class diagrams can be modeled in different ways. All these alternatives extend the UML class diagrams notation. None of them is better suited in all cases, reflecting the wide range of design possibilities that can be modeled by attributes.
- Explicit attributes in the programming language level extend the semantics of the language meta-model without the need to modify or maintain its parsing tools. This results in low-cost language support for a wide range of EDSL that model the product-line abstractions.
- Several modern general-purpose languages, e.g., NET and Java, offer support for attributes as part of their language technology. An arbitrary number of custom attributes can be introduced. Several API-s, e.g., .NET CodeDom and Reflection API-s, process the code

entries annotated with attributes either before compilation or at run-time. Run-time support is enabled via the Reflection API, which uses the structural information saved in the binary meta-data.

- The attribute processing API-s found in .NET and Java do not cover the full range of possible attribute-driven transformation scenarios. Several attribute-based transformations could be represented uniformly despite the origin of the attribute-decorated AST. Generalized and Annotated AST (GAAST) enabled languages offer uniform attribute support for processing source code, or binaries after compilation, or at run-time. When GAAST is supported as part of the language technology no third-party parsing and meta-programming tools are needed. Third-party frameworks add accidental complexity to a product-line. When GAAST is not supported in a language, several aspects of GAAST are easily emulated with any meta-programming tool, keeping the cost of introducing GAAST and attribute-based DSA at a minimum.
- The power of GAAST languages does not stand in the expressiveness of grammars they generate, but in the transformations they apply to an existing core grammar. GAAST makes the language meta-model enhancement implicit. GAAST reduces the meta-model maintenance costs, and removes the need to rely on external third-party transformation frameworks.
- Attributes are cheap to introduce and could be easily over-used. They should not be applied to model semantics that could be expressed in simpler ways with existing language constructs. The scope of attribute annotations needs to be carefully chosen based on the characteristics of the domain assets modeled with attributes.

Chapter §**4** addressed issues related to attribute-driven variability and transformations. Modular attribute-driven transformations implement attribute-based DSA in a structured way. Transformer modularization reduces the overall cost of supporting product-lines with attributes and enables reuse of the individual transformation units.

- The properties of the addressed domain, attribute-based DSA for mobile product-lines, are utilized to modularize transformers. The transformation process is organized in separate transformer units based on the domain assets.
- Inner attributes coordinate between the individual transformation units. Attribute-driven transformations treat inner attributes similarly to other explicit attributes. The similarity results in a uniform model for expressing the transformation composition semantics and enhances traceability.
- The relations between the modeled domain assets specify the transformation workflow. Dependency lists are parsed to automatically create the transformation dependency graph. Conflicts are solved using a workflow language.

- The properties of the specific language meta-model are utilized to vertically modularize attribute-driven transformers. The transformation strategy is structured according to the structural nesting in the meta-model hierarchy. Attribute semantics are separated between the layers. Layering is enforced with special syntax. The operations in each layer are made declarative with specialized strategy operations.
- Meta-attributes express transformation cross-cutting concerns (CCCs) declaratively in a GAAST-enabled language. Meta-attributes are processed separately outside the individual transformers with generic tools. When an attribute is defined, its definition is decorated with meta-attributes representing the generic concerns that need to be validated for that specific attribute.
- Attribute dependencies are an important attribute transformation concern that is expressed declaratively using meta-attributes. Attribute dependencies enable expressing the valid attribute usage context with regard to other attributes in a way similar to representing grammar parser rules. The semantics of the dependency attribute are checked with specialized tools.

Chapter §**5** summarized and evaluated the concepts presented in this book by presenting MobCon, a generative mobile container framework specialized for addressing J2ME MIDP non-functional application concerns.

- J2ME MIDP applications form a domain of interest that exhibits redundant behavior in the form of cross-cutting non-functional concerns. An extended product-line for supporting MIDP domain assets, based on the technology presented in this book, was created. MobCon was used to study the effects of attribute-based DSA and attribute-drive transformations, serving as a basis for the generalization of the concepts presented in the previous chapters.
- A GAAST-like representation was implemented for the Java dialect supported by the MIDP. It was combined with a customizable transformation framework to manage the workflow of the transformation process. Transformation units are modeled as different plug-ins based on the attribute families that cover the addressed MIDP concerns.
- The container abstraction constitutes a centralized point to support variability in a mobile software product-line. Mobile containers are a specialization of the container architectural abstraction for supporting automated mobile product-lines. Mobile containers deal with client-side automation issues. Their functionality and service support extends from the mobile device to the server-side. The container impersonates the mobile client to the rest of the environment services.
- Several MIDP concerns, e.g., data persistence, screen management, image adaptation, data encryption and networking, are organized as part of a mobile container. The container

is modeled as a set of MobCon plug-ins specialized for MIDP. Each plug-in transforms an attribute family, and generates also container infrastructure code to be placed on the server-side as needed.

- The MobCon framework is designed to be extensible. New plug-ins can be added to support new MIDP concerns or existing plug-ins can be modified. New domains could also be supported by a new set of plug-ins. MobCon uses the plug-in meta-data to calculate the transformation workflow, and enables the developers to customize the workflow for specific applications.

> *There is no "one right way" to design and build all systems. ..., we must plan for a series of successes.*
>
> B. Stroustrup, The C++ Programming Language, Third Edition, Addison-Wesley, 1997

6.2 Limitations and Outlook

As discussed in chapter §3, attribute-based DSA model only a limited subset of EDSL. Attributes by definition decorate existing entities. They cannot implement arbitrary EDSL or DSL constructs. Attributes are used in this book to declaratively modify the semantic of OO components. Only static decorations of structural elements are utilized to implement the MobCon container framework.

Attribute transformations addressed in this book (§4) are specialized to support OO mobile device software product-lines. The modularization is dependent on the domain assets and on the orthogonality of the assets. Only the transformations applied over the structural decoration of OO components with attributes are explored. The functionality of the methods is presented internally as a series of code blocks. A finer grained representation may be needed for other domains.

The solution for supporting mobile software product-lines was investigated in details. The introduced concepts were evaluated by various prototypes. There are also several open areas for further exploration.

- **Further improvement of different aspects of the introduced technologies.** One area for future work would be to investigate of the impact of the GAAST-like organization API for representing the annotated source, binary, and run-time in the context of a specific compiler. That is, to find out the best organization for a compiler with integrated GAAST

support. Such a compiler needs to have an API-like front-end accessible from the programming language level. It can then be used to support transparently GAAST-based transformations. Only the static aspects of GAAST-based transformations were explored in this book. Depending on level of reflection supported in the language, GAAST could serve also as an advanced reflective meta-programming system for supporting dynamic attribute-driven transformations.

Another area for future work is to add more MIDP services to MobCon to support other MIDP related tasks. More variability could also be supported for the existing MIDP domain assets. Given the extensibility of MobCon, new features can be added to the current set of MIDP plug-ins to extend the MIDP support. The combination of attribute-based DSA with wizards and visual models could also be explored.

It is also possible to combine the features of Tango and MTE prototypes into a single attribute-driven transformation engine. The specific transformation issues were properly investigated in Tango and MTE. A combined transformation engine could also support a bigger number of specialized transformer operations. It could also enable the developers to distinguish between changing a class in place and creating an adapter or decorator [GHJV95] for a class. The declarative common operations, e.g., declaration and enforcement of attribute dependencies, part of the ADC prototype (§4), could also be generalized and made part of the special operations supported by a unified framework. An extensible plug-in architecture to support meta-attributes could be created.

- **Areas for future research.** An interesting topic for future research is to investigate the applicability of the technology developed in this book to another domain, for example, to automate web service programming concerns. Some of the examples given in the thesis to illustrate various aspects of GAAST-enabled languages and attribute dependencies were motivated from the domain of web services. In the large, it would be also very interesting to evaluate what kind of domains and what types of product-lines could better benefit from attribute-based DSA supported containers.

Appendix A

MobCon Generated Code for "Hello World" MIDP Example

This appendix lists the complete code that MobCon generates for the *Hello world* MIDP example of Figure 5.12, discussed in section §5.2.2.

```
1  import java.util.Hashtable;
2  import javax.microedition.lcdui.*;
3  import javax.microedition.midlet.*;
4  import javax.microedition.midlet.MIDletStateChangeException;
5  import mobcon.message.*;
6  import mobcon.storeables.*;
7
8  public class MobApp extends AbstractMobApp
9  {
10         public MobApp()
11         {
12                 super();
13         }
14 } //EOC
```

Figure A.1: MobCon Generated Code for `MobApp`

```
1  public abstract class AbstractMobApp extends MIDlet implements CommandListener {
2          protected Command exitCommand;
3          protected Display display;
4          private Form form;
5          public static String CID = "6f60ea1de7a3215960b1209c817dad99";
6          protected String firstForm = "form";
7          protected String[] listElements;
8          protected TextBox messageBox;
9          private TextField textField;
10
11         public  AbstractMobApp() {
12                 exitCommand = new Command("Exit", Command.EXIT, 1);
13                 display = Display.getDisplay(this);
14         }
15
16         public void callForm() {
17                 form = new Form("Test");
18                 form.addCommand(exitCommand);
19                 form.setCommandListener(this);
20                 callTextField();
21                 form.append(textField);
22                 display.setCurrent(form);
23         }
24
25         public void callMessageBox( String label,  String text) {
26                 messageBox = new TextBox( label, text, 256, TextField.ANY );
27                 display.setCurrent(messageBox);
28         }
29
30         public void callTextField() {
31                 String text = "";
32                 text = "Hello World";
33                 textField = new TextField("First Application", text, 256, TextField.ANY);
34         }
35
36         public void callTextField( String text) {
37                 textField = new TextField("First Application", text, 256, TextField.ANY);
38         }
39
40         public void commandAction( Command command,  Displayable screen) {
41                 if (command == exitCommand) {
42                         destroyApp(false);
43                         notifyDestroyed();
44                 }
45         }
46
47         public void destroyApp( boolean unconditional) { }
48
49         public void pauseApp() { }
50
51         public void startApp() {
52                 viewDisplay(firstForm);
53         }
54
55         public void viewDisplay( String displayName) {
56                 if(displayName.equals("form")) callForm();
57         }
58 } //EOC
```

Figure A.2: MobCon Generated Code for `AbstractMobApp`

Bibliography

[ABHR99] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A Core Calculus of Dependency. *26th ACM SIGPLAN-SIGACT on Principles of Programming Languages (POPL)*, pages 147–160, 1999.

[ACKM04] G. Alonso, F. Casati, H. Kuno, and V Machiraju. *Web Services - Concepts, Architectures and Applications*. Springer, 2004.

[Agr04] A. Agrawal. A Formal Graph-Transformation Based Language for Model-to-Model Transformations. *PhD Dissertation, Vanderbilt University, Dept of EECS*, 2004.

[AK01] C. Atkinson and T. Kuehne. The Essence of Multilevel Metamodeling. *4th International Conference on the Unified Modeling Language (UML)*, 2001.

[AK03] C. Atkinson and T. Kuehne. Aspect-Oriented Development with Stratified Frameworks. *IEEE Software, Volume 20, Number 1*, pages 81–89, 2003.

[AM00] G. D. Abowd and E. D. Mynatt. Charting Past, Present, and Future Research in Ubiquitous Computing. *ACM Transactions on Computer-Human Interaction (TOCHI). Volume 7, Issue 1*, pages 29–58, 2000.

[And03] AndroMDA Homepage. `http://www.andromda.org/`, 2003.

[Asp04] AspectBench Compiler Homepage. `http://aspectbench.org/`, 2004.

[ASS96] H. Abelson, G. J. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, 1996.

[Aßm94] U. Aßmann. On Edge Addition Rewrite Systems and Their Relevance to Program Analysis. *5th Workshop on Graph Grammars and Their Applications to Computer Science, LNCS 1073*, pages 321–335, 1994.

[Aßm96] U. Aßmann. How to Uniformly Specify Program Analysis and Transformations. *6th International Conference on Compiler Construction, LNCS 1060*, 1996.

[Aßm99] U. Aßmann. *A Conceptual Comparison of Current Component Systems*. Software Best Practice OO Series, Springer, 1999.

[Aßm00] U. Aßmann. Graph Rewrite Systems for Program Optimization. *In AMC Transactions on Programming Languages and Systems (TOPLAS), Volume 22, 4*, 2000.

[Aßm03] U. Aßmann. *Invasive Software Composition*. Springer-Verlag, 2003.

[ASU88] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1988.

[Att02] Attrib4J Homepage. `http://attrib4j.sourceforge.net/`, 2002.

[BB02] G. Banavar and A. Bernstein. Software Infrastructure and Design Challenges for Ubiquitous Computing Applications. *Communications of the ACM, Volume 45, Issue 12, SPECIAL ISSUE: Issues and challenges in ubiquitous computing*, pages 92–96, 2002.

[BC90] G. Bracha and W. Cook. Mixin-Based Inheritance. *Conference on Object-Oriented Programming: Systems, Languages, and Applications / European Conference on Object-Oriented Programming (OOPSLA/ECOOP)*, pages 303–311, 1990.

[BCH03] R. Bodkin, A. Colyer, and J. Hugunin. Applying AOP for Middleware Platform Independence. *Practitioner Reports, Aspect-Oriented Software Development Conference (AOSD)*, 2003.

[BCS00] D. Batory, R. Cardone, and Y. Smaragdakis. Object-Oriented Frameworks and Product Lines. *1st Software Product Line Conference (SPLC)*, pages 227–247, 2000.

[BCVM02] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit Programming. *Aspect-Oriented Software Development Conference (AOSD), ACM Press*, pages 10–18, 2002.

[BGJ99] S. Berner, M. Glinz, and S. Joos. A Classification of Stereotypes for Object-Oriented Modeling Languages. *UML '99 - The Unified Modeling Language: Beyond the Standard, LNCS*, 1723:249 – 264, 1999.

[BH02] J. Backer and W. C. Hsieh. Maya: Multiple-Dispatch Syntax Extension in Java. *Conference on Programming Language Design and Implementation (PLDI), ACM SIGPLAN Notices, Volume 37, Issue 5*, pages 270–281, 2002.

[BHST04] Y. Bontemps, P. Heymans, P.-Y. Schobbens, and J.-C. Trigaux. Semantics of FODA Feature Diagrams. *3rd Software Product Line Conference (SPLC'04), Software Variability Management for Product Derivation - Towards Tool Support*, 2004.

[Big00] T. J. Biggerstaff. A New Control Structure for Transformation-Based Generators. *6th International Conference on Software Reuse (ICSR)*, 2000.

[BJMvH00] D. S. Batory, C. Johnson, B. MacDonald, and D. von Heeder. Achieving Extensibility Through Product-Lines and Domain-Specific Languages: A Case Study. *6th International Conference on Software Reuse (ICSR)*, pages 117–136, 2000.

[BK87] J. Banerjee and W. Kim. Semantics and Implementation of Schema Evolution in Object-Oriented Databases. *ACM SIGMOD, International Conference on Management of Data*, 1987.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis. JTS: Tools for Implementing Domain-Specific Languages. *5th International Conference on Software Reuse (ICSR), IEEE Press*, pages 143–153, 1998.

[Bod04] E. Bodden. A Lightweight LTL Runtime Verification Tool for Java. *ACM Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, 2004.

[Bol03] D. Boling. *Programming Microsoft Windows CE .NET, Third Edition*. Microsoft Press, 2003.

[Bor03] Borland TogetherJ Homepage. `http://www.borland.com/together/`, 2003.

[Bos00] J. Bosch. *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*. Addison-Wesley, 2000.

[Bou04] Bouncy Castle Crypto API. `http://www.bouncycastle.org/`, 2004.

[Box00] D. Box. House of COM. *MSDN Magazine, December Issue*, 2000.

[BP01] J. Bachrach and K. Playford. The Java Syntactic Extender: JSE. *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), ACM SIGPLAN Notices, Volume 36, Issue 11*, pages 31–42, 2001.

[BPL01] A. Bakshi, V. K. Prasanna, and A. Ledeczi. MILAN: A Model Based Integrated Simulation Framework for Design of Embedded Systems. *Workshop on Languages, Compilers, and Tools for Embedded Systems*, 2001.

[BPM02] I. D. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program Transformations for Practical Scalable Software Evolution. *International Workshop on Principles of Software Evolution*, 2002.

[BPSV01] J. R. Burch, R. Passerone, and A. L. Sangiovanni-Vincentelli. Using Multiple Levels of Abstractions in Embedded Software Design. *1st International Workshop on Embedded Software*, 2001.

[BS99] D. Blostein and A. Schürr. Computing with Graphs and Graph Rewriting. *Software - Practice and Experience*, 29(3):1–21, 1999.

[BST+94] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca Model of Software-System Generators. *IEEE Software.*, 11(5):89–94, 1994.

[C#02] C# Programmer's Reference: Introduction to Attributes. *MSDN:* `ms-help://MS.VSCC/MS.MSDNVS/csref/html/vclrfintroductiontoattributes.htm`, 2002.

[Cal03] P. W. Calnan. EXTRACT: Extensible Transformation and Compiler Technology. *Master of Science Thesis, Worcester Polytechnic Institute,* `http://www.wpi.edu/Pubs/ETD/Available/etd-0429103-152947/`, 2003.

[CDK01] G. Coulouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design, 3rd Edition.* Addison Wesley, 2001.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming.* Addison-Wesley, 2000.

[CE02] I. Crnkovic and M. Larsson (Editors). *Building Reliable Component-Based Software Systems*. Artech House, 2002.

[Cep04] V. Cepa. Implementing Tag-Driven Transformers with Tango. *8th International Conference on Software Reuse (ICSR) - LNCS 3107*, pages 296–307, 2004.

[CF05] R. Cobbe and M. Felleisen. Environmental Acquisition Revisited. *32nd ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pages 14–25, 2005.

[CGT89] S. Ceri, G. Gottlog, and L. Tanca. What You Always Wanted to Know About Datalog. *Logic Programming and Databases*, 1989.

[CH03] K. Czarnecki and S. Helsen. Classification of Model Transformation Approaches. *2nd Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Workshop Generative Techniques in the context of MDA*, 2003.

[Cho04] Choosing a UML Modeling Tool. `http://www.objectsbydesign.com/tools/modeling_tools.html`, 2004.

[CK05] V. Cepa and S. Kloppenburg. Representing Explicit Attributes in UML. *7th International Workshop on Aspect-Oriented Modeling (AOM)*, 2005.

[CK06] V. Cepa and S. Kloppenburg. International Conference on Innovative Views of .NET Technologies (IVNET'06). *Journal of Computing and Information Technology*, 2006.

[CM04] V. Cepa and M. Mezini. Declaring and Enforcing Dependencies Between .NET Custom Attributes. *3rd International Conference on Generative Programming and Component Engineering (GPCE'04)*, 2004.

[CM05a] V. Cepa and M. Mezini. Language Support for Model-Driven Software Development. *(Editor M. Aksit) Special Issue Science of Computer Programming (Elsevier) on MDA: Foundations and Applications Model Driven Architecture*, 2005.

[CM05b] V. Cepa and M. Mezini. MobCon: A Generative Middleware Framework for Java Mobile Applications. *38th Hawaii International Conference on System Sciences (HICSS-38)*, 2005.

[CMA94] L. Cardelli, F. Matthes, and M. Abadi. Extensible Syntax with Lexical Scoping, Digital Systems Research Center. Technical Report 121, 1994.

[CN02] P. Clements and L. Northrop. *Software Product Lines*. Addison-Wesley, 2002.

[CN04] Sh. Chiba and K. Nakagawa. Josh: An Open AspectJ-like Language. *3rd Conference on Aspect-Oriented Software Development(AOSD)*, pages 102–111, 2004.

[Cor04] J. Cordy. TXL - A Language for Programming Language Tools and Applications. *4th Workshop on Language Descriptions, Tools and Applications, (LDTA)*, 2004.

[DCO02] Distributed COM (DCOM) Technology Homepage. `http://www.microsoft.com/com/tech/DCOM.asp`, 2002.

[Del02] Delivering Hot Content with Confirmation - Nokia Technology Paper. `http://www.nokia.com/downloads/aboutnokia/press/pdf/CDT_paper_v4_A4n.pdf`, 2002.

[DJ90] N. Dershowitz and J. P. Jouannaud. Rewrite Systems. *Handbook of Theoretical Computer Science, Volume B, Chapter 6, Elsevier*, 1990.

[Dol99] M. Dolgicer. Building a Middleware Platform. *Component Strategies, Number 3*, 1999.

[Don02] J. Dong. UML Extensions for Design Pattern Compositions. *Journal of Object Technology*, 1(5):151–163, 2002. `http://www.jot.fm/issues/issue_2002_11/article3`.

[DSC03] CBOP DSCT, IBM. MOF Query / Views / Transformations. *Initial Submission, OMG Document*, 2003.

[Dug00] D. Duggan. A Mixin-Based, Semantics-Based Approach to Reusing Domain-Specific Programming Languages. *14th European Conference on Object-Oriented Programming (ECOOP)*, pages 179–200, 2000.

[EBM05] C. Enders, A. Butz, and A. MacWilliams. A Survey of Software Infrastructure and Frameworks for Ubiquitous Computing. *Mobile Information Systems Journal, IOS Press*, 2005.

[Ecl05] Eclipse AspectJ Development Tools Homepage. `http://eclipse.org/ajdt/`, 2005.

[ECM02] ECMA-262 ECMAScript Language Specification. `http://www.ecma-international.org/publications/standards/Ecma-262.htm`, 2002.

[Eff02] Efficient MIDP Programming - Nokia Technology Paper. `http://66.77.83.18/downloads/nokia/documents/`, 2002.

[EJB03] *Enterprise JavaBeans Specification, Version 2.1*. Sun Microsystems, 2003.

[EJB04] *Enterprise JavaBeans Specification (Annotations), Version 3.0*. Sun Microsystems, 2004.

[EMOS04] M. Eichberg, M. Mezini, K. Ostermann, and T. Schäfer. Xirc: A kernel for cross-artifact information engineering in software development environments. *11th Working Conference on Reverse Engineering*, 2004.

[Ern99] E. Ernst. *gbeta - a Language with Virtual Attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD Thesis, University of Aarhus, 1999.

[ET96] H. Ehrig and G. Taentzer. Computing by Graph Transformation: A Survey and Annotated Bibliography. *BEATCS: Bulletin of the European Association for Theoretical Computer Science*, 59, 1996.

[Ewa01] T. Ewald. *Transactional COM+: Building Scalable Applications*. Addison-Wesley, 2001.

[FGBA96] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to Network and Client Variability via On-Demand Dynamic Distillation. *7th Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[FGCB98] A. Fox, S. Gribble, Y. Chawathe, and E. Brewer. Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives, Special issue of IEEE Personal Communications on Adaptation, 1998.

[Fin96] R. A. Finkel. *Advanced Programming Language Design*. Addison-Wesley, 1996.

[FKF98] M. Flatt, S. Krishnamurthi, and M. Felleisen. Classes and Mixins. *25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 171–183, 1998.

[FMG02] G. Frick and K. D. Mueller-Glaser. Generative Development of Embedded Real-Time Systems. *European Conference on Object-Oriented Programming (ECOOP), Workshop on Generative Programming*, 2002.

[Fow99] M. Fowler. *Refactoring Improving the Design of Existing Code*. Addison-Wesley, 1999.

[Fow04] M. Fowler. Inversion of Control Containers and the Dependency Injection Pattern. `http://martinfowler.com/articles/injection.html`, 2004.

[Fow05] M. Fowler. Language Workbenches: The Killer-App for Domain Specific Languages. `http://martinfowler.com/articles/languageWorkbench.html`, 2005.

[FPB87] Jr. F. P. Brooks. No Silver Bullet - Essence and Accidents of Software Engineering. *Computer Magazine, First published in Information Processing 1986, ISBN 0444-7077-3, H. J. Kugler, Ed., Elsevia Science Publishers B.V*, 1987.

[Fra03] D. S. Frankel. *Model Driven Architecture - Applying MDA to Enterprise Computing*. Wiley, 2003.

[GB04] E. Gamma and K. Beck. *Contributing to Eclipse*. Addison-Wesley, 2004.

[Gen02] Gen <X> Homepage. `http://www.develop.com/genx/`, 2002.

[GH02] A. Granicz and J. Hickey. Phobos: A Front-End Approach to Extensible Compilers. *36th Hawaii International Conference of System Sciences (HICSS)*, 2002.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[GJ90] D. Grune and C. J. Jacobs. *Parsing Techniques*. Ellis Horwood Limited, 1990.

[GL96] J. Gil and D. H. Lorenz. Environmental Acquisition: A New Inheritance-like Abstraction Mechanism. *11th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 214–231, 1996.

[GMS94] K. Ghosh, B. Mukherjee, and K. Schwan. A Survey of Real-Time Operating Systems. *A Survey of Real-Time Operating Systems. Technical Report Nr. GIT-CC-93/18, Georgia Institute of Technology*, 1994.

[Gor00] A. Gordon. *COM and COM+ Programming Primer*. Prentice Hall, 2000.

[GR89] A. Goldberg and D. Robson. *Smalltalk-80: The Language*. Addison-Wesley, 1989.

[Gra95] Paul Graham. *ANSI Common LISP*. Prentice Hall, 1995.

[Gra96] B. Gramlich. Termination and Confluence Properties of Structures Rewrite Systems. *PhD Thesis*, 1996.

[GSCK04] J. Greenfield, K. Short, S. Cook, and S. Kent. *Software Factories*. Addison-Wesley, 2004.

[Hal02] S. D. Halloway. *Component Development for the Java Platform*. Addison-Wesley, 2002.

[Har97] R. E. Harold. *JavaBeans*. IDG Books Worldwide, 1997.

[Har01] M. Harsu. A Survey of Product-Line Architectures. *Technical Report, Tampere University of Technology*, 2001.

[Har03] N. Harrison. Using the CodeDOM. *O'Reilly Network Article* `http://www.ondotnet.com/pub/a/dotnet/2003/02/03/codedom.html`, 2003.

[HB01] J. Hightower and G. Borriello. A Survey and Taxonomy of Location Systems for Ubiquitous Computing. *Extended paper from Computer, 34(8) p57-66, August*, 2001.

[Hed97] G. Hedin. Attribute Extension - A Technique for Enforcing Programming Conventions. *Nordic Journal of Computing*, 1997.

[Hel02] S. Helal. Pervasive Java. *IEEE - Pervasive Computing*, 2002.

[Hib04] JBoss Hibernate Project Homepage. `http://www.hibernate.org/`, 2004.

[HKC05] Y. Han, G. Kniesel, and A. Cremers. Towards Visual AspectJ by a Meta Model and Modeling Notation. *6th International Workshop on Aspect-Oriented Modeling*, 2005.

[HMU01] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2001.

[HS01] T. Howes and M. Smith. Model Driven Architecture (MDA) Technology Paper. `http://www.omg.org/docs/ormsc/01-07-01.pdf`, 2001.

[HSW+00] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System Architecture for Networked Sensors. *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 93–104, 2000.

[Hud98] P. Hudak. Modular Domain Specific Languages and Tools. *5th International Conference on Software Reuse (ICSR)*, pages 134–142, 1998.

[Hui02] B. Hui. Big Designs for Small Devices - Four J2ME Design Patterns Simplify Interactive Content Development. *JavaWorld*, 2002.

[Hum02] OMG Human-Usable Textual Notation (HUTN): Final Adopted Specification. `http://www.omg.org`, 2002.

[Hyp04] Hypersonic SQL Database Homepage. `http://hsqldb.sourceforge.net/`, 2004.

[IR97] Y. Ichisugi and Y. Roudier. Extensible Java Preprocessor Kit and Tiny Data-Parallel Java. *Scientific Computing in Object-Oriented Parallel Environments, (International Symposium on Computing in Object-Oriented Parallel Environments ISCOPE), LNCS 1343*, pages 153–160, 1997.

[J2E03] *Java 2 Platform Enterprise Edition Specification, Version 1.4*. Sun Microsystems, 2003.

[J2M02a] J2ME Devices Currently in the Market. Sun Technology Page. `http://developers.sun.com/techtopics/mobility/device/device`, 2002.

[J2M02b] J2ME MIDP Specification Version 2.0 JSR118. `http://jcp.org/aboutJava/communityprocess/final/jsr118/`, 2002.

[J2M04] J2ME Licensed Companies. `http://java.sun.com/j2me/reference/licensees/index.html`, 2004.

[Jav02a] Java 2 Enterprise Edition (J2EE) Specification. `http://java.sun.com/j2ee/`, 2002.

[Jav02b] JavaCC LR Parser Builder Tool. `http://www.webgain.com`, 2002.

[Jav04] Sun Java Technology Homepage. `http://java.sun.com`, 2004.

[Jav05] Java 2 Micro Edition (J2ME) Technology Homepage. `http://java.sun.com/j2me/`, 2005.

[Jay04] N. Jayaratchagan. Declarative Programming in Java. `http://www.OnJava.com./2004/04/21/declarative.html`, 2004.

[Jec04] M. Jeckle. UML Tools. `http://jeckle.de`, 2004.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, 1993.

[JH04] R. Johnson and J. Hoeller. *J2EE Development without EJB*. Wrox, 2004.

[JHF01] J.Ellis, L. Ho, and M. Fisher. JDBC 3.0 Specification. `http://java.sun.com/products/jdbc/`, 2001.

[JIM04] JIMI Software Development Kit. `http://java.sun.com/products/jimi/`, 2004.

[Joh03] R. Johnson. Introducing the Sprint Framework. `http://www.theserverside.com/articles/content/SpringFramework/article.html`, 2003.

[JRvdL00] M. Jazayeri, A. Ran, and F. van der Linden. *Software Architecture for Product Families: Principles and Practices*. Addison-Wesley, 2000.

[JSR03] JSR 175: A Metadata Facility for the Java Programming Language. `http://www.jcp.org/en/jsr/detail?id=175`, 2003.

[Kam98] S. Kamin. Research on Domain-Specific Embedded Languages and Program Generators. *Electronic Notes in Theoretical Computer Science, Elsevier Press, 12*, 1998.

[KCA04] G. Kniesel, P. Costanza, and M. Austermann. JMangler - A Powerful Back-End for Aspect-Oriented Programming. *In R. Filman, T. Elrad and S. Clarke, M. Aksit (Eds.): "Aspect-oriented Software Development", Prentice Hall*, 2004.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-Oriented Domain Analysis (FODA) Feasibility Study. *Technical Report, CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University*, 1990.

[KE02] C. Kuloor and A. Eberlein. Requirements Engineering for Software Product Lines. *15th International Conference on Software and Systems Engineering and their Applications (ICSSEA'02)*, 2002.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. *European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, LNCS 2072*, pages 327–353, 2001.

[KKP99] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next Century Challenges: Mobile Networking for "Smart Dust". *Conference on Mobile Computing and Networking (MOBICOM)*, pages 271–278, 1999.

[KLM+97] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-Oriented Programming. *European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, LNCS 1241*, pages 220–243, 1997.

[KM05] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[Knu90] D. Knuth. The Genesis of Attribute Grammars. *Conference on Attribute Grammars and their Applications*, 1990.

[Lad03] R. Laddad. *AspectJ in Action*. Manning Publications Co., 2003.

[Lad05] R. Laddad. AOP and Metadata: A Perfect Match. `http://www-128.ibm.com/developerworks/java/library/j-aopwork3/`, March 2005.

[Law02] G. Lawton. Moving Java into Mobile Phones. *IEEE Computer, Volume 35*, 2002.

[Lib01] J. Liberty. *Programming C#*. O'Reilly, 2001.

[LM99] D. Leijen and E. Meijer. Domain Specific Embedded Compilers. *2nd Conference on Domain-Specific Languages*, 1999.

[LMB92] J. Levine, T. Mason, and D. Brown. *Lex & Yacc*. Oreilly, 1992.

[LMB+01] A. Ledeczi, M. Maroti, A. Bakay, G. Karsai, J. Garrett, C. Thomason IV, G. Nordstrom, J. Sprinkle, and P. Volgyesi. The Generic Modeling Environment. *International Workshop on Intelligent Signal Processing*, 2001.

[LMB+03] M. Lindwer, D. Marculescu, T. Basten, R. Zimmermann, R. Marculescu, S. Jung, and E. Cantatore. Ambient Intelligence Visions and Achievements: Linking Abstract Ideas to Real-World Concepts. *Design Automation & TEst in Europe (DATE)*, 2003.

[Low01] J. Lowy. *COM and .NET Component Services*. O'Reilly, 2001.

[Low03] J. Lowy. Contexts in .NET: Decouple Components by Injecting Custom Services into Your Object's Interception Chain. *MSDN Magazine, March Issue*, 2003.

[LRZJ04] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting Product Lines Evolution with Framed Aspects. *3rd AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS)*, 2004.

[LS00] R. Lee and S. Seligman. *Java Naming and Directory Interface (JNDI) API Tutorial and Reference: Building Directory-Enabled Java Applications*. Addison-Wesley, 2000.

[MA99] A. Habel B. Hoffmann H. Kreowski S. Kuske D. Plump A. Schuerr G. Taentzery M. Andries, G. Engels. Graph Transformation for Specification and Programming. *Science of Computer Programming, Volume 34, 1*, 1999.

[Mar96] R. C. Martin. The Dependency Inversion Principle. *The C++ Report*, 1996.

[McL01] B. McLaughlin. *Java and XML*. O'Reilly, Second Edition edition, 2001.

[MDA03] OMG Model Driven Architecture (MDA) Guide Version 1.0. `http://www.omg.org/`, 2003.

[Men99] T. Mens. *A Formal Foundation for Object-Oriented Software Evolution*. PhD Dissertation / Vrije University Brussel, 1999.

[Met02] OMG Meta Object Facility (MOF) Specification. `http://www.omg.org`, 2002.

[Met03a] Automated Product Family Development: Nokia TETRA Terminal. *MetaCase White Paper* `http://metacase.com`, 2003.

[Met03b] MetaEdit+ Homepage. `http://metacase.com`, 2003.

[Met04] OMG Metamodel and UML Profile for Java and EJB Specification, Version 1.0. `http://www.omg.org`, 2004.

[Mez97a] M. Mezini. Maintaining the Consistency of Class Libraries During Their Evolution. *ACM 12th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Sigplan Notices, Volume 32, Number 10*, pages 1–22, 1997.

[Mez97b] M. Mezini. *Variation-Oriented Programming Beyond Classes and Inheritance*. PhD Thesis, University of Siegen, 1997.

[MF03] M. Majkut and B. Franczyk. Generation of Implementations for the Model Driven Architecture with Syntactic Unit Trees. *2nd Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Workshop Generative Techniques in the Context of MDA*, 2003.

[MH00] R. Monson-Haefel. *Enterprise JavaBeans*. Addison-Wesley, 2000.

[Mic05] Microsoft .NET Compact Framework Homepage. `http://msdn.microsoft.com/mobility/netcf/`, 2005.

[Mic06a] Microsoft Corporation. C# Version 3.0 Language Specification. `http://msdn.microsoft.com/vcsharp/future/`, 2006.

[Mic06b] Microsoft Research. The LINQ Project. `http://research.microsoft.com/Comega/`, 2006.

[Min98] N. H. Minsky. Why Should Architectural Principles be Enforced? *IEEE Computer Security, Dependability, and Assurance: From Needs to Solutions*, 1998.

[MK02] H. Masuhara and G. Kiczales. Compilation Semantics of Aspect-Oriented Programs. *2nd Conference on Aspect-Oriented Software Development (AOSD)*, 2002.

[MK03] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[MO03] M. Mezini and K. Ostermann. Modules for Crosscutting Models. *Invited Paper at 8th International Conference on Reliable Software Technologies, LNCS 2655*, 2003.

[MO04] M. Mezini and K. Ostermann. Variability Management with Feature-Oriented Programming and Aspects. *Foundations of Software Engineering (FSE-12), ACM SIGSOFT*, 2004.

[Mob02] Mobile Information Device Profile (MIDP). `http://java.sun.com/products/midp/`, 2002.

[Mob04] MobCon Homepage. `http://www.st.informatik.tu-darmstadt.de/static/pages/projects/mobcon/index.html`, 2004.

[MOF03] MOF 2.0 Query / Views / Transformations RFP. *OMG Document:* `http//:www.omg.org/docs/ad/02-03-03.rtf`, 2003.

[Moo05] D. Moore. Working with J2ME. `http://www.theserverside.com/articles/article.tss?l=WorkingwithJ2ME`, 2005.

[Muc97] S. Muchnick. *Advanced Compiler Construction*. Morgan Kaufmann Publishers, 1997.

[Nag96] M. Nagl, editor. *The IPSEN Project*. Lecture Notes in Computer Science. Springer-Verlag, 1996.

[New00] J. Newmarch. *A Programmer's Guide to Jini Technology*. APress, 2000.

[NFH+03] M. Nolin, J. Fredriksson, J. Hammarberg, J. G Huselius, J. Hakansson, A. Karlsson, O. Larses, M. Lindgren, G. Mustapic, A. Moeller, T. Nolte, J. Norberg, D. Nystroem, A. Tesanovic, and M. Akerholm. Component Based Software Engineering for Embedded Systems - A literature survey. *MRTC Report, Maelardalen University, ISSN 1404-3041*, 2003.

[NN99] F. Nielson and H. R. Nielson. *Semantics with Applications - A Formal Introduction*. Wiley, 1999.

[Nok04] Nokia Homepage. `http://nokia.com/`, 2004.

[NV02] J. Newkirk and A. Vorontsov. How .NET's Custom Attributes Affect Design. *IEEE SOFTWARE, Volume 19(5)*, pages 18–20, 2002.

[Ohl02] E. Ohlebusch. Hierarchical Termination Revisited. *Information Processing Letters*, 84(4):207–214, 2002.

[OMB05] K. Ostermann, M. Mezini, and C. Bockisch. Expressive Pointcuts for Increased Modularity. *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[OMG03] OMG Unified Modeling Language Specification. `http://www.org.org`, 2003.

[OSG02] OSGi Specification Release 2. `http://www.osgi.org/resources/`, 2002.

[Ost03] K. Ostermann. Modules for Hierarchical and Crosscutting Models. *PhD Thesis, Computer Science Department, Darmstadt University of Technology*, 2003.

[Paa95] J. Paakki. Attribute Grammars Paradigms - A High-Level Methodology in Language Implementation. *ACM Computing Surveys, Volume 27, Number 2*, pages 196–255, 1995.

[PAG03] A. Popovici, G. Alonso, and T. Gross. Spontaneous Container Services. *European Conference on Object-Oriented Programming (ECOOP)*, 2003.

[Pal04] Palm Homepage. `http://www.palm.com/`, 2004.

[Par72] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Par76] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering 2(1), March*, pages 1–9, 1976.

[Par02] T. Parr. ANTLR - ANother Tool for Language Recognition. `http://antlr.org`, 2002.

[PGA02] A. Popovici, T. Gross, and G. Alonso. Dynamic Weaving for Aspect Oriented Programming. *1st International Conference on Aspect-Oriented Software Development (AOSD)*, 2002.

[Pic03] PicoContainer Homepage. `http://www.picocontainer.org/`, 2003.

[Plu95] D. Plump. On Termination of Graph Rewriting. *Conference on Graph Theoretic Concepts in Computer Science*, 1995.

[Plu01] D. Plump. Essentials of Term Graph Rewriting. *Electronic Notes in Theoretical Computer Science, Volume 51*, 2001.

[PMD03] PMD Java Source Code Scanner. `http://pmd.sourceforge.net`, 2003.

[Poc04] Pocket PC Homepage. `http://www.microsoft.com/windowsmobile/pocketpc/ppc/default.mspx`, 2004.

[Pol00] M. Pollack. Code Generation Using Javadoc. *JavaWorld*, `http://www.javaworld.com/javaworld/jw-08-2000/jw-0818-javadoc.html`, 2000.

[POM03] R. Pichler, K. Ostermann, and M. Mezini. On Aspectualizing Component Models. *Software Practice and Experience, Volume 33, Issue 10, pp. 957-974*, 2003.

[PQVR+00] P. Pominville, F. Qian, R. Valle-Rai, L. Hendren, and C. Verbrugge. A Framework for Optimizing Java Using Attributes. *IBM Center for Advanced Studies (CAS) Conference*, 2000.

[Pro02] J. Prosise. *Programming Microsoft .NET*. Microsoft Press, 2002.

[PT02] R. Pohjonen and J. Tolvanen. Automated Production of Family Members: Lessons Learned. *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Workshop on Product Line Engineering*, 2002.

[Pus02] M. Pussinen. A Survey of Software Product-Line Evolution. *Technical Report, Tampere University of Technology*, 2002.

[QAL05] QUALCOMM BREW Homepage. `http://brew.qualcomm.com/`, 2005.

[QDo03] QDox Java Tag Parser. `http://qdox.codehaus.org/`, 2003.

[Rad03] Radiology Manager Medical Application. `http://www.mysterian.com`, 2003.

[Ram02] I. Rammer. *Advanced .NET Remoting (C# Edition)*. APress, 2002.

[RH04] P. Van Roy and S. Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.

[Rie96] A. Riel. *Object-Oriented Design Heuristics*. Addison Wesley, 1996.

[RS97] J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.

[RVDA01] Editors G. Riva, F. Vatalaro, F. Davide, and M. Alcaniz. *Ambient Intelligence*. IOS Press `http://www.emergingcommunication.com/volume6.html`, 2001.

[SAW94] B. Schilit, N. Adams, and R. Want. Context-Aware Computing Applications. In *IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[SB00] M. Svahnberg and J. Bosch. Issues Concerning Variability in Software Product Lines. *Software Architectures for Product Families, IW-SAPF-3, LNCS, Volume 1951*, pages 146–157, 2000.

[SB02] Y. Smaragdakis and D. Batory. Mixin layers: an Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255, 2002.

[Sch94a] A. Schreiner. *Object-Oriented Programming with ANSI-C*. Hanser, Munich, 1994.

[Sch94b] A. Schürr. Rapid Programming with Graph Rewrite Rules. *USENIX Symposium Very High Level Languages (VHLL), Berkeley: USENIX Association*, 1994.

[SDN04] N. Schärli, S. Ducasse, and O. Nierstrasz. Composable Encapsulation Policies. *18th European Conference on Object-Oriented Programming (ECOOP)*, 2004.

[SG01] A. Skonnard and M. Gudgin. *Essential XML: Quick Reference*. Addison-Wesley, 2001.

[SGM02] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, 2002.

[SK97] J. Sztipanovits and G. Karsai. Model-Integrated Computing. *IEEE Computer*, 1997.

[Spr04] Spring Application Framework Homepage. *`http://www.springframework.org/`*, 2004.

[SQL03] SQLj Homepage. *`http://www.sqlj.org/`*, 2003.

[SSRB00] D. Schmidt, M. Stal, H. Rohnert, and F. Buchmann. *Pattern Oriented Software Architecture. Volume 2*. Wiley, 2000.

[Ste99] J. R. Steel. Generating Human-Usable Textual Notations For Information Models. *Diploma Thesis - Department of Computer Science and Electrical Engineering, University of Queensland, Brisbane, Australia*, 1999.

[Str97] B. Stroustrup. *The C++ Programming Language, Third Edition*. Addison-Wesley, 1997.

[SWZ95] A. Schürr, A. Winter, and A. Zürndorf. Graph Grammar Engineering with PROGRESS. *European Software Engineering Conference ESEC 5, LNCS 989*, pages 219–234, 1995.

[SY01] M. Shomrat and A. Yehudai. Obvious or Not? Regulating Architectural Decisions Using Aspect-Oriented Programming. *Aspect-Oriented Software Development (AOSD)*, 2001.

[TBG03] T. Tourwe, J. Brichau, and K. Gybels. On the Existence of the AOSD-Evolution Paradox. *Aspect-Oriented Software Development Conference - Workshop on Software-Engineering Properties of Languages for Aspect Technologies*, 2003.

[TCKI00] M. Tatsubori, S. Chiba, M. Killijian, and K. Itano. OpenJava: A Class-based Macro System for Java. *Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Reflection and Software Engineering Workshop*, 2000.

[The02] The Self Programming Language. *`http://research.sun.com/self/language.html`*, 2002.

[Tho98] E. O. Thorp. The Invention of the First Wearable Computer. *2nd Symposium on Wearable Computers*, pages 4–8, 1998.

[TS97] W. Taha and T. Sheard. Multi-stage Programming. *ACM SIGPLAN Notices*, 32(8), 1997.

[TS01] D. Tidwelland and S. S.Laurent. *XSLT*. O'Relly, 2001.

[UML03] UML Profile for EJB, JSR26. `http://jpc.org/jsr/detail/26.jsp`, 2003.

[vdBHKO02] M. G. J. van der Brand, J. Heering, P. Klint, and P.A. Oliver. Compiling Language Definitions: The ASF+SDF Compiler. *ACM Transactions on Programming Languages and Systems, Volume 24, Number 4*, pages 334–368, 2002.

[vDK01] A. van Deursen and P. Klint. Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 2001.

[vDKV00] A. van Deursen, P. Klint, and J. Visser. Domain-Specific Languages. *ACM SIGPLAN Notices*, 35:26–36, 6 2000.

[vDo03] vDoclet Java Code-Generation Framework. `http://vdoclet.sourceforge.net/`, 2003.

[Vel03] Velocity Template Scritp Engine for Java. `http://jakarta.apache.org/velocity/index.html`, 2003.

[Vel04] Velocity Powered Applications. `http://jakarta.apache.org/velocity/powered.html`, 2004.

[Vis01a] E. Visser. A Survey of Rewriting Strategies in Program Transformation Systems. *Workshop on Reduction Strategies in Rewriting and Programming (WRSRP)*, 2001.

[Vis01b] E. Visser. Stratego: A Language for Program Transformations Based on Rewriting Strategies. *Rewriting Techniques and Applications (RTA), LNCS 2051*, pages 357–361, 2001.

[VL02] P. Voelgyesi and A. Ledezi. Component-Based Development of Networked Embedded Applications. *28th Euromicro Conference, Component-Based Software Engineering Track*, 2002.

[Voe03a] M. Voelter. A collection of Patterns for Program Generation. *European Conference on Pattern Languages of Programs (EuroPLoP)*, 2003.

[Voe03b] M. Voelter. A Generative Component Infrastructure for Embedded Systems. `http://www.voelter.de/data/pub/SmallComponents.pdf`, 2003.

[VSW02] M. Voelter, A. Schmid, and E. Wolf. *Server Components Patterns, Illustrated with EJB.* Wiley & Sons, 2002.

[vWV03] J. van Wijngaarden and E. Visser. Program Transformation Mechanics. A Classification of Mechanisms for Program Transformation with a Survey of Existing Transformation Systems. *Technical Report UU-CS-2003-048. Institute of Information and Computing Sciences, Utrecht University*, 2003.

[WAP03] WAP Forum Homepage. `http://www.wapforum.org/`, 2003.

[War94] M. Ward. Language Oriented Programming. *Software - Concept and Tools*, pages 147–161, 1994.

[WB00] D. A. Watt and D. F. Brown. *Programming Language Processors in Java*. Prentice Hall, 2000.

[Wea04] Weave.NET. `http://www.dsg.cs.tcd.ie/sites/Weave.NET.html`, 2004.

[Web03] WebMacro Template Script Language. `http://www.webmacro.org/`, 2003.

[Wei91] M. Weiser. The Computer for the Twenty-First Century. *Scientific American*, 1991.

[Wei93] M. Weiser. Some Computer Science Problems in Ubiquitous Computing. *Communications of the ACM*, 1993.

[Wei94] M. Weiser. The World is not a Desktop. *ACM Interactions*, 1994.

[Wir77] N. Wirth. Toward a Discipline of Real-Time Programming. *Communications of ACM*, 20(8):577–583, 1977.

[WLLP01] B. Warneke, M. Last, B. L., and K. S. J. Pister. Smart Dust: Communicating with a Cubic-Millimeter Computer. *Computer*, 34(1):44–51, 2001.

[WMBK02] E. Van Wyk, O. De Morr, K. Backhouse, and P. Kwiatkowski. Forwarding in Attribute Grammars for Modular Language Design. *Compiler Construction*, pages 128–142, 2002.

[xDo03] xDoxclet Code Generation Engine. `http://xdoclet.sourceforge.net/`, 2003.

[XML99] XML Path Language (XPath). `http://www.w3.org/TR/xpath`, 1999.

[XQu05] XQuery - An XML Query Language Homepage. `http://www.w3.org/TR/xquery/`, 2005.

[YCS+02] Z. Yang, B. H. C. Cheng, R. E. K. Stirewalt, J. Sowell, S. M. Sadjadi, and P. K. McKinley. An Aspect-Oriented Approach to Dynamic Adaptation. *Workshop on Self-healing Systems, AMC SIGSOFT*, pages 85–92, 2002.

[YD04] P. Yao and D. Durant. *.NET Compact Framework Programming with C#.* Addison-Wesley, 2004.

[Zde02] I. Zderadicka. CS CODEDOM Parser. `http://ivanz.webpark.cz/csparser.html`, 2002.

[Zen04] M. Zenger. Keris: Evolving Software with Extensible Modules. *Journal of Software Maintenance and Evolution: Research and Practice (Special Issue on USE), Wiley*, 2004.

[ZJF01] T. Ziadi, J.-M. Jezequel, and F. Fondement. Product Line Engineering with UML. *Workshop on Model Driven Architecture and Product Line Engineering, (SPLC'02)*, 2001.

[ZO01] M. Zenger and M. Odersky. Implementing Extensible Compilers. *Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2001.