1

The MDA Development Process

This chapter describes some of the major problems in software development. It explains the concepts of the Model Driven Architecture (MDA), and discusses how MDA can help to solve these problems.

1.1 Traditional Software Development

Software development is often compared with hardware development in terms of maturity. While in hardware development there has been much progress, e.g., processor speed has grown exponentially in twenty years, the progress made in software development seems to be minimal. To some extent this is a matter of appearances. The progress made in software development cannot be measured in terms of development speed or costs.

Progress in software development is evident from the fact that it is feasible to build much more complex and larger systems. Just think how quickly and efficiently we would be able to build a monolithic mainframe application that has no graphical user interface and no connections to other systems. We never do this anymore, and that is why we do not have solid figures to support the idea that progress has been made.

Still, software development is an area in which we are struggling with a number of major problems. Writing software is labor intensive. With each new technology, much work needs to be done again and again. Systems are never built using only one technology and systems always need to communicate with other systems. There is also the problem of continuously changing requirements.

To show how MDA addresses these problems, we will analyze some of the most important problems with software development and discover the cause of these problems.

1.1.1 The Productivity Problem

The software development process as we know it today is often driven by low-level design and coding. A typical process, as illustrated in Figure 1-1, includes a number of phases:

1. Conceptualization and requirements gathering
2. Analysis and functional description
3. Design
4. Coding
5. Testing
6. Deployment

Whether we use an incremental and iterative version of this process, or the traditional waterfall process, documents and diagrams are produced during phases 1 through 3. These include requirements descriptions in text and pictures, and often many Unified Modeling Language (UML) diagrams like use cases, class diagrams, interaction diagrams, activity diagrams, and so on. The stack of paper produced is sometimes impressive. However, most of the artifacts from these phases is *just paper* and nothing more.

The documents and corresponding diagrams created in the first three phases rapidly lose their value as soon as the coding starts. The connection between the diagrams and the code fades away as the coding phase progresses. Instead of being an exact specification of the code, the diagrams usually become more or less unrelated pictures.

When a system is changed over time, the distance between the code and the text and diagrams produced in the first three phases becomes larger. Changes are often done at the code level only, because the time to update the diagrams and other high-level documents is not available. Also, the added value of updated diagrams and documents is questionable, because any new change starts in the code anyway. So why do we use so much precious time building high-level specifications?

The idea of Extreme Programming (XP) (Beck 2000) has become popular in a rapid fashion. One of the reasons for this is that it acknowledges the fact that the code is the driving force of software development. The only phases in the development process that are really productive are coding and testing.

As Alistair Cockburn states in *Agile Software Development* (Cockburn 2002), the XP approach solves only part of the problem. As long as the same team works on the software, there is enough high-level knowledge in their heads to understand the system. During initial development this is often the case. The problems start when the team is dismantled, which usually happens after delivery of the first release of the software. Other people need to maintain (fix bugs, enhance functionality, and so on) the

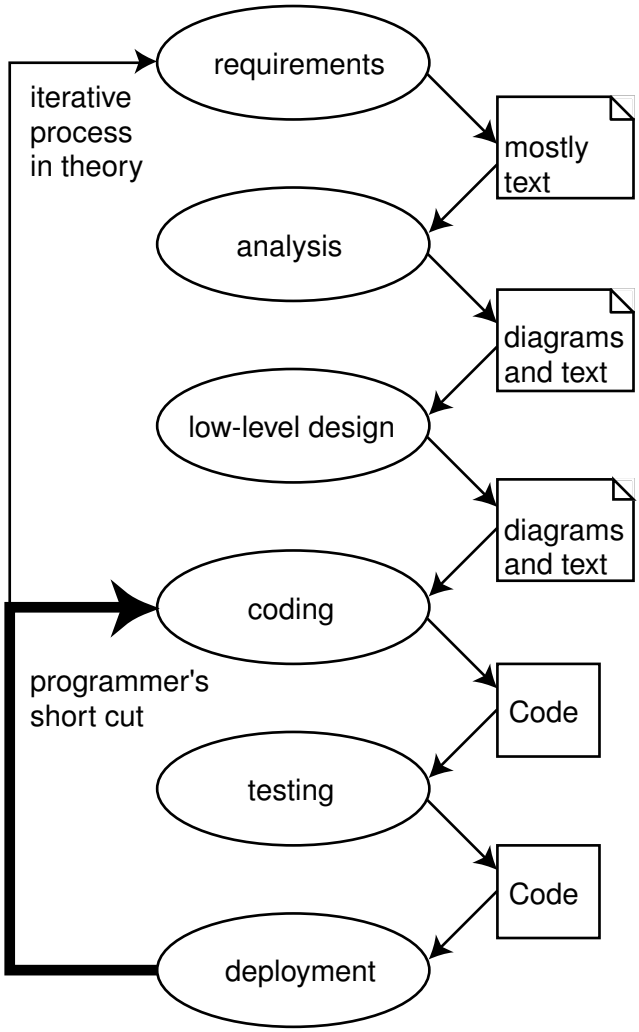

Figure 1-1 *Traditional software development life cycle*

software. Having just code and tests makes maintenance of a software system very difficult. Given five-hundred thousand lines of code (or even much more), where do you start to try and understand how a system works?

Alistair Cockburn talks about "markers" that need to be left behind for new people who will work on the software. These markers often take the form of text and higher-level diagrams. Without them you are literally lost. It is like being dropped in a foreign city without a map or road signs that show the directions.

So, either we use our time in the first phases of software development building high-level documentation and diagrams, or we use our time in the maintenance phase finding out what the software is actually doing. Neither way we are directly productive in the sense that we are producing code. Developers often consider these tasks as being overhead. Writing code is being productive, writing models or documentation is not. Still, in a mature software project these tasks need to be done.

1.1.2 The Portability Problem

The software industry has a special characteristic that makes it stand apart from most other industries. Each year, and sometimes even faster, new technologies are being invented and becoming popular (for example Java, Linux, XML, HTML, SOAP, UML, J2EE, .NET, JSP, ASP, Flash, Web Services, and so on). Many companies need to follow these new technologies for good reasons:

- The technology is demanded by the customers (e.g., Web interfaces).
- It solves some real problems (e.g., XML for interchange or Java for portability).
- Tool vendors stop supporting old technologies and focus on the new one (e.g., UML support replaces OMT).

The new technologies offer tangible benefits for companies and many of them cannot afford to lag behind. Therefore, people have to jump on these new technologies quite fast. As a consequence, the investments in previous technologies lose value and they may even become worthless.

The situation is even more complex because the technologies themselves change as well. They come in different versions, without a guarantee that they are backwards compatible. Tool vendors usually only support the two or three most recent versions.

As a consequence, existing software is either ported to the new technology, or to a newer version of an existing technology. The software may remain unchanged utilizing the older technology, in which case the existing, now legacy, software needs to interoperate with new systems that will be built using new technology.

1.1.3 The Interoperability Problem

Software systems rarely live in isolation. Most systems need to communicate with other, often already existing, systems. As a typical example we have seen that over the past years many companies have been building new Web-based systems. The new end-user application runs in a Web browser (using various technologies like HTML, ASP, JSP, and so on) and it needs to get its information from existing back-end systems.

Even when systems are completely built from scratch, they often span multiple technologies, sometimes both old and new. For example, when a system uses Enter-

prise Java Beans (EJB), it also needs to use relational databases as a storage mechanism.

Over the years we have learned not to build huge monolithic systems. Instead we try to build components that do the same job by interacting with each other. This makes it easier (or at all possible) to make changes to a system. The different components are all built using the best technology for the job, but need to interact with each other. This has created a need for interoperability.

1.1.4 The Maintenance and Documentation Problem

In the previous sections, we touched upon the problem of maintenance. Documentation has always been a weak link in the software development process. It is often done as an afterthought. Most developers feel their main task is to produce code. Writing documentation during development costs time and slows down the process. It does not support the developer's main task. The availability of documentation supports the task of those that come later. So, writing documentation feels like doing something for the sake of prosperity, not for your own sake. There is no incentive to writing documentation other than your manager, who tells you that you must.

This is one of the main reasons why documentation is often not of very good quality. The only persons that can check the quality are fellow developers who hate the job of writing documentation just as much. This also is the reason that documentation is often not kept up to date. With every change in the code the documentation needs to be changed as well—by hand!

The developers are wrong, of course. Their task is to develop systems that can be changed and maintained afterwards. Despite the feelings of many developers, writing documentation is one of their essential tasks.

A solution to this problem at the code level is the facility to generate the documentation directly from the source code, ensuring that it is always up to date. The documentation is in effect part of the code and not a separate entity. This is supported in several programming languages, like Eiffel and Java. This solution, however, only solves the low-level documentation problem. The higher-level documentation (text and diagrams) still needs to be maintained by hand. Given the complexity of the systems that are built, documentation at a higher level of abstraction is an absolute must.

1.2 THE MODEL DRIVEN ARCHITECTURE

The Model Driven Architecture (MDA) is a framework for software development defined by the Object Management Group (OMG). Key to MDA is the importance of

models in the software development process. Within MDA the software development process is driven by the activity of modeling your software system.

In this section we first explain the basic MDA development life cycle, and next illustrate how MDA can help to solve (at least part of) the problems mentioned in the previous sections.

1.2.1 The MDA Development Life Cycle

The MDA development life cycle, which is shown in Figure 1-2, does not look very different from the traditional life cycle. The same phases are identified. One of the major differences lies in the nature of the artifacts that are created during the development process. The artifacts are formal models, i.e., models that can be understood by computers. The following three models are at the core of the MDA.

Platform Independent Model

The first model that MDA defines is a model with a high level of abstraction that is independent of any implementation technology. This is called a Platform Independent Model (PIM).

A PIM describes a software system that supports some business. Within a PIM, the system is modeled from the viewpoint of how it best supports the business. Whether a system will be implemented on a mainframe with a relational database or on an EJB application server plays no role in a PIM.

Platform Specific Model

In the next step, the PIM is transformed into one or more Platform Specific Models (PSMs). A PSM is tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology. For example, an EJB PSM is a model of the system in terms of EJB structures. It typically contains EJB-specific terms like “home interface,” “entity bean,” “session bean,” and so on. A relational database PSM includes terms like “table,” “column,” “foreign key,” and so on. It is clear that a PSM will only make sense to a developer who has knowledge about the specific platform.

A PIM is transformed into one or more PSMs. For each specific technology platform a separate PSM is generated. Most of the systems today span several technologies, therefore it is common to have many PSMs with one PIM.

Code

The final step in the development is the transformation of each PSM to code. Because a PSM fits its technology rather closely, this transformation is relatively straightforward.

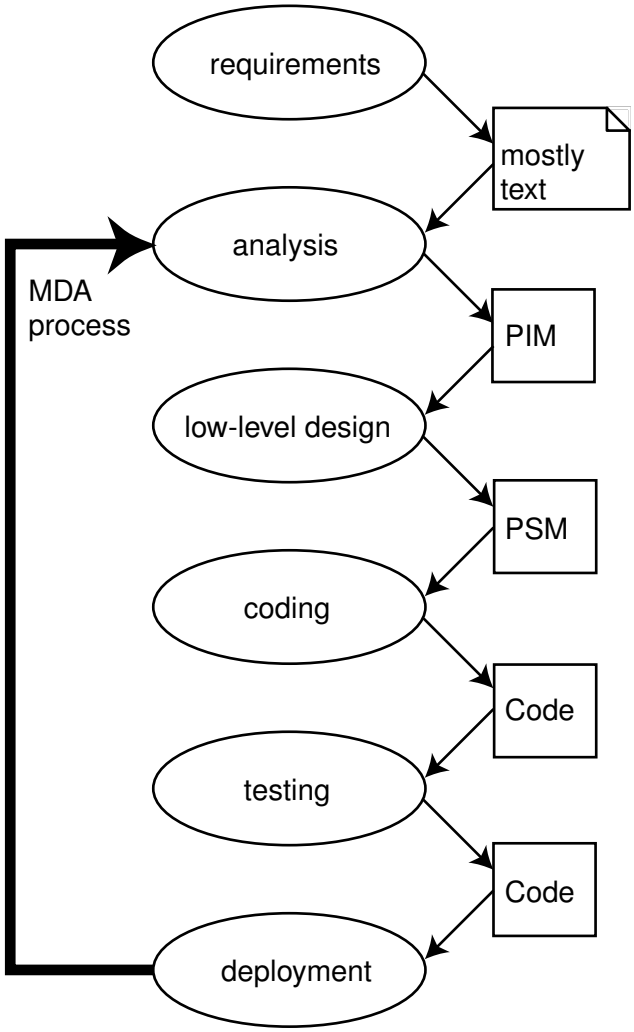

Figure 1-2 *MDA software development life cycle*

The MDA defines the PIM, PSM, and code, and also defines how these relate to each other. A PIM should be created, then transformed into one or more PSMs, which then are transformed into code. The most complex step in the MDA development process is the one in which a PIM is transformed into one or more PSMs.

Raising the Level of Abstraction

The PIM, PSM, and code are shown as artifacts of different steps in the development life cycle. More importantly, they represent different abstraction levels in the system specification. The ability to transform a high level PIM into a PSM raises the level of abstraction at which a developer can work. This allows a developer to cope with more complex systems with less effort.

1.2.2 Automation of the Transformation Steps

The MDA process may look suspiciously much like traditional development. However, there is a crucial difference. Traditionally, the transformations from model to model, or from model to code, are done mainly by hand. Many tools can generate some code from a model, but that usually goes no further than the generation of some template code, where most of the work still has to be filled in by hand.

In contrast, MDA transformations are always executed by tools as shown in Figure 1-3. Many tools are able to transform a PSM into code; there is nothing new to that. Given the fact that the PSM is already very close to the code, this transformation isn't that exciting. What's new in MDA is that the transformation from PIM to PSM is automated as well. This is where the obvious benefits of MDA come in. How much effort has been spent in your projects with the painstaking task of building a database model from a high-level design? How much (precious) time was used by building a COM component model, or an EJB component model from that same design? It is indeed about time that the burden of IT-workers is eased by automating this part of their job.

At the time of writing, the MDA approach is very new. As a result of this, current tools are not sophisticated enough to provide the transformations from PIM to PSM and from PSM to code for one hundred percent. The developer needs to manually enhance the transformed PSM and/or code models. However, current tools are able to generate a running application from a PIM that provides basic functionality, like creating and changing objects in the system. This does allow a developer to have immediate feedback on the PIM that is under development, because a basic prototype of the resulting system can be generated on the fly.

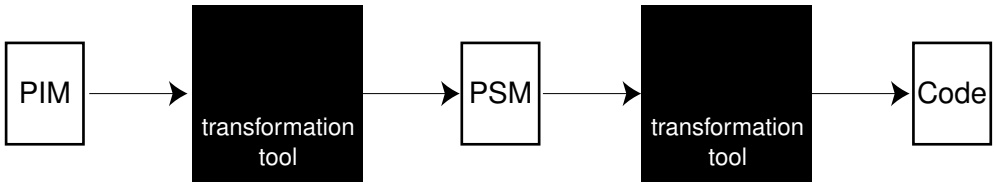

Figure 1-3 *The three major steps in the MDA development process*

1.3 MDA BENEFITS

Let us now take a closer look at what application of MDA brings us in terms of improvement of the software development process.

1.3.1 Productivity

In MDA the focus for a developer shifts to the development of a PIM. The PSMs that are needed are generated by a transformation from PIM to PSM. Of course, someone still needs to define the exact transformation, which is a difficult and specialized task. But such a transformation only needs to be defined once and can then be applied in the development of many systems. The payback for the effort to define a transformation is large, but it can only be done by highly skilled people.

The majority of developers will focus on the development of PIMs. Since they can work independently of details and specifics of the target platforms, there is a lot of technical detail that they do not need to bother with. These technical details will be automatically added by the PIM to PSM transformation. This improves the productivity in two ways.

In the first place, the PIM developers have less work to do because platform-specific details need not be designed and written down; they are already addressed in the transformation definition. At the PSM and code level, there is much less code to be written, because a large amount of the code is already generated from the PIM.

The second improvement comes from the fact that the developers can shift focus from code to PIM, thus paying more attention to solving the business problem at hand. This results in a system that fits much better with the needs of the end users. The end users get better functionality in less time.

Such a productivity gain can only be reached by the use of tools that fully automate the generation of a PSM from a PIM. Note that this implies that much of the information about the application must be incorporated in the PIM and/or the generation tool. Because the high-level model is no longer "just paper," but directly related to the generated code, the demands on the completeness and consistency of the high-level model (PIM) are higher than in traditional development. A human reading a paper model may be forgiving—an automated transformation tool is not.

1.3.2 Portability

Within the MDA, portability is achieved by focusing on the development of PIMs, which are by definition platform independent. The same PIM can be automatically transformed into multiple PSMs for different platforms. Everything you specify at the PIM level is therefore completely portable.

The extent to which portability can be achieved depends on the automated transformation tools that are available. For popular platforms, a large number of tools will undoubtedly be (or become) available. For less popular platforms, you may have to use a tool that supports plug-in transformation definitions, and write the transformation definition yourself.

For new technologies and platforms that will arrive in the future, the software industry needs to deliver the corresponding transformations in time. This enables us to quickly deploy new systems with the new technology, based on our old and existing PIMs.

1.3.3 Interoperability

We have been incomplete regarding the overall MDA picture. As shown in Figure 1-4, multiple PSMs generated from one PIM may have relationships. In MDA these are called *bridges*. When PSMs are targeted at different platforms, they cannot directly talk with each other. One way or another, we need to transform concepts from one platform into concepts used in another platform. This is what interoperability is all about. MDA addresses this problem by generating not only the PSMs, but the necessary bridges between them as well.

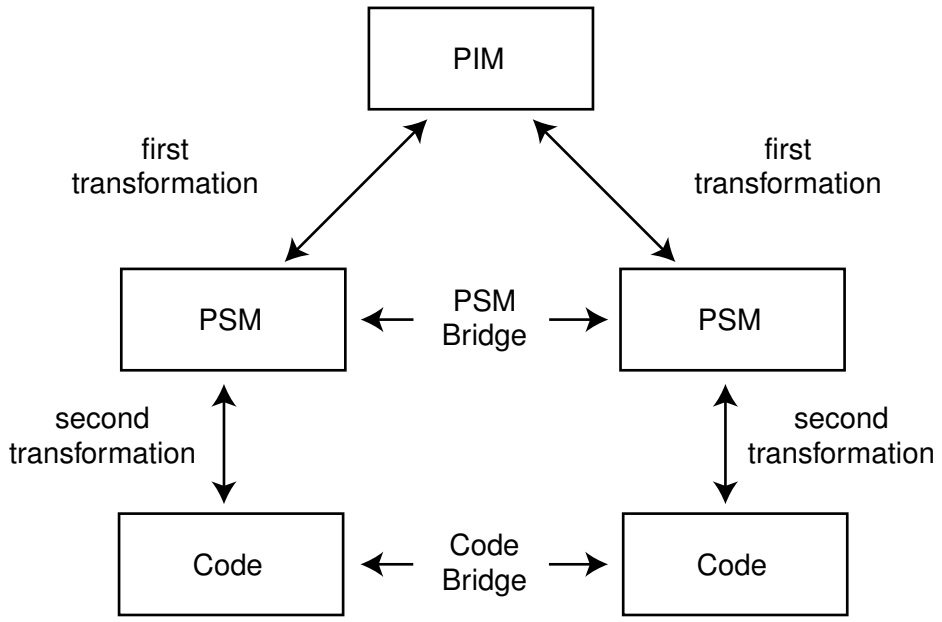

Figure 1-4 *MDA interoperability using bridges*

If we are able to transform one PIM into two PSMs targeted at two platforms, all of the information we need to bridge the gap between the two PSMs is available. For each element in one PSM we know from which element in the PIM it has been transformed. From the PIM element we know what the corresponding element is in the second PSM. We can therefore deduce how elements from one PSM relate to elements in the second PSM. Since we also know all the platform-specific technical details of both PSMs (otherwise we couldn't have performed the PIM-to-PSM transformations), we have all the information we need to generate a bridge between the two PSMs.

Take, for example, one PSM to be a Java (code) model and the other PSM to be a relational database model. For an element *Customer* in the PIM, we know to which Java class(es) this is transformed. We also know to which table(s) this *Customer* element is transformed. Building a bridge between a Java object in the Java-PSM and a table in the Relational-PSM is easy. To retrieve an object from the database, we query the table(s) transformed from *Customer,* and instantiate the class(es) in the other PSM with the data. To store an object, we find the data in the Java object and store it in the "Customer" tables.

Cross-platform interoperability can be realized by tools that not only generate PSMs, but the bridges between them, and possibly to other platforms, as well. You can "survive" technology changes while preserving your investment in the PIM.

1.3.4 Maintenance and Documentation

Working with the MDA life cycle, developers can focus on the PIM, which is at a higher level of abstraction than code. The PIM is used to generate the PSM, which in turn is used to generate the code. The model is an exact representation of the code. Thus, the PIM fulfills the function of high-level documentation that is needed for any software system.

The big difference is that the PIM is not abandoned after writing. Changes made to the system will eventually be made by changing the PIM and regenerating the PSM and the code. In practice today, many of the changes are made to the PSM and code is regenerated from there. Good tools, however, will be able to maintain the relationship between PIM and PSM, even when changes to the PSM are made. Changes in the PSM will thus be reflected in the PIM, and high-level documentation will remain consistent with the actual code.

In the MDA approach the documentation at a high level of abstraction will naturally be available. Even at that level, the need to write down additional information, which cannot be captured in a PIM, will remain. This includes, for example, argumentation for choices that have been made while developing the PIM.

1.4 MDA BUILDING BLOCKS

Now what do we need to implement the MDA process? The following are the building blocks of the MDA framework:

- High-level models, written in a standard, well-defined language, that are consistent, precise, and contain enough information on the system.
- One or more standard, well-defined languages to write high-level models.
- Definitions of how a PIM is transformed to a specific PSM that can be automatically executed. Some of these definitions will be "home-made," that is, made by the project that works according to the MDA process itself. Preferably, transformation definitions would be in the public domain, perhaps even standardized, and tunable to the individual needs of its users.
- A language in which to write these definitions. This language must be interpreted by the transformation tools, therefore it must be a formal language.
- Tools that implement the execution of the transformation definitions. Preferably these tools offer the users the flexibility to tune the transformation step to their specific needs.
- Tools that implement the execution of the transformation of a PSM to code.

At the time of writing, many of the above building blocks are still under development. Chapter 3 provides an overview of where we stand today.

In the following chapters each of the building blocks is further examined and we show how it fits into the overall MDA framework.

1.5 SUMMARY

The Model Driven Architecture is a framework for software development, defined by the OMG. Key to MDA is the importance of models in the software development process. Within MDA the software development process is driven by the activity of modeling your software system.

The MDA development life cycle is not very different from the traditional life cycle. The artifacts of the MDA are formal models, i.e., models that can be understood by computers. The following three models are at the core of the MDA:

- Platform Independent Model (PIM), a model with a high level of abstraction, that is independent of any implementation technology.

- Platform Specific Model (PSM), a model tailored to specify your system in terms of the implementation constructs that are available in one specific implementation technology. A PIM is transformed into one or more PSMs.
- Code, a description (specification) of the system in source code. Each PSM is transformed into code.

Traditionally the transformations from model to model, or from model to code, are done mainly by hand. In contrast, MDA transformations are always executed by tools. Many tools have been able to transform a PSM to code; there is nothing new to that. What's new in MDA is that the transformation from PIM to PSM is automated as well.