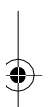
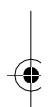


Chapter 1

Introduction



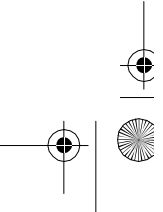
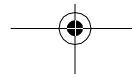
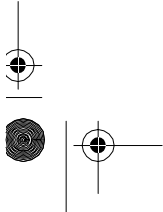
There's no doubt about it: Software is expensive. The United States alone devotes at least \$250 billion each year to application development of approximately 175,000 projects involving several million people. For all of this investment of time and money, though, software's customers continue to be disappointed, because over 30 percent of the projects will be canceled before they're completed, and more than half of the projects will cost nearly twice their original estimates.¹



The demand for software also continues to rise. The developed economies rely to a large extent on software for telecommunications, inventory control, payroll, word processing and typesetting, and an ever-widening set of applications. Only a decade ago, the Internet was text-based, known only to a relatively few scientists connected using DARPAnet and email. Nowadays, it seems as if everyone has his or her own website. Certainly, it's become difficult to conduct even non-computer-related business without email.

There's no end in sight. A Star Trek world of tiny communications devices, voice-recognition software, vast searchable databases of human (for the moment, anyway) knowledge, sophisticated computer-controlled sensing devices, and intelligent display are now imaginable. (As software

1. Consult sources such as Ovum (<http://www.ovum.com>) and the Standish Group (<http://www.standishgroup.com>) for more definitive numbers. Each analyst uses different criteria to establish his or her numbers.



professionals, however, we know just how much ingenuity will be required to deliver these new technologies.)

Software practitioners, industrial experts, and academics have not been idle in the face of this need to improve productivity. There have been significant improvements in the ways in which we build software over the last fifty years, two of which are worthy of note in our attempts to make software an asset. First, we've raised the level of abstraction of the languages we use to express behavior; second, we've sought to increase the level of reuse in system construction.

These techniques have undoubtedly improved productivity, but as we bring more powerful tools to bear to solve more difficult problems, the size of each problem we're expected to tackle increases to the point at which we could, once again, barely solve it.

MDA takes the ideas of raising the levels of abstraction and reuse up a notch. It also introduces a new idea that ties these ideas together into a greater whole: design-time interoperability.

Raising the Level of Abstraction²

The history of software development is a history of raising the level of abstraction. Our industry used to build systems by soldering wires together to form hard-wired programs. Machine code let us store programs by manipulating switches to enter each instruction. Data was stored on drums whose rotation time had to be taken into account so that the head would be able to read the next instruction at exactly the right time. Later, assemblers took on the tedious task of generating sequences of ones and zeroes from a set of mnemonics designed for each hardware platform.

At some point, programming languages, such as FORTRAN, were born and "formula translation" became a reality. Standards for COBOL and C enabled portability among hardware platforms, and the profession developed

2. This section is drawn from *Executable UML: A Foundation for Model-Driven Architecture* by Stephen J. Mellor and Marc J. Balcer (Addison-Wesley, 2002), with permission of the authors. The arguments made there for executable UML rely on raising the level of abstraction as a foundation for model-driven architecture. The same arguments apply here. Reuse in action!

techniques for structuring programs so that they were easier to write, understand, and maintain. We now have languages like Smalltalk, C++, Eiffel, and Java, each with the notion of object-orientation, an approach for structuring data and behavior together into classes and objects.

As we moved from one language to another, generally we increased the level of abstraction at which the developer operates, which required the developer to learn a new, higher-level language that could then be mapped into lower-level ones, from C++ to C to assembly code to machine code and the hardware. At first, each higher layer of abstraction was introduced only as a concept. The first assembly languages were no doubt invented without the benefit of an (automated) assembler to turn mnemonics into bits, and developers were grouping functions together with the data they encapsulated long before there was any automatic enforcement of the concept. Similarly, the concepts of structured programming were taught before there were structured programming languages in widespread industrial use (for instance, Pascal).

Over time, however, the new layers of abstraction became formalized, and tools such as assemblers, preprocessors, and compilers were constructed to support the concepts. This had the effect of hiding the details of the lower layers so that only a few experts (compiler writers, for example) needed to concern themselves with the details of how those layers work. In turn, this raises concerns about the loss of control induced by, for example, eliminating the GOTO statement or writing in a high-level language at a distance from the “real machine.” Indeed, sometimes the next level of abstraction has been too big a reach for the profession as a whole, only of interest to academics and purists, and the concepts did not take a large enough mindshare to survive. (ALGOL-68 springs to mind. So does Eiffel, but it has too many living supporters to be a safe choice of example.)

As the profession has raised the level of abstraction at which developers work, we have developed tools to map from one layer to the next automatically. Developers now write in a high-level language that can be mapped to a lower-level language automatically, instead of writing in the lower-level language that can be mapped to assembly language, just as our predecessors wrote in assembly language and had that translated automatically into machine language.

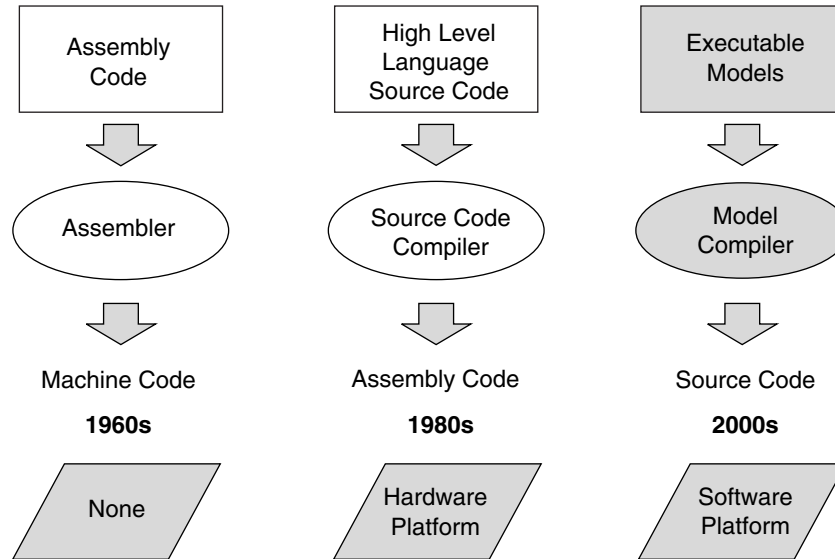


Figure 1-1 Raising the level of abstraction

Clearly, this forms a pattern: We formalize our knowledge of an application in as high a level a language as we can. Over time, we learn how to use this language and apply a set of conventions for its use. These conventions become formalized and a higher-level language is born that is mapped automatically into the lower-level language. In turn, this next-higher-level language is perceived as low level, and we develop a set of conventions for its use. These newer conventions are then formalized and mapped into the next level down, and so forth.

The next level of abstraction is the move, shown in Figure 1-1, to model-based development, in which we build software-platform-independent models.

Software-platform independence is analogous to hardware-platform independence. A hardware-platform-independent language, such as C or Java, enables the writing of a specification that can execute on a variety of hardware platforms with no change. Similarly, a software-platform-independent language enables the writing of a specification that can execute on a variety of software platforms, or software architecture designs, with no change. So, a software-platform-independent specification could be mapped to a multiprocessor/multitasking CORBA environment, or a client-server relational database environment, with no change to the model.

In general, the organization of the data and processing implied by a conceptual model may not be the same as the organization of the data and processing in implementation. If we consider two concepts, those of “customer” and “account,” modeling them as classes using the UML suggests that the software solution should be expressed in terms of software classes named Customer and Account. However, there are many possible software designs that can meet these requirements, many of which are not even object-oriented. Between concept and implementation, an attribute may become a reference; a class may be divided into sets of object instances according to some sorting criteria; classes may be merged or split; statecharts may be flattened, merged, or separated; and so on. A modeling language that enables such mappings is software-platform independent.

Raising the level of abstraction changes the platform on which each layer of abstractions depends. Model-based development relies on the construction of models that are independent of their software platforms, which include the likes of CORBA, client-server relational database environments, and the very structure of the final code.

Raising the Level of Reuse

Though some pundits have suggested that there has been more reuse of the word “reuse” than practice of it, it’s undoubtedly the case that a major area of progress in our industry has involved enabling reuse. In the earliest systems, memory was so expensive that it was often necessary to save memory by reusing inline code. If those ten lines of assembly code were the same for one context as for a second, then the confines of limited memory required their reuse. Of course, over time, the minor distinctions between one context and another required flags to distinguish each case, and reuse in this manner deservedly acquired a poor reputation. The solution to this problem was the invention of the callable function.

Functions, in the mathematical sense of the word, are ideal for encouraging reuse because they transform their inputs into outputs without recourse to any kind of memory, or “state.” The square root function, for example, returns the same result for a given input every time. Mathematical functions lend themselves to reusable libraries for just this reason, and they also increase the granularity of reuse.

However, many functions, such as a payroll function, whose output depends on knowledge of previous deductions, employ stored data saved from one invocation to the next. The controlled use of such stored data increased the range of what could be done with reusable functions—more properly, subroutines—and libraries of these subroutines increased mightily in the 1960s and 1970s.

It quickly became apparent that there's value in sharing data between subroutines. The mechanism commonly chosen to implement this concept was a shared (global) data structure. Here swims the fly in this particular ointment: Just as the flags in shared inline code became a maintenance nightmare, so too did shared data structures. When several subroutines each have uncontrolled access to shared data, a change to a data structure in one subroutine leads to the need to change all the other subroutines to match. Thus was born the object.

Objects encapsulate a limited number of subroutines and the data structures on which they operate. By encapsulating data and subroutines into a single unit, the granularity of reuse is increased from the level of a single subroutine, with implicit interfaces over other (unnamed) subroutines, to a group of subroutines with explicit (named) interfaces over a limited group of subroutines. Objects enable reuse on a larger scale.

Objects are still small-scale, though, given the size of the systems we need to build. There is advantage in reusing collections of related objects together. An Account belongs to a Customer, for example; similarly, the object corresponding with a telephone call is conceptually linked to the circuit on which the call is made. In each case, these objects can, and should, be reused together, connected explicitly in the application.

A set of closely related objects, packaged together with a set of defined interfaces, form a component. A component enables reuse at a higher level, because the unit of reuse is larger. However, just as each of the previous stages in increasing granularity raised issues in its usage, so do components. This time, the problem derives from the interfaces. What happens if an interface changes? The answer, of course, is that we have to find each and every place where the interface is used, change it to use the new interface, test that new code, reintegrate, and then retest the system as a whole. A small change in the interface, therefore, leads to many changes in the code.

Dividing work across vertical problem areas and defining interfaces between these areas is also problematic. It's all too typical for a project team to begin

with an incomplete understanding of the problem and then divide the work involved in solving the problem amongst several development teams. The teams share defined interfaces, working to build components that can simply be plugged together at the end of the project. Of course, it doesn't usually work that way: Teams can have different understanding of the specifications of each of the components, and even the best-specified interface can be misinterpreted. Components, and their big brothers, frameworks, are rarely plug-and-play, and organizations can spend inordinate amounts of time writing "glue code" to stick components together properly.

The problem is even worse in systems engineering and hardware/software co-design because the teams don't even share a common language or a common development process. The result tends to be a meeting of the two sides in the lab, some months later, with incompatible hardware and software.

Dividing work into horizontal subject-matter areas, or domain models, such as bank, database, authorization, user interface, and so forth, exposes interfaces at the level of rules. "The persistent data of a class is stored as database tables" and "All updates must be authorized" and "Each operation that affects stored data must be confirmed" are all rules that apply uniformly between different domain models. Glue code can be produced automatically based on rules like these. Figure 1-2 illustrates how this progression increases the granularity of reuse.

Design-Time Interoperability

Even with these advances in the level of reuse, we nonetheless have a problem: There's still little reuse of applications.

Over and over, we see systems that are reimplementations of existing functionality built to make use of improved technology, and we see systems that are unable to reuse existing platforms because they've become interwoven with an existing application. Components and frameworks are helping, but there's still significantly more reuse of those closer to the machine. We see more reuse of databases and data servers—general services that rely on implementation technologies—than we see reuse of customer objects, which in turn rely on general services.

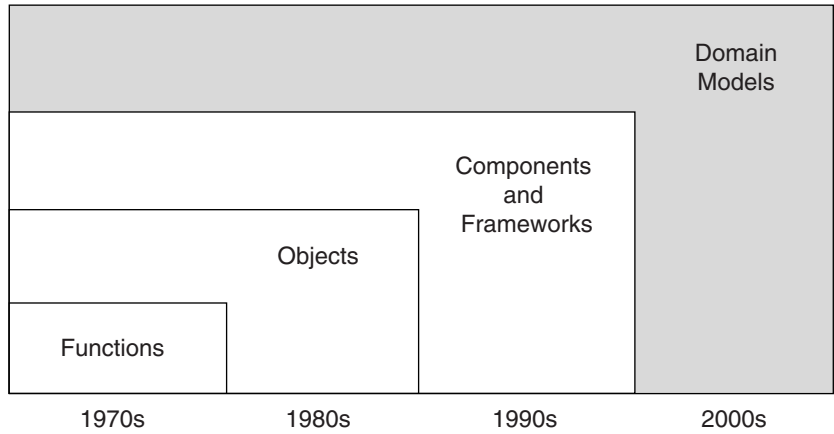


Figure 1-2 Raising the level of reuse

Figure 1-3 shows the overall effect. Each line between layers represents an opportunity for standardization to support run-time interoperability. Standards allow one layer to be replaced by a different implementation that conforms to the same standard. This is the value of interoperability: By defining a standard interface, we may replace one CORBA implementation with another, say, or one SQL database with another.

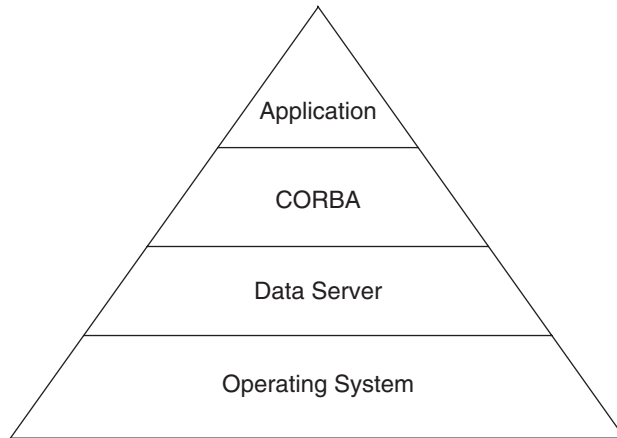


Figure 1-3 The difficulty of reusing applications

Standards and interoperability of this nature certainly help, but the problem still remains: What happens if the CORBA implementation you prefer relies on some operating system or database that you don't want? Tough. You're stuck, because each layer in the pyramid relies on all of the layers below it.

Moreover, components and frameworks may not fit together architecturally. This problem, dubbed *architectural mismatch* by David Garlan (1994), comes about when the several components and/or frameworks in a system have differing concepts about how the system fits together.

Here are some examples, moving from the concrete to the more abstract:

- Two components might each think they have sole control over a resource or device (a printer, for example).
- A component relies on an infrastructure that's completely different from another.
- One component thinks it must request updates, while another thinks it will be told about them.
- One component is event-driven, receiving one set of related data elements at a time, while another periodically updates *all* data elements, whether they're related or not.

In each of these cases, the problem is not merely one of interfaces, though often that's how the problem presents itself, but rather completely different concepts about the software architecture of the system.

Expressed abstractly, reuse at the code level is *multiplicative*, not additive. For example, if there are three possible operating systems, three possible data servers, and three possible CORBA implementations, there are 27 possible implementations ($3 \times 3 \times 3$).

The chances of the stars aligning so we have the right database, the right operating system, and so forth are relatively small ($1/27$), even though there are only ten components ($3 + 3 + 3$, plus one for the application).

The consequences of this problem are ghastly and gargantuan. Even as we increase the level of abstraction and the level of reuse, we'll continue to have difficulties as the number of layers increases, as it must as we come to tackle ever larger problems. The main reason is that once we've mixed the pieces of code together, it's impossibly difficult to reuse each of the parts, because each part relies so much on code that glues the pieces together—and glue makes everything it touches sticky.

To realize an *additive* solution, one that allows reuse of each layer independently of the others, we must glue layers together using mechanisms that are independent of the content of each layer. These mechanisms are *bridges* between layers, which are expressed as a system of mappings between elements in the layer. Bridges localize the interfaces so that an interface can be changed and subsequently propagated through the code.

Relying on reuse of code, no matter how chunky that code is, addresses only a part of the problem. The dependencies between the layers must be externalized and added in *only when the system is deployed*. The glue must be mixed and applied only at the last moment. Each model is now a reusable, stand-alone asset, not an expense.

Model-driven architecture, then, imposes the system's architecture only at the last moment. In other words, by deferring the gluing of the layers together and combining models at the last (design) minute, model-driven architecture enables *design-time interoperability*.

Models as Assets

Some years ago, one of us was working with a large telecommunications company that was implementing a level-four protocol stack not once, but three times. There were three groups, each in a different part of the U.S., each building essentially the same system. As it turned out, each team was working a slightly different subset on top of different technologies (operating systems, languages, and the like) for different markets. We were able to bring these groups together somewhat, but the reuse we achieved was limited to concepts as expressed informally through the models. Geographical distribution, divergent goals, and just plain politics resulted in three almost completely separate projects.

The cost of building systems this way is enormous. The same system, or a simple subset of it, was implemented with three teams, which tripled the costs. Three times as much code was produced as was required—and that code was then added to the pile of code the company needed to maintain over time.

In short, software is an expense. And, as we discussed in the previous section, reuse on the application level is often prohibitively difficult. Contrast this situation with the vision promulgated by MDA:

1. Take a model of the protocol stack off the shelf.
2. Subset the model as necessary.
3. Take models of the implementation technologies off the shelf.
4. Describe how the models are to be linked.
5. Generate the system.

When it comes time to change the application, we make the changes in the application model and leave the models of the implementation technologies alone. When we need to retarget an application to a different implementation environment, we select the models for the new environment and regenerate. There's no need to modify the application models. Costs are lower; productivity is higher, based on increased reuse of models; maintenance is cheaper—and each new model that gets built is an asset that can be subsequently reused.

The cost of building and maintaining systems this way is significantly lower. The incremental cost resides primarily in selecting the appropriate models and linking them together. The models themselves need to be constructed, of course, but once they're complete, they have greater longevity than code because they evolve independently of other models. In other words, they become corporate *assets*.

This is *not* just a vision: Systems are being built this way today. But not many systems, unfortunately—most folk are stuck pushing bits in Java or something else. The issue now is to increase the rate of adoption, which we hope will happen as people gain a rich understanding of MDA, starting with Chapter 2.

