Model–driven engineering (MDE) is an approach to software development where mod– els rather than programs are the principal outputs of the development process (Kent, 2002; Schmidt, 2006). The programs that execute on a hardware/software platform are then generated automatically from the models. Proponents of MDE argue that this raises the level of abstraction in software engineering so that engineers no longer have to be concerned with programming language details or the specifics of execution platforms. Model–driven engineering has its roots in model–driven architecture (MDA) which was proposed by the Object Management Group (OMG) in 2001 as a new software development paradigm. Model–driven engineering and model–driven architecture are often seen as the same thing. However, I think that MDE has a wider scope than MDA. As I discuss later in this section, MDA focuses on the design and implementa– tion stages of software development whereas MDE is concerned with all aspects of the software engineering process. Therefore, topics such as model–based require– ments engineering, software processes for model–based development, and model– based testing are part of MDE but not, currently, part of MDA. Although MDA has been in use since 2001, model–based engineering is still at an early stage of development and it is unclear whether or not it will have a significant effect on software engineering practice. The main arguments for and against MDE are: 1. For MDE Model–based engineering allows engineers to think about systems at a high level of abstraction, without concern for the details of their implementa– tion. This reduces the likelihood of errors, speeds up the design and implemen– tation process, and allows for the creation of reusable, platform–independent application models. By using powerful tools, system implementations can be generated for different platforms from the same model. Therefore, to adapt the system to some new platform technology, it is only necessary to write a transla– tor for that platform. When this is available, all platform–independent models can be rapidly rehosted on the new platform. 2. Against MDE As I discussed earlier in this chapter, models are a good way of facilitating discussions about a software design. However, it does not always follow that the abstractions that are supported by the model are the right abstrac– tions for implementation. So, you may create informal design models but then go on to implement the system using an off–the–shelf, configurable package. Furthermore, the arguments for platform independence are only valid for large long–lifetime systems where the platforms become obsolete during a system's lifetime. However, for this class of systems, we know that implementation is not the major problem—requirements engineering, security and dependability, inte– gration with legacy systems, and testing are more significant. There have been significant MDE success stories reported by the OMG on their Web pages (www.omg.org/mda/products_success.htm) and the approach is used within large companies such as IBM and Siemens. The techniques have been used suc– cessfully in the development of large, long–lifetime software systems such as air traffic management systems. Nevertheless, at the time of writing, model–driven approaches are not widely used for software engineering. Like formal methods of software engi– neering, which I discuss in Chapter 12, I believe that MDE is an important develop– ment. However, as is also the case with formal methods, it is not clear whether the costs and risks of model–driven approaches outweigh the possible benefits. 5.5.1 Model–driven architecture Model–driven architecture (Kleppe, et al., 2003; Mellor et al., 2004; Stahl and Voelter, 2006) is a model–focused approach to software design and implementation that uses a sub–set of UML

models to describe a system. Here, models at different levels of abstraction are created. From a high-level platform independent model it is possible, in principle, to generate a working program without manual intervention. The MDA method recommends that three types of abstract system model should be produced: 1. A computation independent model (CIM) that models the important domain abstractions used in the system. CIMs are sometimes called domain models. You may develop several different CIMs, reflecting different views of the sys- tem. For example, there may be a security CIM in which you identify important security abstractions such as an asset and a role and a patient record CIM, in which you describe abstractions such as patients, consultations, etc. 2. A platform independent model (PIM) that models the operation of the system without reference to its implementation. The PIM is usually described using UML models that show the static system structure and how it responds to exter- nal and internal events. 3. Platform specific models (PSM) which are transformations of the platform-independent model with a separate PSM for each application platform. In principle, there may be layers of PSM, with each layer adding some platform- specific detail. So, the first-level PSM could be middleware-specific but database independent. When a specific database has been chosen, a database- specific PSM can then be generated. As I have said, transformations between these models may be defined and applied automatically by software tools. This is illustrated in Figure 5.19, which also shows a final level of automatic transformation. A transformation is applied to the PSM to generate executable code that runs on the designated software platform. At the time of writing, automatic CIM to PIM translation is still at the research pro- totype stage. It is unlikely that completely automated translation tools will be available in the near future. Human intervention, indicated by a stick figure in Figure 5.19, will be needed for the foreseeable future. CIMs are related and part of the translation process may involve linking concepts in different CIMs. For example, the concept of a role in a security CIM may be mapped onto the concept of a staff member in a hospital CIM. Mellor and Balcer (2002) give the name 'bridges' to the information that sup- ports mapping from one CIM to another. The translation of PIMs to PSMs is more mature and several commercial tools are available that provide translators from PIMs to common platforms such as Java and J2EE. These rely on an extensive library of platform-specific rules and patterns to convert the PIM to the PSM. There may be several PSMs for each PIM in the system. If a software system is intended to run on different platforms (e.g., J2EE and .NET), then it is only necessary to maintain the PIM. The PSMs for each platform are auto- matically generated. This is illustrated in Figure 5.20. Although MDA-support tools include platform-specific translators, it is often the case that these will only offer partial support for the translation from PIMs to PSMs. In the vast majority of cases, the execution environment for a system is more than the standard execution platform (e.g., J2EE, .NET, etc.). It also includes other application systems, application libraries that are specific to a company, and user interface libraries. As these vary significantly from one com- pany to another, standard tool support is not available. Therefore, when MDA is introduced, special purpose translators may have to be created that take the char- acteristics of the local environment into account. In some cases (e.g., for user interface generation), completely automated PIM to PSM translation may be impossible. There is an uneasy relationship between agile methods and model-driven archi- tecture. The notion of extensive up-front modeling contradicts the fundamental

ideas in the agile manifesto and I suspect that few agile developers feel comfortable with model-driven engineering. The developers of MDA claim that it is intended to sup- port an iterative approach to development and so can be used within agile methods (Mellor, et al., 2004). If transformations can be completely automated and a com- plete program generated from a PIM, then, in principle, MDA could be used in an agile development process as no separate coding would be required. However, as far as I am aware, there are no MDA tools that support practices such as regression test- ing and test-driven development. 5.5.2 Executable UML The fundamental notion behind model-driven engineering is that completely automated transformation of models to code should be possible. To achieve this, you have to be able to construct graphical models whose semantics are well defined. You also need a way of adding information to graphical models about the ways in which the operations defined in the model are implemented. This is possible using a subset of UML 2, called Executable UML or xUML (Mellor and Balcer, 2002). I don't have space here to describe the details of xUML, so I simply present a short overview of its main features. UML was designed as a language for supporting and documenting software design, not as a programming language. The designers of UML were not concerned with semantic details of the language but with its expressiveness. They introduced useful notions such as use case diagrams that help with the design but which are too informal to support execution. To create an executable sub-set of UML, the number of model types has therefore been dramatically reduced to three key model types: 1. Domain models identify the principal concerns in the system. These are defined using UML class diagrams that include objects, attributes, and associations. 2. Class models, in which classes are defined, along with their attributes and operations. 3. State models, in which a state diagram is associated with each class and is used to describe the lifecycle of the class. The dynamic behavior of the system may be specified declaratively using the object constraint language (OCL) or may be expressed using UML's action lan- guage. The action language is like a very high-level programming language where you can refer to objects and their attributes and specify actions to be carried out.