CHAPTER 1

Software Development Methodologies and Metamodelling

A major area of interest within the computing discipline of software engineering is that of software development methodologies. A methodology has several constituent parts including a full lifecycle process, a comprehensive set of concepts, a set of rules, heuristics and guidelines underpinning appropriate development techniques, a set of metrics, information on quality assurance, a set of coding and other organizational standards, and advice on reuse and project management [9]. In order to describe all these parts in a consistent and useful manner, we need some kind of formalism. The formalism chosen here is that of metamodelling.

We begin this book with an examination of what is meant by a methodology and how metamodelling can be useful in creating robust and effective methodology models. Later we describe how the basic metamodelling ideas can be used in various domains, such as for modelling work products and processes. We evaluate the current state-of-the-art in these and other application areas before introducing some advanced ideas and how they have led to the creation of international standards that support the development of industry-strength methodologies for use in professional, commercial software development endeavours.

1.1 WHAT IS A METHODOLOGY?

Dictionaries often provide two basic meanings for the word "methodology". On the one hand, a methodology is an approach to doing something; on the other hand, methodology is the study of methods. According to WordNet [17], a methodology is "the system of methods followed in a particular discipline", a method being "a way of doing something, esp. a systematic one". Also according to WordNet, methodology means "the branch of philosophy that analyzes the principles and procedures of inquiry in a particular discipline". Therefore, the two possible meanings of "methodology" are:

- The collection of methods followed in a particular discipline.
- The study of methods followed in a particular discipline.

These two accepted meanings are closely related but are clearly different. The first one refers to a *piece of information* that describes how things are done within a given discipline. The second one refers to the *activity* of studying how things are done within a given discipline. The first usage is common and widely accepted, although the second one is closer to the etymological origin of the word "methodology" since the *-ology* suffix means "study of", in Greek.

It is also worth emphasizing that the first accepted meaning, denoting a thing (a piece of information), corresponds to a countable noun: it is possible to speak about *one* methodology or *multiple* methodologies, and the article is always used in the singular (e.g. "this methodology is flawed"). The second accepted meaning, in contrast, corresponds to an uncountable noun, very much like "biology" or "archaeology": we usually omit the definite article ("biology is exciting" rather than "the biology is exciting"). Most uses of the word "methodology" in the realm of engineering pertain to the first meaning; it is very common to hear people speaking about *this* or *that* methodology, but extremely uncommon to hear people saying things like "methodology is an exciting area to work in" (this would sound much better with any other "-ology" noun). This is a grammatical reason that supports the first meaning of "methodology" better than the second.

Also, the first meaning relates the terms "methodology" and "method" by a whole–part relationship (i.e. a methodology is a system or collection of methods), whereas the relationship implicit in the second meaning is much more complex (methods are studied by methodology). Interestingly, most uses of "method" and "methodology" in engineering tend to blur the semantic differences between them: a method is a systematic way of doing something and a methodology is a collection of methods, which, appealing to common sense, also defines a way of doing something. This fits well with the first meaning of "methodology" and is a semantic reason that supports the first meaning of "methodology" better than the second.

Finally, the second meaning of "methodology" pertains to philosophy and is rarely used without a connection to this field. Although modelling and metamodelling have strong connections to cognitive science and psychology, we believe that the three reasons here outlined are important enough as to encourage the adoption of the first meaning of "methodology" as far as engineering disciplines are concerned. Therefore, we can say that:

A **methodology** is a systematic way of doing things in a particular discipline.

This definition places the concept of "methodology" very close to that of "method"; as we said above, a collection of ways to do things ("methods", according to the dictionary) is also, using common sense, a way to do things. This means that, for practical purposes, and as far as this book is concerned:

A **method** is a methodology: the two terms are synonymous.

1.1.1 Further Characterization

Let us dig deeper into the definition of what a methodology is. We will do this by analyzing the phrase "a systematic way of doing things in a particular discipline".

To start with, and central to the definition, a methodology is a *way*. In other words, it is a manner, a means, a course of conduct. This means that no methodology is an end in itself but is a means to an end. It also means that, being *a* manner, it is arguable that *other* alternative manners also exist.

In addition, a methodology is *systematic*, i.e. orderly, planned and, at least to a certain extent, predictable. If we assume that methodologies are often shared by multiple individuals, this means that subjectivity must be reduced to a minimum in methodologies; otherwise, they wouldn't be systematic in an objective way.

Furthermore, a methodology is a way of *doing things*. This means that methodologies can be applied to change the state of the world that surrounds us (i.e. to *do* things), becoming the cause of effects that should be observable. If this were not so, the methodology wouldn't be *doing* things. Also, the things done by use of a methodology are the persistent outcomes of its application; the methodology is applied in order to obtain these outcomes. In other words, the things done by a methodology encompass the purpose of the methodology as a whole. The methodology is used because some community pursues that purpose.

Finally, any given methodology pertains to a *particular discipline*. In other words, it is focussed on a particular domain of knowledge and, consequently, works within a particular conceptual environment. In this book, our focus is the discipline of software engineering i.e. building a software application that meets the user's stated requirements.

As a summary, we can say that methodologies are used by communities that work in well-defined fields in order to produce persistent outcomes in their environment in an orderly and predictable fashion. For example, a team of software developers (the community) may use Extreme Programming [2] (a methodology) in the field of information systems development in order to produce a working system (the persistent outcome) by following some techniques, such as "pair programming" and "test first", that are known to render good results (orderliness and predictability).

Since this book is about software development methodologies, we assume throughout that the well-defined field where methodologies will work is software engineering. It can be argued that most of the methodological knowledge in this area is also valid in related areas, such as systems engineering or even business-process engineering; we believe so but, in this book, we assume from now on that the methodologies we are discussing occur in the realm of software engineering. As a consequence, the community that uses a methodology is always assumed to be a team of software engineers or developers, and the persistent outcome that is pursued by the application of a methodology is assumed to be a software system, either a new one or a modification of a pre-existing one. Again, it can be argued that most of the knowledge related to the development of software systems can also be applied to the development and delivery of related services and even hardware.

There is one last issue to be considered before we can claim to have a complete characterization of software development methodologies. We have agreed that methodologies are *used* by software engineers in order to produce working systems; this means that methodologies must be *usable* by software engineers, with regard to both their content and form. This is the topic of the next section.

1.1.2 One Size Does Not Fit All

We believe that software development methodologies must be *useful*. If they are not, they will not be used and we will be wasting our time. Too often methodology books spend most of their life gathering dust on a shelf rather than being read, studied and annotated. For methodologies to be useful, we need to determine who their users are and focus on their needs. As we explained in the previous section, software engineers use methodologies in order to obtain software systems; these systems, however, can be of many kinds: embedded systems, database-oriented information systems, web applications, thick-client desktop applications, etc. Also, the teams and their context may be of very different sizes, hierarchical organizations, geographical distributions, skill sets and cultures. Other project parameters, such as calendar or budget latitude, requirements volatility and customer availability must also be considered. Given the large number of variables, it is clear that no well-defined set of needs can be clearly defined, at a useful level of detail, for the whole of the user community. The three major aspects (organization, project and product) can vary so much that the well-known adage "one size fits all" has often been reversed to convey that no single methodology can be devised so that it provides a useful systematic way of doing things in all software engineering endeavours [5].

Therefore, methodologies must be purpose-fit, i.e. adapted to the particular characteristics of the anticipated scenarios of usage. Of course, some compromises can be made; for example, it is easier to devise a methodology for the development of dynamic websites by co-located teams, regardless of the team size, than a methodology for the development of web sites, regardless of team distribution *and* size. As usual in engineering, the more specific the tool is, the more effective it can be – at the expense of constraining the range of its applicability.

There are two major ways of obtaining purpose-fit methodologies. The first is usually called *tailoring*, and is based on the adaptation of a pre-existing generic methodology to the particular needs of a user. "Tailoring" means adaptation or customization; this implies that some generic, template-like product must exist *a priori*, from which the final, purpose-fit product is obtained, e.g. [1]. Tailoring assumes that the fixed parts of a methodology can be clearly separated from the variable parts, because the template methodology that serves as input to the tailoring process must be defined without any specific knowledge of the properties of its future users. Often, this means abstracting out template elements and compromising on solutions that are either too abstract or, if concrete, out of scope. If a template element is very abstract, the tailoring process will have to make it more concrete by taking into account the needs of the user; if, on the contrary, a template element is not abstract but directly usable, it may well fall out of scope, since it will certainly respond to the needs of some users but not of others. As a result, the final methodology will, in both cases, be less than optimal. Also, there is a tension between the need to maximize tailorability, which pushes in the direction of abstracting out and making most aspects variable rather than fixed, and the need to provide a readily usable product, which pushes in the direction of making it concrete enough to be useful without modification and making most aspects fixed rather than variable. Tailoring is considered a viable approach to methodology development by many authors, being used by well-known approaches such as RUP [10]. However, each approach that uses tailoring as a purpose-fit mechanism must commit to a fixed compromise between abstraction and utility, which, once again, defines a supposedly universally applicable solution that should fit all.

The second possible approach to obtaining a purpose-fit methodology is called *method engineering*[1] [3; 11; 18; 19] and is based on the idea that methodologies can be dynamically assembled from pre-existing components according to the user's needs. With this approach, no templates exist; the only universals that are considered are fine-grained, self-contained *method components* that have been demonstrated to work well in different situations. A method component, therefore, is a small, reusable piece of information about a very specific aspect of a methodology (such as a particular task to be performed or a particular product to be used) that can be stored in a database to be selected and incorporated into a methodology as necessary. This approach benefits from a large degree of reuse, since method components are reused each time they are selected for a particular methodology and are evaluated when the methodology is used, thus establishing a feedback loop that can be used to alter the specification of a method component from the data obtained during its real-world usage (Figure 1.1). At the same time, this approach benefits from the fact that no pre-existing overall fixed parts exist, and therefore the assumptions that need to be made about the users of the methodologies are minimal.

An architectural simile is useful to understand the differences between these two approaches. The tailoring approach is akin to buying a house and refurbishing it by knocking down some walls and building a few extensions; the resulting house will be better suited to its owners than the original one but will always be constrained by the original assumptions and overall design. The method engineering approach, on the other hand, is analogous to defining what kind of house you would like to have, drafting some blueprints, obtaining the components (such as doors, windows, bricks and tiles) and building the house yourself. Although this is likely to be a more time-consuming task, the outcome will be optimally adjusted to your needs.

In this book, we adopt a method engineering approach because of its evident advantages. This approach, however, emphasizes an additional aspect of methodologies that must be considered carefully: if methodologies are to be

[1]More accurately, *situational method engineering*. However, for the sake of simplicity, in this book we use the shorter "method engineering" (ME).

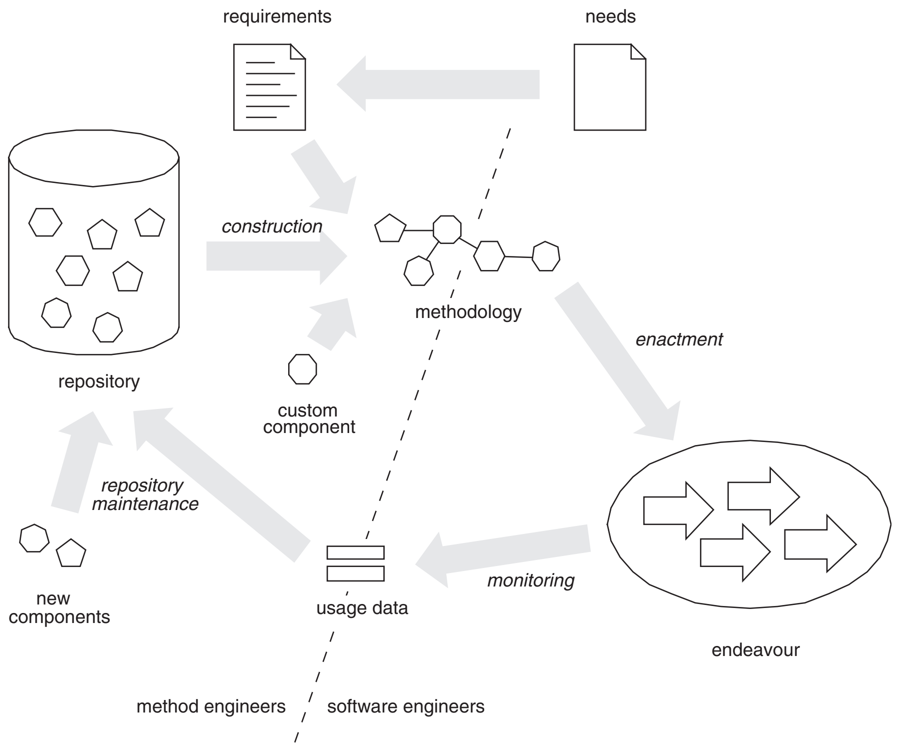

Figure 1.1: Method engineering at work: the dashed line separates the domains of method engineers and software engineers; processes (such as "enactment" or "monitoring") are indicated by arrows and labelled in italics; entities (such as "repository" or "endeavour") are depicted as shapes and labelled in roman typeface

created as complex assemblies of components, how is this done? Who should do it? What tools and techniques are available to do it? Following our previous example, an average citizen would not build a house herself, but would hire an architect; similarly, the role of method engineers, as the community that creates and maintain methodologies as first-order artefacts, must be introduced and its relation with the community of methodology users must be described.

1.1.3 Communities and Uses

There are two contrasting communities of people that interact with and utilize software development methodologies. Firstly, and most obviously, *software engineers* use methodologies in order to create software systems. Secondly, *method engineers* create and maintain methodologies according to the needs of software engineers. Notice that the concept of a method engineer, i.e. a person who designs and build methodologies according to some well-known needs, is not peculiar to the software engineering discipline; any other discipline that uses methodologies may utilize this concept as well.

Some vocabulary must be introduced. Method engineers are said to *construct* methodologies from method components. Software engineers are said to *enact* methodologies on *endeavours*. Enacting a methodology means, basically, applying it to solve a problem of the kind that it has been designed to solve. An endeavour is the organizational and psychological scenario where the methodology is applied. Typically, endeavours are visible as *projects*, where a project has as its purpose the development of a software system. However, non-project endeavours, such as infrastructure management or organizational support activities, are also common. These are endeavours but not, strictly speaking, projects, so we will use the broader term "endeavour" rather than "project".

An idealized method engineering scenario would appear as in Figure 1.1. The raw material that method engineers use is method components, usually stored in a specialized database often called a *repository* (some authors call this a *methodbase*). This repository contains tools that allow method engineers to browse and query the repository in order to locate the optimal components for a specific need. Tools are used to create new components, modify or delete existing components, transfer components between repositories and, in general, maintain repositories and components in good working order.

When a team of software developers needs a methodology, the method engineer negotiates with them a set of requirements for the methodology itself, requirements that are analogous to the requirements that a software engineer would gather and document from his customers and other stakeholders prior to constructing a software application. Thus, the method engineer gathers, analyzes, documents and validates a set of *methodological requirements* with the future users of the methodology and other stakeholders. These requirements are formalized and fed into some tool that assists the method engineer in the process of selecting the optimal components from the repository so that the best possible methodology is created. Not all the requested method components are necessarily available in the repository; sometimes, methodological requirements are so special or deal with such unexpected situations that some custom-made method components must be created by the method engineer for a specific endeavour. In this case, she may choose to incorporate them into the repository, so that future developments can benefit from them, or keep them "private" to the current effort. This has obvious parallels to the challenges of maintaining class libraries in object-oriented programming.

Once the methodology has been assembled, it is delivered to its users, in an appropriate form. By "appropriate" we mean a form that allows users to apply the methodology to solve problems (e.g. to develop software systems). A methodology that is delivered, for example, as a hard-copy document, can be used by software engineers only at a high cost, since they would have to read the document, make the correct interpretations and carry out the gathered instructions manually. A better delivery form would be an electronic specification of the instructions, which can be fed into an appropriate tool in order to be enacted in an endeavour. During the lifetime of the endeavour, the actual use of the methodology can be monitored by the same tools and the collected data sent back to the method engineers so that they can gain further understanding of how the selected method components operate in real-world conditions. From these data, they will be able to make the necessary adjustments to the method components, thus closing a loop. Thus, the quality both of the components and of the constructed methodology can increase in time. This means that this approach can provide an underpinning to the area of software engineering known as software process improvement (SPI).

From our discussion here and from Figure 1.1, it seems clear that the two communities involved in method engineering must be able to communicate fluently and as unequivocally as possible. Misunderstandings often arise when two different communities try to communicate, either because one community does not understand the meaning of some terms used by the other or because the first community uses the same term as the second one, but with a different meaning. An obvious way to combat this problem is to establish a shared, agreed-upon set of concepts and terms – often called an ontology – and use it systematically for every communication. It is worth emphasizing that not only must terms and concepts be standardized across communities, but also the mappings between them. This means that there is little value in agreeing upon the terms to be used and the major concepts in play if the relationships between the terms and the concepts they denote are not equally well defined.

An excellent example of terminology clashes is that of the ever-present term "task". For a software developer or a software project manager (both belonging to the software engineering community), a task is something like a period of time, with well-defined start and end times and a duration in which a certain kind of work is done. Roughly speaking, a task is a bar in a Gantt chart. However, when a method engineer is using, for example, OMG's SPEM 2.0 [16], a task (interchangeably referred to as a "task definition"; see [16, p. 87]) is something along the lines of a description of the work being performed by a role. Notice the word "description": for a software engineer, the task is the work actually being undertaken, while for a SPEM-oriented method engineer, the task is the *description* of what work must be done. These are different concepts, although the same term is being used.

A good test that can be used to verify whether two possibly different meanings of the same term are, in fact, different is to try to characterize each of the concepts using properties. For example, it is obvious that, for a software engineer, tasks have a duration. However, a SPEM-oriented method engineer would disagree; for her, tasks have a description. It is the instances of the task that have a duration. Relationships to other concepts are also useful in performing the test. For the SPEM method engineer, each task has a list of mandatory inputs and a list of optional inputs. The concept of "optional" is revealing, because it indicates that something is still to be decided at the endeavour level. In fact, for the software engineer, a task has a list of *effective* inputs, some of which would be considered mandatory by SPEM and others optional.

At the same time that we stress the differences between the two communities, we must also acknowledge the strong relationships that unite them. After all, a *software engineering task* is an instance of a *SPEM method engineering task* (task definition). As we saw with the mandatory versus optional inputs above, the meaning of "task" for the method engineer influences the meaning of "task" for the software engineer. In fact, it should be the other way around, since the software engineer is the user of the methodology and, as in any other engineering discipline, the user should be the one who determines the requirements of the product and therefore its overall semantics. In other words, the concepts used by method engineers to describe the methodologies that they will construct for software engineers must be chosen, shaped and adjusted according to the needs of the latter.

Both communities need to be able to describe the particular aspects of a software development methodology that they require. The software engineer needs to be able to access advice on what to do, how to do it, when to do it and who to do it. This advice may be available electronically or on paper. In both cases, the advice is in the form of a written description of the methodological element. For instance, it may be a description of the task of constructing a class model or how to use a technique such as robustness analysis; it may be a description of the role of a producer such as a business analyst or a description of an interim work product such as an agent class diagram. In contrast, the method engineer needs to be able to describe in a formal manner the concepts underlying the notion of task, technique, producer and work product.

Both communities fulfil their needs by the use of models. However, since the models used by the method engineer are at a higher level of conceptualization (often called, somewhat inaccurately, a higher level of abstraction), we will denote this specific kind of model by the name “metamodel” (where *meta-* indicates something “further” or “beyond”).

In the next section of this chapter, we will introduce metamodelling and its characterization and needs in a summary fashion before giving an in-depth presentation of modelling and metamodelling basics in Chapter 2.

1.2 METAMODELLING NEEDS

Metamodelling is not an end in itself; it is a means to achieve a different end, namely, in the context of this book, the delivery of methodologies that are as useful as possible within any given set of constraints. The usefulness of a methodology is given by a combination of, at least, two elements: its content and its form. The content aspect refers to what the methodology says; for example, a methodology that prescribes a waterfall lifecycle for a four-person team used to agile development is unlikely to be highly useful. Similarly, an extremely agile and coding-oriented methodology would be of little utility to a large, formal organization that works with highly distributed teams of dozens of people. As we said, methodologies must be fit for purpose; adjusting the contents of the methodology to the characteristics of its users is essential for this. Since the characteristics of the users vary from time to time, from organization to organization and even from project to project, it is impossible to “freeze” methodological content; as we have discussed, one size does *not* fit all.

1.2.1 Language

There is a second aspect to utility: the form of the methodology. This means how the methodology is expressed, regardless of what it says. A linguistic simile can be used. The content of a methodology is like the meaning of what I utter; since I utter different sentences at different times, depending on myriad conditions, it is impossible to predict what I will say next. The form of a methodology, on the other hand, is like the language I use to utter my words. I can certainly change my language, but it is *possible* to fix a particular language and utter almost anything using it. Similarly, it is possible to choose what language we want to use to express methodologies, and express any necessary methodology using it. Agreeing on a language and “freezing” it has the advantage that all the stakeholders will know how to interpret what I say. Usually, the agreed-upon language is a language known by all the parties involved; it offers the ability to be able to express with richness of detail any foreseeable meaning. Of course, some languages are better than others at expressing very specific things; for example, some natural languages cannot express the difference between green and blue colours [12, pp. 330–34], whereas others are extremely rich in their colour-related vocabulary. This is why selecting the optimal language to express something is not a trivial task; as in the linguistic simile, the language in which a methodology is expressed can enhance or hinder the ability of people to understand it and, as a consequence, to use it [4].

The most immediate language used to express a methodology is natural language, e.g. English or Spanish. For example, the Catalysis approach [6] has been expressed using English for the most part. Natural language is readily understandable by humans but hardly intelligible by computers (at least not currently). This means that Catalysis can be easily assimilated by a human but only manipulated by a computer after tedious translation work. Natural language is highly ambiguous with some words having a multitude of meanings, even in one context. Natural language is also linear, meaning that it takes the form of a sequence of symbols. This means that complex, non-linear structures must be "serialized" and flattened into a sequential equivalent. For example, consider a bidimensional matrix of figures printed on a sheet of paper. The only way to dictate it to another person is to read it row by row or column by column; in either case, it is being serialized into a sequential form.

A methodology is a much more complex structure than a bidimensional matrix of figures; it is composed of numerous concepts that refer to each other in complicated ways. A methodology can be visualized as a mesh of nodes interconnected by arcs. It is possible to serialize it onto paper and describe it in natural language (as was done in the early object-oriented methodology books, for instance), but any non-local reference must be maintained by hand. For example, imagine that the chosen serialization strategy consists of listing the complete process specification first, followed by all of the product specification. Process elements will necessarily refer to product elements; then the necessary links, in the form of notes or citations, will need to be created. If a product element changes its name later, it will be necessary to search for all the references to it and update them accordingly. Also, completeness must be checked by hand. Are all the product elements referred to in the process section? Are all the product elements created and used by at least one process element? If not, there is a consistency defect in the methodology. With natural language, the only way to check this is by hand. Such maintenance of non-trivial methodologies is daunting.

As an alternative to natural language, a modelling language can be used. This idea may appear odd at first, but we must recall that a methodology is a systematic way of doing things, i.e. the specification of the work to be done. In other words, a methodology describes how a software development team is expected to behave during an endeavour. From this point of view, a methodology is a model of the future endeavours that are conceivably possible, like the blueprints of a building are a model of any future buildings that conform to them.

Methodologies are models (indeed, some authors refer to them as "process models") and, therefore, a suitable modelling language rather than a natural language can be used to express a methodology, gaining in accuracy, reducing ambiguity and gaining the possibility of expressing their complex, non-linear structures of information, with no need for serialization. The ideal modelling language used to express methodologies must be generic enough so that any conceivable methodology can be expressed but, at the same time, concrete enough so that major methodological concepts can be treated with specific semantics. For example, some authors have used the Unified Modeling Language (UML) [15] as a language to express methodologies. UML is a generic object-oriented language described by a number of class diagrams. By the very nature of object orientation, UML is multi-purpose – although at the same time limited to the set of concepts its authors chose to include. Almost anything that we can think of can be represented as a class. This is useful but, at the same time, means that representing something as a class says very little about that something. A domain-specific language (DSL), on the other hand, would have first-order language elements that correspond to the major concepts of the subjects being modelled. For example, a DSL for software development methodologies would probably include elements such as Task, Team and Product whereas a DSL for a software architecture model might focus instead on classes, patterns, components and quality evaluation.

1.2.2 What to Represent

If a methodology is to be expressed as a model of all the elements necessary to execute an endeavour, we need to study what an endeavour is made of in order to determine what aspects must be included in a methodology. Traditionally, three major aspects have been identified: processes, products and people [7].

The process aspect is concerned with the work that must be done. Usually, this is stated in terms of tasks, steps, activities, techniques, processes, breakdown elements or some other types of work-related elements. In general, these elements can be named, collectively, "work units". The product aspect, on the other hand, is concerned with the artefacts that must be produced or used during the endeavour. This is often expressed in terms of models, documents, hardware or software items, etc., all of them collectively named "work products". Finally, the people aspect is related to the organizational structures that actually perform the endeavour, usually in terms of people, roles, teams and tools. All of these elements can be grouped under the name "producers".

The relationships between these three major aspects (Figure 1.2) can be summarized in a single sentence: work products are the result of work units performed by producers. However, the true relationships that exist between these aspects are more rich and complex than this sentence may suggest. First of all, work products are not just the result of work units but can also be the starting point of work units. For example, a needs statement that acts as raw material from which requirements are specified and, eventually, a system is built, is a product that must be represented by a methodology, but is not created during the endeavour. From an intuitive point of view, we can say that:

A **work product** is an artefact of interest for the endeavour.

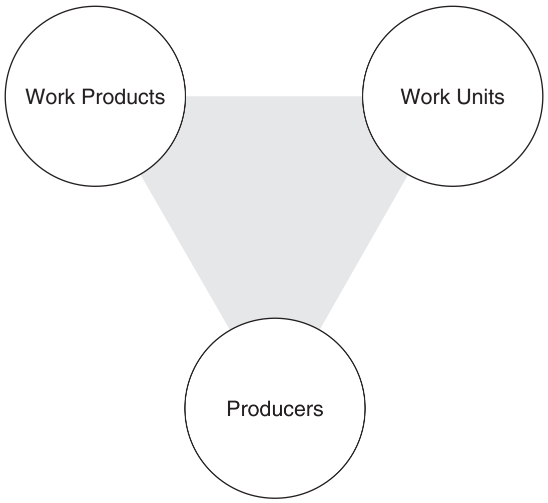

Figure 1.2: The three major aspects that any methodology must represent

This means that any artefact that is involved with the endeavour, either because it is created during it or because it is used (and possibly modified), is considered as a work product. Of course, the final system, which comprises the ultimate objective of the methodology, is also a work product.

Secondly, we said that work units are performed by producers. This is true but, from a project management perspective, we must acknowledge that work units may be defined well in advance before a producer actually performs them. For example, a project plan usually includes the tasks and activities that are expected to be performed within the project. Consequently, we can say that

A **work unit** is a job performed, or intended to be performed, within an endeavour.

We said that producers perform work units. Since work units can be planned in advance, it is better to define the relationship between producers and work units in terms of responsibilities rather than actual performance. In other words, we can say that

A **producer** is an agent that has the responsibility to execute work units.

This means that any person or team that has the responsibility to execute work units in an endeavour is considered a producer.

In summary, a methodology must describe work products, work units and producers. But, how should elements of these kinds be represented? The chosen representational form must satisfy two purposes. First of all, it must be able to supply the appropriate information to methodology users about the elements. For example, the specification of a task should include its name, its purpose and possibly a description of the steps to be taken. The specification of a producer, on the other hand, should include a name and a collection of responsibilities. Secondly, and in addition to information about each element, the representational form must be adequate so that methodology users can instantiate the methodology elements for use in their endeavour. In other words, most methodology elements need to serve as templates to create related endeavour elements. For example, the specification of a particular task kind must not only describe the tasks of such a kind but must also serve as a template from which actual tasks of that kind can be created in the endeavour. Similarly, a specification of a particular work product kind not only describes work products of that kind but must also be able to generate specific work products of that kind in the endeavour.

1.2.3 Summary of Metamodelling Needs

In this section we have explained that a methodology needs to be useful to its users. In order to achieve this, the following needs have been identified:

- The methodology contents must be purpose-fit, i.e. optimally adjusted to the users' needs, including organization, project and product aspects.
- The methodology must be expressed using a specialized modelling language, so that it is minimally ambiguous and easily processed by computers.

- The methodology must deal with three major aspects: work products, work units and producers, as well as the relationships between them.
- The methodology must describe the endeavour and also serve as a template so that the endeavour can be generated from it.

How should metamodelling help us address the needs described in the previous section? Let us analyze each need in turn.

With regard to methodology contents, metamodelling must ensure that organization, project and product requirements of the future users of the methodology are taken into account, and that the methodology contents are established according to them. Different requirements usually mean different solutions and, therefore, metamodelling must be able to deliver optimal methodologies for each project, domain or organization upon demand.

Regarding the modelling language, metamodelling must provide a language (generic or, more likely, domain-specific) that offers the necessary expressiveness so that any conceivable methodology can be appropriately specified. As few assumptions as possible should be made about the organization, project and product aspects of the future users when defining this language.

The modelling language must be able to describe at least the three major aspects of the methodology (kinds of work products, work units and producers). Also, the modelling language must be able to express concepts that, on the one hand, carry information (so that the methodology can be described) and, on the other hand, act as templates (so that endeavour elements can be generated from them).

A very important point must be made here. We have said that the modelling language used to represent the methodology should be able to express concepts such as kinds of work products, kinds of work units and kinds of producers. From a cognitive perspective, this means categorizing work products, work units and producers into well-defined groups, which, in turn, involves having a clear notion of the concepts of work product, work unit and producer. If the modelling language lacks the concept of work product, for example, how could it implement the concept of work product kind? Again, from a cognitive perspective, the only way to implement a categorization of a concept is based on that particular concept. The consequence of this is very clear: the modelling language must also be able to describe the endeavour-level, uncategorized counterparts of the methodology-level, categorized concepts that we have mentioned above.

1.3 WHAT IS METAMODELLING?

The term "metamodel" is, evidently, a qualified variant of "model". This suggests that metamodelling is a specific kind of modelling. In fact, metamodelling is the act and science of creating metamodels, which are a qualified variant of models. The difference between a regular model (one that is not a metamodel) and a metamodel is that the information represented by a metamodel is itself a model. The prefix "meta-", in Greek, means "higher" or "posterior". In epistemology and other branches of philosophy, the prefix "meta-" is often used to mean "about its own category"; in our case, "meta-" means that the model that we are building represents other models i.e. a metamodel is a model of models [8; 14]. This relationship is often described as paralleling the Type–Instance relationship. In other words, the metamodel (Type) and each of the models (Instances) are of "different stuff" and are described using different languages (natural or software engineering languages). Such type models therefore use classification abstraction – which leads to a metamodelling hierarchy.

One might indeed argue that modelling skills and techniques should be independent of the subject being modelled; a good modeller should be able to model an aeroplane, a business or a social group using the same skills and techniques (given the appropriate domain knowledge). If this is true, then the fact that a metamodel represents another model rather than "real-world" information should not be a reason to use a new term ("metamodelling") rather than the usual one (simply "modelling"). Following this line of reasoning, metamodelling should not be more special than aeroplane modelling or business modelling. However, there is one aspect that makes metamodelling a very special kind of modelling: the subject of metamodelling is models; in other words, the input and output artefacts of a metamodelling job are "made of the same stuff", i.e. they are of the same *type*. This gives metamodelling a recursive nature that makes it much more complex than other modelling areas in which the subject being modelled is of a different nature, making it especially tricky. As a simple but pervasive example, consider the Class class in UML. Parsing this phrase ("the Class class in UML") is already complex, since it involves a certain degree of recursiveness. The fact that UML contains a class named "Class" can bring up questions such as: is the Class class the same as the Class class in MOF [13]? What do we mean by "the same" here? Answering these questions is far from trivial.

The description of metamodelling and metamodels that we have provided so far is not especially related to methodologies; in fact, any model that provides the language to represent models is a metamodel. For example, any modelling language is a metamodel, according to this definition, since a modelling language, by definition, represents any possible model that can be expressed with it. The link to methodologies comes from the fact, explained earlier in this chapter, that methodologies are models of endeavours. If a methodology is a model, creating that methodology is modelling, whereas creating the language concepts used to describe the methodology is metamodelling. Consequently, the language that we use to express a methodology is a metamodel. Although the concepts of metamodel and metamodelling are not necessarily tied to methodologies (as specific kinds of model), this book is about software development methodologies; therefore we will focus on this particular application of metamodelling. According to this, we can say that:

A **metamodel** is a domain-specific language oriented towards the representation of software development methodologies and endeavours.

We can also say:

Metamodelling is the act and science of engineering metamodels.

The rest of this book describes the characteristics that a good metamodel must present and gives advice on constructing and using one. Since a metamodel is a specific kind of model, the first step is to introduce the necessary modelling and metamodelling concepts that, used as building blocks, will allow us to reason about models (and metamodels) with minimal ambiguity. This is the focus of Chapter 2.

1.4 SUMMARY

We have introduced the idea of a software development methodology as a systematic way of doing things in the discipline of software engineering. We have argued that it is fruitless to pursue the creation of a one-size-fits-all methodology for software development, leading to our acceptance of situational method engineering as currently the best way of proceeding. A single "frozen" or "branded" methodology can be regarded as one implicitly constructed using ME and then having the method components welded together permanently. Thus, our metamodelling discussion is equally applicable in this case also.

We have drawn attention to the needs of two software communities: the method engineer, or methodologist, who takes part in the creation of a methodology specific to the organization's situation and the software development team who use that methodology on real endeavours.

This leads us to enquire about the way in which such a methodological approach can be made rigorous. The answer currently used in the software engineering community is metamodels (although ontologies are likely to be used much more in the future). Metamodels can be used to underpin modelling languages, process models and methodology creation.

REFERENCES

1. Bajec, M., Vavpotič, D. & Krisper, M. 2007. Practice-driven approach for creating project-specific software development methods. *Information and Software Technology*. 49: 345–365.
2. Beck, K. 2000. *Extreme Programming Explained*. Upper Saddle River, NJ: Addison-Wesley.
3. Brinkkemper, S. 1996. Method engineering: engineering of information systems development methods and tools. *Inf. Software Technol.* 38(4): 275–280.
4. Carroll, J.B. (ed.). 1956 and 1997. *Language, Thought and Reality: Selected writings of Benjamin Lee Whorf*. Cambridge, MA: Technology Press of Massachusetts Institute of Technology.
5. Cockburn, A. 2000. Selecting a project's methodology. *IEEE Software*. 17(4): 64–71.
6. D'Souza, F.D. & Wills, A.C. 1999. *Objects, Components and Frameworks with UML: the Catalysis approach*. Object Technology Series, G. Booch, I. Jacobson & J. Rumbaugh (eds). Upper Saddle River, NJ: Addison-Wesley.
7. Firesmith, D.G. & Henderson-Sellers, B. 2002. *The OPEN Process Framework: An introduction*. The OPEN Series. London: Addison-Wesley.
8. Flatscher, R.G. 2002. Metamodeling in EIA/CDIF: meta-metamodel and metamodels. *ACM Trans. Modeling and Computer Simulation*. 12(4): 322–342.
9. Henderson-Sellers, B. 1995. Who needs an OO methodology anyway? *J. Obj.-Oriented Programming*. 8(6): 6–8.

10. Kruchten, P. 1999. *The Rational Unified Process: An introduction*. Reading, MA: Addison-Wesley.

11. Kumar, K. & Welke, R.J. 1992. Methodology Engineering: a proposal for situation-specific methodology construction. In: W.W. Cotterman & J.A. Senn (eds). *Challenges and Strategies for Research in Systems Development*. Chichester, UK: John Wiley & Sons 257–269.

12. Lakoff, G. 1987. *Women, Fire and Dangerous Things: what categories reveal about the mind*. University of Chicago Press.

13. OMG. 2002. formal/2002-04-03. *Meta Object Facility (MOF) Specification*, version 1.4. Object Management Group.

14. OMG. 2003. omg/03-06-01. *MDA Guide Version 1.0.1*. Object Management Group.

15. OMG. 2006. formal/05-07-05. *Unified Modeling Language Specification: Infrastructure*, version 2. Object Management Group.

16. OMG. 2007. ptc/07-03-03. *Software Process Engineering Metamodel Specification*, version 2.0. Object Management Group.

17. Princeton University Cognitive Science Laboratory. 2006. *WordNet*. http://wordnet.princeton.edu/. Accessed on 4 February 2007.

18. Ralyté, J. & Rolland, C. 2001. An approach for method engineering. *Procs 20th Int. Conf on Conceptual Modelling (ER2001)*, LNCS 2224. Berlin: Springer-Verlag. 471–484.

19. Tolvanen, J.-P., Rossi, M. & Liu, H. 1996. Method Engineering: current research directions and implications for future research. In: S. Brinkkemper, K. Lyytinen & R.J. Welke (eds). *Method Engineering: principles of method construction and tool support. Procs. IFIP TC8, WG8.1/8.2 Working Conference on Method Engineering, 26–28 August 1996, Atlanta, USA*. London: Chapman & Hall. 296–317.